ĐẠI HỌC KHOA HỌC TỰ NHIÊN TPHCM
KHOA CÔNG NGHỆ THÔNG TIN

PHƯƠNG PHÁP TOÁN TRONG TIN HỌC VÀ GIẢI THUẬT

ĐỒ ÁN 3: LỖ HỔNG NGUYÊN TRONG PHÉP TOÁN $x \cdot n$

Nhóm 5

Họ và tên	**MSSV**
Phạm Thị Anh Đào	23C11003
Lê Minh Trí	24C11030
Dương Tiến Vinh	24C11034
Bùi Thế Vinh	24C11070
Đỗ Hoài Nam	24C12021
Trần Quang Duy	24C12027

Giảng viên hướng dẫn: PGS.TS. Trần Đan Thư

TP. Hồ Chí Minh, Tháng 03 Năm 2025

Nội dung đồ án 3: Tập trung vào phép toán $x \cdot n$ với x là số double và $n \in \mathbb{N}^+$ (nguyên dương) có khả năng lấy giá trị lớn

a) Tìm 3 trường hợp (x, n) có thể xác định rõ giá trị $x \cdot n$ bằng cách tính trên giấy ra kết quả chính xác, trong khi đó tính bằng Excel hay hàm nhân có sẵn ra kết quả sai lệch đáng kể ấn tượng

b) Cài đặt thuật toán lũy thừa nhanh và chạy thử ra kết quả đúng cho 3 ví dụ trên

c) Minh họa cho thuật toán tuyến tính $O(n)$ tính $x \cdot n$ rất chậm

Bài làm

1 Biểu diễn số thực theo chuẩn IEEE 754

1.1 Giới thiệu

Chuẩn IEEE 754 (IEEE Standard for Floating-Point Arithmetic) là một tiêu chuẩn quốc tế được sử dụng để biểu diễn số thực trong máy tính, cho phép thực hiện các phép tính số học chính xác và hiệu quả. Tiêu chuẩn này bao gồm nhiều định dạng, trong đó phổ biến nhất là **binary32** (single precision) và **binary64** (double precision).

1.2 Cấu Trúc Dữ Liệu

Một số thực trong IEEE 754 được biểu diễn theo công thức:

$$x = (-1)^{\text{Sign}} \times (1.\text{Mantissa}) \times 2^{(\text{Exponent}-\text{Bias})}$$

Các thành phần bao gồm:

- **Bit dấu (Sign)**: 1 bit, biểu diễn dấu của số (0 là số dương, 1 là số âm).
- **Số mũ (Exponent)**: 8 bit đối với binary32 hoặc 11 bit đối với binary64. Số mũ được mã hóa theo dạng bù (biased exponent).
- **Phần định trị (Mantissa/Fraction)**: 23 bit (binary32) hoặc 52 bit (binary64), lưu trữ phần thập phân của số được chuẩn hóa.
- **Giá Trị Bù (Bias)** Giá trị bù được sử dụng để dịch chuyển số mũ sang miền dương, giúp biểu diễn cả số mũ dương và âm. Công thức tính:

  $$\text{Bias} = 2^{n-1} - 1$$

  Trong đó n là số bit của phần số mũ. Cụ thể:

  – Đối với binary32: $Bias = 127$.
  – Đối với binary64: $Bias = 1023$.

1.3 Ví Dụ Minh Họa

Biểu diễn số -10.75 theo chuẩn IEEE 754 (binary32):
Bước 1: Chuyển sang nhị phân

$$-10.75 = -(1010.11)_2$$

Bước 2: Chuẩn hóa

$$-(1.01011)_2 \times 2^3$$

Bước 3: Mã hóa số mũ

$$\text{Exponent} = 3 + 127 = 130 = (10000010)_2$$

Bước 4: Mã hóa phần Mantissa Loại bỏ phần 1., ta được:

$$01011000000000000000000$$

Bước 5: Kết quả cuối cùng

$$1\ 10000010\ 01011000000000000000000$$

2 3 trường hợp (x, n) có thể xác định rõ giá trị $x \times n$ bằng cách tính trên giấy ra kết quả chính xác nhưng tính bằng phép nhân của ngôn ngữ lập trình có kết quả sai lệch đáng kể

