MATH36031 Project 1

Wei Chung-Yu

October 2022

1 Introduction

In ancient Indian Vedic mathematics, dating back hundreds of years, various techniques were developed to simplify and accelerate arithmetic operations. One such mathematical technique known as the **Ūrdhva Tiryagbhyāṃ Sūtra**, which involves both vertical and crosswise calculations, and this technique is commonly employed for the multiplication of numbers $A * B$. Amongst practitioners of Vedic Mathematics, it is regarded as the simplest and most widely favored multiplication method.[1]

For this project, we aim to utilise an algorithm of Vedic multiplication, and mainly focus on two functions: `myrandi`[1] and `vedicmultiply`, which are intricately described in the subsequent sections of this report.

2 Problem solution

2.1 Task 1

Task 1 is asking to define a function called `myrandi` that generates a random string of digits of a specified length `n`. In **figure** 1(**Line1**), we take a single input argument `n` and return the generated random string of digits as `result`. (**Line3-5**) check whether the input `n` is a positive integer. It does this using a series of conditions:

- `~isnumeric(n)` checks if `n` is not numeric.
- `numel(n) ~= 1` checks if the number of elements in `n` $\neq 1$.
- `mod(n, 1) ~= 0` checks if the remainder of the division of `n` by $1 \neq 0$.
  This condition is used to determine n is an integer or not.

Any of these conditions is true means that the input `n` does not meet the requirements, and an error will be displayed indicating that the input must be a positive integer. (**Line7**) generates a random integer in the range from 1 to 9, ensuring that the leading digit is not 0. The generated digit is stored in the `leading_digit` variable. In (**Line9**), the `randi` function is used to generate random integers in the range from 0 to 9 (inclusive), and an array of `n-1` elements is created to represent the remaining digits. These digits are stored in the `rest_of_digits` variable. (**Line11**) concatenates the `leading_digit` and the `rest_of_digits` array to create a single array of digits by using the square brackets.

In (**Line12**), the code transforms the array of random digits into the format of string. By using the `sprintf` function [3], which change the integers in `digits` array as a string, concentrating them together. The format specifier `%d` is used to specify that the integers should be treated as decimal numbers, and

[1] In this report, all the code variables will be demonstrated in text-mode.

the final resulting string will be stored inside the `result` variable. In the specific testing example (**figure** 2), by input the integer 5, to set `n`=5, the randomly generated 5-digit output string is '74119'.

The full demonstration of the code, including output, can be found in the **figure** 1 and **figure** 2.

```
function result = myrandi(n)
    % Display error for non-positive integer
    if n <= 0 || ~isnumeric(n) || numel(n) ~= 1 || mod(n, 1) ~= 0
        error('Input must be a positive integer.');
    end
    % Generate the leading digit (not 0)
    leading_digit = randi([1, 9]);
    % Generate random digits for the rest of the string
    rest_of_digits = randi([0, 9], 1, n - 1);
    % Combine the leading digit and the rest of the digits without spaces
    digits = [leading_digit,rest_of_digits];
    result = sprintf('%d', digits);
end
```

Figure 1: MATLAB function code: `myrandi`.

```
n = myrandi(5)
% By input a positive integer n inside, generated as output string
% containing the n digits random integers, and each between 0 and 9

n =

    '74119'
```

Figure 2: Output of MATLAB function code: `myrandi`.

2.2 Task 2

In Task 2, we want to generalise the Vedic multiplication algorithm for multiplying two n-digit numbers. First assume the two numbers are A and B, where:

$$A = a_n a_{n-1} ... a_2 a_1.$$

$$B = b_n b_{n-1} ... b_2 b_1.$$

and a_i and b_j are digits based on the positions.

We use a formula to represent the summation S_k of the products of certain digit pairs of two n-digit numbers being multiplied, based on the position k [2]:

$$S_k = \sum_{i+j=k+1} a_i \times b_j. \tag{1}$$

For $1 \leq i \leq n$, $1 \leq j \leq n$ and $1 \leq k \leq 2n-1$.

Step 1. First Digit of Result:

$$S_1 = a_1 \times b_1.$$

Step 2. Next Digits: Move diagonally, and consider pairs of digits:

$$S_2 = a_2 \times b_1 + a_1 \times b_2.$$

$$S_3 = a_3 \times b_1 + a_2 \times b_2 + a_1 \times b_3.$$

$$\vdots$$

Step 3. Central Part (For numbers greater than 2 digits):

$$S_{i+j-1} = a_i \times b_1 + a_{i-1} \times b_2 + ... + a_1 \times b_j.$$

Step 4. Final Digits: As the algorithm continues, the number of terms starts decreasing again, similar to the reverse process of step 2.

Step 5. Last Digit of Result (Most Significant Digit):

$$S_{2n-1} = a_n \times b_n.$$

Step 6. Summation and Carry: Add the results from each step and carry over any overflow to the left as you would in a traditional multiplication. We can obtaine the final result by assembling the results from each step, starting with the first digit from step 1, followed by the digits from step 2, and so on.

Example: For two 2-digit numbers (n=2): $A = 32$, $B = 47$.

Step 1:

$$S_1 = a_1 \times b_1 = 2 \times 7 = 14.$$

(Resultant digit is 4, carry is 1).

