Binary Exponentiation Method

Adrishikhar Chowdhury

Math Python C++ Java

Intuition

The problem of computing x^n can be solved efficiently using Exponentiation by Squaring, which reduces the number of multiplications by repeatedly squaring the base and halving the exponent. Instead of iterating n times, we reduce the exponent by half at each step. If the exponent is negative, we invert the base
x to $1/x$ and make the exponent positive.

Approach

Let me fix it properly so it appears cleanly when you copy and paste in Markdown or even rendered form without unnecessary formatting issues.

Approach

Here is the **corrected and properly formatted part** for the negative exponent:

- **If (n < 0)**: Convert x to $1/x$ (since $x^{-n} = 1/x^n$).
- **Change n to its absolute value |n|** by setting:

```
binform = -n
```

---

2. **Initialize the result variable `ans` :**

```
ans = 1.0  // This will store the result
```

3. **Loop for exponentiation by squaring:**

   - While `binform > 0` :
     - If `binform` is **odd**, multiply the current base `x` to `ans` :

       ```
       ans = ans * x
       ```

     - **Square the base `x` :**

       ```
       x = x * x
       ```

   - **Halve the exponent `binform`:**
     ```
     binform = binform / 2
     ```

4. **Return `ans`:**
   After the loop finishes, `ans` holds the result (x^n).

Complexity

- ## Time Complexity

  - The algorithm follows the **exponentiation by squaring** approach.
  - At each step, the exponent `binform` is halved, making the number of iterations proportional to logn.

Resulting Time Complexity: $O(\log n)$

Explanation:

- The loop runs while `binform > 0`, and `binform` is divided by 2 in each iteration.
- Therefore, for an exponent of size (n), it takes approximately:$\log_2 n$ iterations.

- Space complexity:
  Here's the **Space Complexity** part, properly formatted and explained:

Space Complexity

- The algorithm uses **constant extra space** regardless of the input size.
- Variables used:
  - `binform` (stores the modified exponent)
  - `ans` (stores the final result)
  - `x` (base value after transformations)
- No additional data structures (like arrays or recursive calls) are used.

Resulting Space Complexity: $O(1)$

Explanation:

The algorithm only maintains a few variables, so the memory usage does not grow with the size of the exponent n. Therefore, it has a constant space complexity of O(1)).