Các Trường Hợp Sai Số

Bảng số liệu dưới đây trình bày ba trường hợp mà kết quả tính toán bằng phép nhân của ngôn ngữ lập trình có sai lệch đáng kể so với giá trị đúng cho kiểu dữ liệu `float` (32 bit):

x	n	Giá trị kỳ vọng	Giá trị thực tế
0.1	10^{12}	10^{11}	99,999,997,952
0.1	10^{13}	10^{12}	999,999,995,904
0.1	10^{14}	10^{13}	9,999,999,827,968

Tương tự như trên, bảng số liệu dưới đây chỉ ra ba trường hợp mà kết quả tính toán bằng phép nhân của ngôn ngữ lập trình có sai lệch đáng kể so với giá trị đúng cho kiểu dữ liệu `double` (64 bit):

x	n	Giá trị kỳ vọng	Giá trị thực tế
0.1	10^{24}	10^{23}	100,000,000,000,000,008,388,608
0.1	10^{25}	10^{24}	1,000,000,000,000,000,117,440,512
0.1	10^{26}	10^{25}	10,000,000,000,000,000,905,969,664

Nhận xét:

- Sai số xảy ra khi giá trị n rất lớn
- Giá trị n càng lớn thì sai số càng ấn tượng

Lý giải sai số

Ta xét trường hợp data type là `double` (64 bit) với x = 0.1 và n = 1,000,000,000,000,000,000,000,000. Các trường hợp khác có thể được chứng minh tương tự.

Như đã được nêu ở phần giới thiệu về tiêu chuẩn IEEE 754, data type `double` (64 bit) có cấu trúc bit như sau:

- 1 bit dấu (sign bit)
- 11 bit số mũ (exponent)
- 52 bit phần thập phân (mantisa)

Khi chuyển đổi x (0.1) sang dạng nhị phân, ta nhận thấy không thể biểu diễn chính xác giá trị của x với hữu hạn số bit:

$$00111111\ 10111001\ 10011001\ 10011001$$

$$10011001\ 10011001\ 10011001\ 10011001...$$

Do số `double` chỉ có thể lưu trữ tối đa 52 bit phần thập phân nên phần dư bị làm tròn, gây ra sai số.

$$00111111\,10111001\,10011001\,10011001$$

$$10011001\,10011001\,10011001\,10011010$$

Khi chuyển về dạng thập phân (base-10), số nhị phân trên sẽ có giá trị là:

$$1.00000000000000005551115123126 \times E^{-1}$$

Ta thấy được giá trị này không còn bằng chính xác 0.1 nữa. Tiếp tục với. n = 1,000,000,000,000,000,000,000,000 chuyển sang dạng nhị phân đầy đủ sẽ là:

$$11010011\,11000010\,00011011\,11001110\,11001100$$

$$11101101\,10100001\,00000000\,00000000\,00000000$$

Để lưu trữ n như một số `double`, cần chuyển n về dạng $1.(mantisa)^{(exponent)}$:

$$1.10100111100001000011011110011101100110011101101101000 01 \times 2^{79}$$

Phần thập phân của n có độ dài 79 bit. Vì số `double` chỉ có thể chứa tối đa 52 bit thập phân, phần thập phân của n sẽ được làm tròn và lược bỏ số bit dư. n khi được lưu trữ bằng `double` (64 bit) sẽ có dạng nhị phân như sau:

$$01000100\,11101010\,01111000\,01000011$$

$$01111001\,11011001\,10011101\,10110100$$

Khi chuyển giá trị trên về lại thập phân, sẽ có giá trị:

$$9.99999999999999983222784 \times E^{23}$$

Nhân x và n lại với nhau, ta được giá trị:

$$100,000,000,000,000,003,873,393.523125999906867742538448502784$$

Giá trị này cũng sẽ được tự động làm tròn khi lưu trữ bằng số `double`. Giá trị nhi phân sau sau khi làm tròn và sẵn sàng lưu trữ như một số `double` là:

$$01000100\,10110101\,00101101\,00000010$$

$$11000111\,11100001\,01001010\,11110111$$

Khi chuyển giá trị nhị phân trên về lại thập phân, sẽ có giá trị:

$$1.00000000000000008388608 \times E^{23} = 100,000,000,000,000,008,388,608$$

Đây là kết quả mà phép nhân của ngôn ngữ lập trình trả về!
Kết luận:

- Data type `float` và `double` không thích hợp để biểu diễn giá trị lớn vì hạn chế của tiêu chuẩn IEEE 754
- Cần lưu ý về khả năng sai số khi nhân hai số `double` giá trị lớn, dù kết quả cho ra không bị overflow.

3 Cài đặt thuật toán lũy thừa nhanh và chạy thử ra kết quả đúng cho 3 ví dụ trên)

3.1 Thuật toán nhân truyền thống

```
def linear_multiply(x, n):
    """
    Multiply a number by n using repeated addition.
    Time complexity: O(n)
    """
    result = 0
    for _ in range(n):
        result += x
    return result
```

Phép nhân $x \cdot n$ được hiểu là việc cộng số x với chính nó n lần. Ví dụ, $5 \cdot 3$ được tính bằng cách lấy $5 + 5 + 5 = 15$.

3.1.1

Các bước thực hiện Hàm `linear_multiply` triển khai nguyên lý này một cách trực tiếp:

a) Khởi tạo biến `result = 0` để lưu trữ kết quả phép nhân

b) Sử dụng vòng lặp for trong Python để lặp đúng n lần

c) Trong mỗi vòng lặp, cộng giá trị a vào biến result: `result += x`

d) Sau khi hoàn thành n vòng lặp, kết quả cuối cùng chính là $x \cdot n$

3.1.2 Độ phức tạp thuật toán

Thuật toán này có độ phức tạp $O(n)$ vì:

- Số lần lặp tỷ lệ thuận với giá trị của n
- Khi n tăng lên gấp đôi, thời gian thực hiện cũng tăng gấp đôi
- Mỗi vòng lặp chỉ thực hiện một phép cộng đơn giản $O(1)$

Điều này làm cho thuật toán trở nên kém hiệu quả với các giá trị n lớn, nhưng minh họa rõ ràng mối liên hệ giữa phép nhân và phép cộng lặp lại trong toán học cơ bản.

3.2 Thuật toán nhân nhanh

```
def fast_multiply(x, binary_n):
    """
    Multiply x number by n using binary multiplication
    technique.
    Time complexity: O(log(n))

    Parameters:
    x: number to multiply
    binary_n: string of '0's and '1's representing n in
    binary
    """
    result = 0

    # Get index for processing from right to left
```

```
idx = len(binary_n) - 1

# Process binary digits from right to left
while idx >= 0:
    bit = binary_n[idx]
    if bit == '1':
        result += x
    x += x  # Double x for next bit position
    idx -= 1
return result
```

3.2.1 Nguyên lý hoạt động của thuật toán

Thuật toán này tận dụng biểu diễn nhị phân của số nguyên để thực hiện phép nhân hiệu quả. Thay vì cộng số x với chính nó n lần như phương pháp nhân O(n), phương pháp này xử lý từng bit của số n và thực hiện các bước tương ứng.

3.2.2 Các bước thực hiện

Khi thực hiện `fast_multiply`, thuật toán hoạt động như sau:

a) Khởi tạo biến `result = 0` để lưu trữ kết quả phép nhân và chuyển n sang dạng binary: `binary_n`

b) Trong khi `len(binary_n) >= 0`, thuật toán:

b1) Kiểm tra bit cuối cùng của n

b2) Nếu bit là 1, cộng giá trị hiện tại của x vào *result*

b3) Nếu bit là 0, không cộng giá trị hiện tại của x vào *result*

b4) Nhân đôi giá trị của x (tương đương với phép dịch trái một bit)

Về bản chất, thuật toán này cộng $x \cdot 2^i$ vào kết quả cho mỗi vị trí bit thứ i mà bit trong n là 1.
Giải thích: Khi một số nguyên n được biểu diễn trong hệ nhị phân, mỗi bit đại diện cho một lũy thừa của 2. Ví dụ, số $n = 13$ được biểu diễn là 1101_2, có nghĩa là:

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Khi nhân x với n, chúng ta có thể viết:

$$\begin{aligned} x \cdot n = x \cdot 13 &= x \cdot (1 \cdot 2^3 + 1 \cdot 2^2 + 0 \times 2^1 + 1 \cdot 2^0) \\ &= x \cdot 1 \cdot 2^3 + x \cdot 1 \cdot 2^2 + x \cdot 0 \cdot 2^1 + x \cdot 1 \cdot 2^0 \end{aligned}$$

Do đó, chỉ có những bit là 1 mới cần cộng vào kết quả

4 Độ phức tạp thuật toán

Phương pháp này có độ phức tạp thời gian là $O(\log n)$ vì:

a) Vòng lặp chính chạy theo số bit trong biểu diễn nhị phân của n.

b) Một số nguyên n có khoảng $\log_2(n)$ bit trong biểu diễn nhị phân của nó.
Chứng minh:

Nếu có m bit, sẽ biểu diễn được các số từ 0 đến $2^m - 1$ trong hệ nhị phân (do mỗi bit có 2 trạng thái là 0 và 1, do đó m bit sẽ là 2^m trạng thái). Do đó, nếu biểu diễn một số n trong hệ nhị phân cần một số lượng bit m sao cho:

$$2^m - 1 \geq n$$

hay tương đương với:

$$m \geq \log_2(n+1).$$

Khi n đủ lớn, số lượng bit m cần để biểu diễn n là

$$m \geq \log_2(n+1) \approx \log_2(n)$$

Vậy một số nguyên n có khoảng $\log_2(n)$ bit trong biểu diễn nhị phân của nó.

c) Do có khoảng $\log_2(n)$ bit trong biểu diễn nhị phân của n, thuật toán sẽ lặp $\log_2(n)$ lần.

5 So sánh hiệu năng

So sánh hiệu năng (thời gian thực thi) và độ chính xác của hai thuật toán nhân: *Linear Multiply* và *Fast Multiply*, khi được thực hiện trên hai kiểu dữ liệu `float(32 bit)` (độ chính xác đơn) và `double(64 bit)` (độ chính xác kép). Dữ liệu được lấy từ kết quả benchmark với các giá trị n từ 10 đến 100,000,000,000 và $x = 0.1$

Bài so sánh này được thực hiện bằng ngôn ngữ lập trình C++.

5.1 Kết quả so sánh

Bảng 1: Kết quả benchmark cho kiểu dữ liệu `float`

n	Linear Result	Fast Result	Expected	Linear Error (%)	Fast Error (%)s	Linear Time (ms)	Fast Time (ms)
10	1.0000001192	1.0000000000	1.0	0.0000001192	0.0000000000	0.1250	0.2080
100	10.0000019073	10.0000000000	10.0	0.0000001907	0.0000000000	1.1670	0.2080
1,000	99.9990463257	100.0000000000	100.0	0.0000095367	0.0000000000	10.5000	0.2500
10,000	999.9028930664	1000.0000000000	1000.0	0.0000971069	0.0000000000	104.0000	0.2920
100,000	9998.5566406250	10000.0000000000	10000.0	0.0001443359	0.0000000000	1041.8330	0.3330
1,000,000	100958.3437500000	100000.0000000000	100000.0	0.0095834378	0.0000000000	9578.3330	0.6670
10,000,000	1087937.0000000000	1000000.0000000000	1000000.0	0.0879369974	0.0000000000	51153.9580	0.2500
100,000,000	2097152.0000000000	10000000.0000000000	10000000.0	0.7902848125	0.0000000000	358622.3750	0.5000
1,000,000,000	2097152.0000000000	100000000.0000000000	100000000.0	0.9790284634	0.0000000000	3605166.4580	0.2920
10,000,000,000	2097152.0000000000	1000000000.0000000000	1000000000.0	0.9979028702	0.0000000000	35451535.1670	0.3750
100,000,000,000	2097152.0000000000	10000000000.0000000000	10000000000.0	0.9997903109	0.0000000000	349873786.2500	0.5410

Bảng 2: Kết quả benchmark cho kiểu dữ liệu `double`

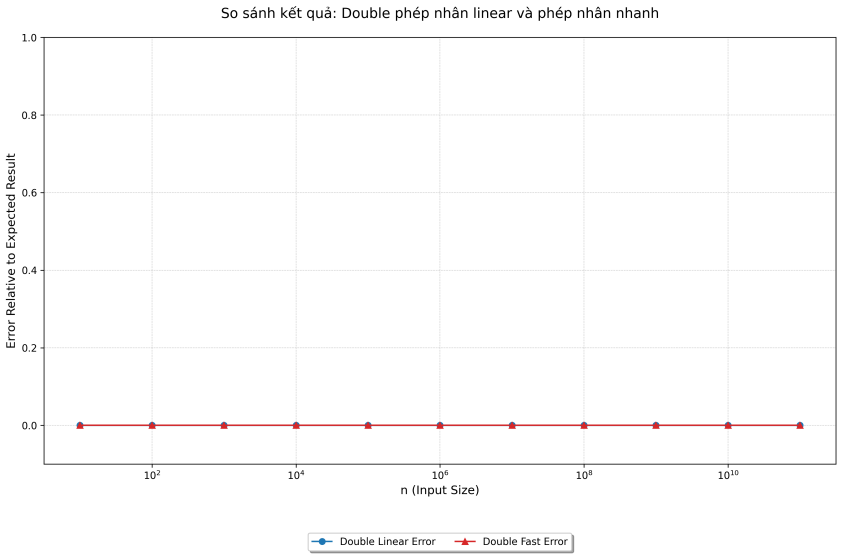
n	Linear Result	Fast Result	Expected	Linear Error	Fast Error	Linear Time (ms)	Fast Time (ms)
10	1.0000000000	1.0000000000	1.0	0.0000000000	0.0000000000	0.0000	0.0830
100	10.0000000000	10.0000000000	10.0	0.0000000000	0.0000000000	0.3750	0.0830
1,000	100.0000000000	100.0000000000	100.0	0.0000000000	0.0000000000	3.5000	0.0830
10,000	1000.0000000002	1000.0000000000	1000.0	0.0000000000	0.0000000000	35.2500	0.1250
100,000	10000.0000000188	10000.0000000000	10000.0	0.0000000000	0.0000000000	366.0840	0.1670
1,000,000	100000.0000013329	100000.0000000000	100000.0	0.0000000000	0.0000000000	3454.6670	0.1250
10,000,000	999999.9998389754	1000000.0000000000	1000000.0	0.0000000002	0.0000000000	35304.9160	0.3750
100,000,000	9999999.9811294507	10000000.0000000000	10000000.0	0.0000000019	0.0000000000	360873.2080	0.3750
1,000,000,000	99999998.7454178184	100000000.0000000000	100000000.0	0.0000000125	0.0000000000	3517857.0410	0.4160
10,000,000,000	1000000163.1244580746	1000000000.0000000000	1000000000.0	0.0000001631	0.0000000000	35160741.7080	0.4170
100,000,000,000	10000018871.6645336151	10000000000.0000000000	10000000000.0	0.0000018872	0.0000000000	349189730.7920	0.4170

5.2 So sánh độ chính xác

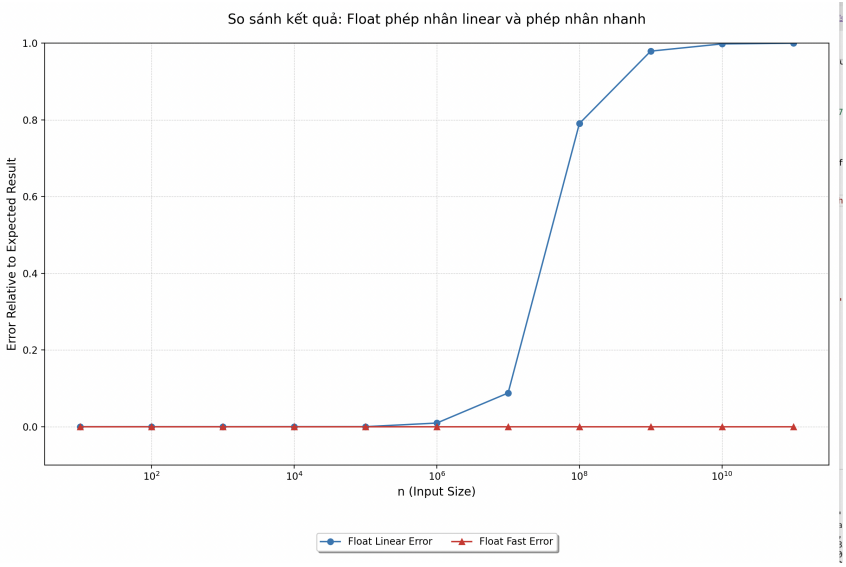
Tất cả các kết quả tính toán bằng *Fast Multiply* đều cho ra độ chính xác cao hơn đáng kể so với *Linear Multiply*. Cụ thể, trong khi *Fast Multiply* duy trì sai số gần như bằng 0 trong mọi trường hợp, *Linear Multiply* lại xảy ra sai số lớn khi n tăng cao, đặc biệt trên cả hai kiểu dữ liệu `float` và `double`. Nguyên nhân chủ yếu đến từ hai yếu tố sau:

Thứ nhất, với *Linear Multiply*, số lượng phép tính tăng tuyến tính theo n, dẫn đến việc làm tròn lặp đi lặp lại nhiều lần, gây tích lũy sai số đáng kể. Ngược lại, *Fast Multiply* chỉ yêu cầu $\log(n)$ lần tính toán, giúp giảm thiểu số lần làm tròn và hạn chế sai số phát sinh.

Thứ hai, trong kiểu dữ liệu `float`, *Linear Multiply* gặp hiện tượng "kẹt số", khi kết quả không tăng thêm sau khi cộng thêm x. Đặc biệt, từ $n \geq 100,000,000$, giá trị bị cố định ở $2097152 = 2^{21}$ do giới hạn độ chính xác của `float`. Trong khi đó, `double` với độ chính xác cao hơn vẫn chịu ảnh hưởng từ sai số tích lũy, dù ít nghiêm trọng hơn so với `float`.



Hình 1: Kết quả so sánh về độ chính xác double



Hình 2: Kết quả so sánh về độ chính xác float

5.3 So sánh thời gian thực thi

5.3.1 Độ phức tạp tính toán

Thuật toán *Linear Multiply* có độ phức tạp tính toán là $O(n)$, nghĩa là thời gian thực thi tăng tuyến tính theo kích thước đầu vào n. Trong khi đó, thuật toán *Fast Multiply* có độ phức tạp là $O(\log n)$, cho thấy thời gian thực thi chỉ tăng theo logarit của n, giúp nó xử lý các giá trị lớn hiệu quả hơn nhiều so với *Linear Multiply*.

5.3.2 Đối với kiểu dữ liệu `float`

Dựa trên kết quả benchmark:

- **Tổng thời gian thực thi:**
  - *Linear Multiply*: 389,351.013 ms (389.351013 giây)
  - *Fast Multiply*: 0.003958 ms (0.000003958 giây)
- **Tỷ lệ tốc độ:** *Linear Multiply* chậm hơn *Fast Multiply* khoảng $98,341,311$ lần. Sự khác biệt này phản ánh rõ ràng độ phức tạp $O(n)$ của *Linear Multiply*, khi thời gian tăng mạnh với n, trong khi $O(\log n)$ của *Fast Multiply* giữ thời gian gần như không đổi.
- **Thời gian trung bình mỗi phép toán:**
  - *Linear Multiply*: 35,395.5466 ms (35.3955466 giây)
  - *Fast Multiply*: 0.0003598 ms (gần bằng 0 giây)

Khi n tăng từ 10 lên 100,000,000,000, thời gian thực thi của *Linear Multiply* tăng từ 0.1250 ms lên 349,873,786.2500 ms (gấp khoảng 2.8 triệu lần), trong khi *Fast Multiply* chỉ tăng từ 0.2080 ms lên 0.5410 ms (gấp khoảng 2.6 lần), phù hợp với dự đoán từ $O(n)$ và $O(\log n)$.

5.3.3 Đối với kiểu dữ liệu `double`

Dựa trên kết quả benchmark:

- **Tổng thời gian thực thi:**
  - *Linear Multiply*: 388,268.367 ms (388.268367 giây)
  - *Fast Multiply*: 0.002666 ms (0.000002666 giây)
- **Tỷ lệ tốc độ:** *Linear Multiply* chậm hơn *Fast Multiply* khoảng $145,611,013$ lần. Tương tự như với `float`, độ phức tạp $O(n)$ khiến *Linear Multiply* kém hiệu quả hơn nhiều so với $O(\log n)$ của *Fast Multiply*.
- **Thời gian trung bình mỗi phép toán:**
  - *Linear Multiply*: 35,297.1243 ms (35.2971243 giây)
  - *Fast Multiply*: 0.0002424 ms (gần bằng 0 giây)

Với `double`, thời gian của *Linear Multiply* tăng từ 0.0000 ms ($n = 10$) lên 349,189,730.7920 ms ($n = 100$ tỷ), trong khi *Fast Multiply* chỉ tăng từ 0.0830 ms lên 0.4170 ms, một lần nữa minh chứng cho ưu thế của $O(\log n)$ so với $O(n)$.

6 Lý giải hiện tượng kẹt giá trị trong kiểu dữ liệu `float`

Dựa vào kết quả trong bảng 1, ta có thể thấy khi n đạt giá trị 100000000 thì kết quả không tăng mà đứng lại ở giá trị $2097152 = 2^{21}$. Phần này sẽ lý giải nguyên nhân của hiện tượng trên.

6.1 Cơ chế biểu diễn số thực trong chuẩn IEEE 754

Trong kiểu dữ liệu `float` (32 bit), các giá trị được biểu diễn theo chuẩn IEEE 754 với 1 bit dấu (sign), 8 bit mũ (exponent) và 23 bit phần định trị (mantissa). Khi giá trị đạt tới $2097152 = 2^{21}$, biểu diễn của nó trong chuẩn IEEE 754 là:

- **Dấu (Sign):** $+1$ (bit 0)
- **Mũ (Exponent):** 2^{21} (10010100)
- **Phần định trị (Mantissa):** 1.0 (bit ẩn 1, các bit còn lại là 0)

Cụ thể, dạng nhị phân của 2097152 là: `01001010 00000000 00000000 00000000`. Số lớn hơn liền kề trong biểu diễn `float` là: `01001010 00000000 00000000 00000001`, tương ứng với:

- **Dấu (Sign):** $+1$
- **Mũ (Exponent):** 2^{21}
- **Phần định trị (Mantissa):** $1 + 2^{-23} \approx 1 + 0.00000011920928955078125$

Khi chuyển sang hệ thập phân, giá trị này là 2097152.25. Khoảng cách giữa 2097152 và số lớn hơn gần nhất là 0.25, do giới hạn độ chính xác của 23 bit mantissa trong `float`.

Vì vậy, khi thực hiện phép tính $2097152 + 0.1 = 2097152.1$, kết quả nằm giữa 2097152 và 2097152.25. Theo quy tắc làm tròn của IEEE 754 (thường là làm tròn xuống hoặc về số gần nhất), 2097152.1 bị làm tròn về 2097152. Điều này giải thích tại sao trong benchmark với kiểu dữ liệu `float`, giá trị của *Linear Multiply* bị kẹt ở 2097152 khi $n \geq 100,000,000$.

Ngược lại, kiểu dữ liệu double có độ chính xác cao hơn khoảng cách giữa 2 số liên tiếp tại $n = 100,000,000$ cũng nhỏ hơn nên không gặp trường hợp này. Tuy nhiên, double sẽ gặp trường hợp tương tự ở n lớn hơn.

7 Benchmark các thư viện decimal

7.1 Giới thiệu

Phần này so sánh tốc độ giữa các thư viện hỗ trợ kiểu decimal trong các ngôn ngữ lập trình khác nhau nhằm khắc phục sai số khi tính toán số thực. Các thư viện được đánh giá bao gồm:

- **Javascript:**
  - `decimal.js`
  - `bignumber.js`
- **Python:**
  - `decimal`
  - `gmpy2`
  - `mpmath`
- **Golang:**
  - `ericlagergren/decimal`
  - `shopspring/decimal`

7.2 Phương pháp đo lường

Chúng tôi thực hiện benchmark với ba trường hợp tính toán:

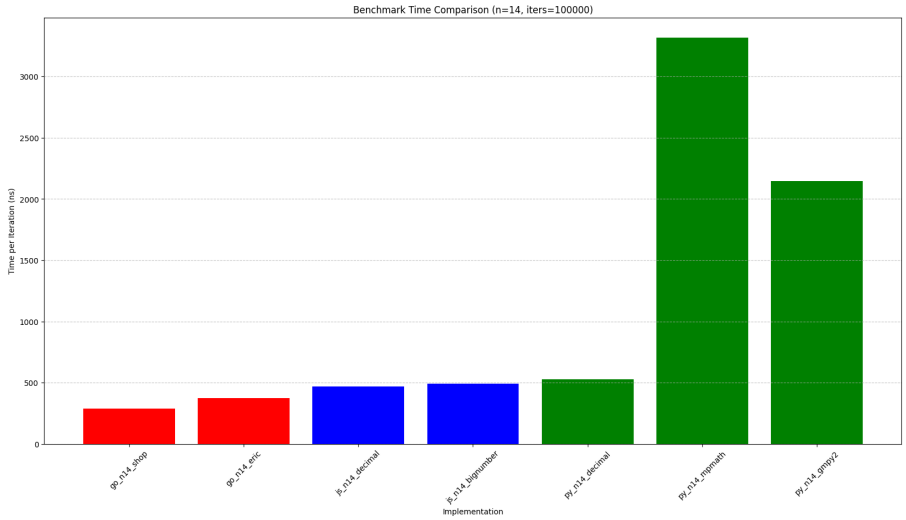
a) **Trường hợp 1:** $x = 0.1$, $n = 10^{14}$, vòng lặp: 100000, kết quả kỳ vọng: 10^{13}.

b) **Trường hợp 2:** $x = 0.1$, $n = 10^{26}$, vòng lặp: 100000, kết quả kỳ vọng: 10^{25}.

c) **Trường hợp 3:** $x = 98765432198765432198765432$, $n = 12345678912345678912345678$, vòng lặp: 100000, kết quả kỳ vọng: 12193263135650053159106843158177106934720316911263526 9.

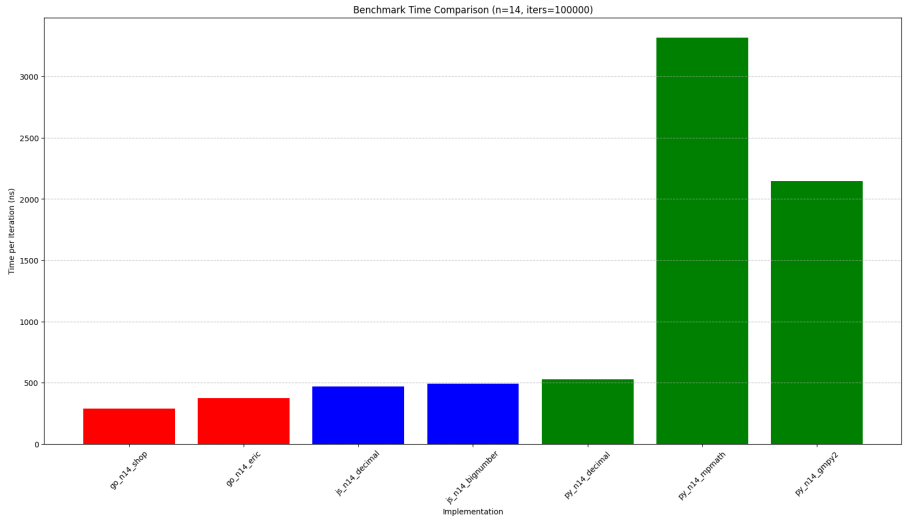
7.3 Kết quả benchmark

7.3.1 Trường hợp 1: $n = 10^{14}$

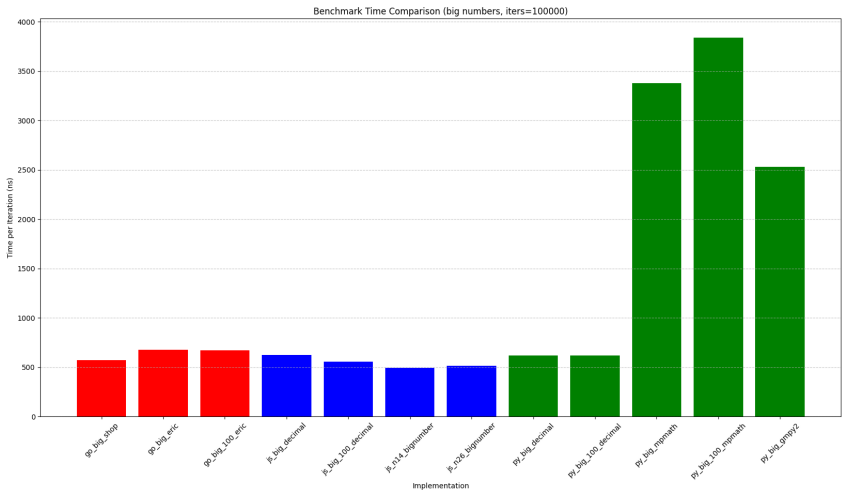


Hình 3: Kết quả benchmark với $n = 10^{14}$ và vòng lặp = 100000

7.3.2 Trường hợp 2: $n = 10^{26}$



Hình 4: Kết quả benchmark với $n = 10^{26}$ và vòng lặp = 100000



Hình 5: Kết quả benchmark với số nguyên lớn và vòng lặp = 100000

7.3.3 Trường hợp 3: Số nguyên lớn

7.4 Phân tích kết quả

Qua các biểu đồ Hình3, Hình4 và Hình5, ta nhận thấy:

- Các thư viện của Javascript (`decimal.js`, `bignumber.js`) và Golang (`ericlagergren/decimal`, `shopspring/decimal`) có hiệu suất tốt hơn so với Python (`decimal`, `gmpy2`, `mpmath`).
- Khi thay đổi hệ số precision (ví dụ: `go_big_100_eric`, `js_big_100_decimal`), thời gian thực thi tăng lên đáng kể, đặc biệt trong các trường hợp số lớn.

7.5 Kết luận

Các thư viện của Javascript và Golang phù hợp hơn cho các ứng dụng yêu cầu tính toán nhanh với số lớn, trong khi Python phù hợp hơn cho các phép tính cần độ chính xác cao nhưng không ưu tiên tốc độ.