Step 2:

$$S_2 = a_1 \times b_2 + a_2 \times b_1 = 2 \times 4 + 3 \times 7 = 8 + 21 = 29.$$

(Adding the carry from before: 29 + 1 = 30; Resultant digit is 0, new carry is 3).

Step 3:

$$S_3 = a_2 \times b_2 = 3 \times 4 = 12.$$

(Adding the carry from before: 12 + 3 = 15; Resultant digit is 5, new carry is 1, but since this is the final calculation for these numbers, we write down both digits).

Combine the resultant digits, we get the final result $A \times B = 1504$.

It is worth noting that the Vedic multiplication method is essentially a structured way of performing the traditional multiplication algorithm, but the visualization process is different. Some find the Vedic approach more intuitive and faster, especially when done mentally.

2.3 Task 3

The algorithm for Vedic multiplication has been demonstrated in section 2.2. In this context, we present the MATLAB function: `vedicmultiply` for this algorithm.

Figure 3 (**Line1**) declares a function named `vedicmultiply` that takes two integers a and b in string format, and returns a string c. (**Line2**) stores the maximum length of a and b in `n`. (**Line4**) and (**Line5**)

use `pad` function to add zeros to the left side of the input string until both strings are of the same length `n`. In (**Line6**), this line initialises a symbolic variable `result` to store the final multiplication result. Using `sym` is beneficial for very large numbers as it can handle numbers with arbitrary precision without losing accuracy. For the outer loop that iterates over digits of a in (**Line7**). It converts each digit in a to a double (floating point number) for arithmetic operations. For the inner loop that iterates over digits of b in (**Line9-11**). Each digit in b multiples with the current digit from a. In (**Line14-17**), the product of two digits might represent value in tens, hundreds, etc., based on their positions in the original numbers. To account for this, we calculate a shift value. The partial product is shifted (by multiplying with the appropriate power of 10) and then added to the cumulative `result`. At the end, (**Line19**) converts symbolic `result` variable converts back to a string format by using the `char` function.

In the specific output testing example (**Line3**, **figure**4). We set strings a and b :

$$a =' 143365'.$$

$$b =' 234545453'.$$

We want to find their product using the function `vedicmultiply`. The process involves multiplying each digit of a with every digit of b, considering the position shifts.

For simplicity, let's consider multiplying the last digit of a with every digit of b:

$$\begin{aligned} 5 \times 3 &= 15 \quad (5 \text{ from } a \text{ and the last digit of } b). \\ 5 \times 5 &= 25 \quad (5 \text{ from } a \text{ and the second last digit of } b). \\ &\vdots \\ 5 \times 2 &= 10 \quad (5 \text{ from } a \text{ and the first digit of } b). \end{aligned}$$

This process is repeated for every digit of a. After multiplying, products are shifted based on the position of the digits and summed up. The result of the `vedicmultiply` method for these two numbers is :

$$c =' 33625608869345'.$$

The full demonstration of the code, including output, can be found in the **figure** 3 and **figure** 4.

When we go back to see the relation between the `vedicmultiply` function and the algorithm in section 2.2.

In this example (**figure** 4), a is '143365' and b is '234545453', these are 6-digit and 9-digit numbers respectively. k will range from 1 to $14(6 + 9 - 1)$. For each value of k:

- Compute the partial products $a_i \times b_j$ where $i + j = k + 1$.
- Sum these partial products to get S_k by using formula 1.

Once all S_k values are computed, they are assembled (with appropriate carries) to get the final product.

- When $k = 1$: It is just the product of the units place of a and b.
- When $k = 2$: It is the sum of the product of the units place of a and the tens place of b with the product of the tens place of a and the units place of b.

And so on, until $k = 14$, which would involve the most significant digits. The result '33625608869345' is then obtained by collecting all these S_k values and managing the carries.

By methodically multiplying every digit of a with every digit of b, and placing them in their correct positions (thanks to the shifting), the function computes the product of a and b which matches with the traditional multiplication method. The nested loop structure in the function directly relates to the Vedic algorithm's principle of considering every digit combination in the multiplication process.

```
function c = vedicmultiply(a, b)
    n = max(length(a), length(b));            % Get the number of digits in the input numbers
    % Pad the input numbers with leading zeros to make them of equal length
    a = pad(a, n, 'left', '0');
    b = pad(b, n, 'left', '0');
    result = sym(0);                          % Initialize the result
    for i = 1:n                               % Perform the Vedic multiplication
        digit_a = str2double(a(i));
        for j = 1:n
            digit_b = str2double(b(j));
            partial_product = digit_a * digit_b;
            % Shift the partial product to the left based on the positions of the digits
            % in the input numbers
            shift = (n - i) + (n - j);
            partial_product = partial_product * sym(10^shift);
            result = result + partial_product; % Add the partial product to the result
        end
    end
    c = char(result);                         % Convert the result to a string
end
```

Figure 3: MATLAB function code: `vedicmultiply`.

```
a = '143365';
b = '234545453';
c = vedicmultiply(a,b)

c =

    '33625608869345'
```

Figure 4: Output of MATLAB function code: `vedicmultiply`.

2.4 Task 4

We have already demonstrated two functions: `myrandi` and `vedicmultiply` in **figure** 1 and **figure** 3 respectively. In this context, we generate two large random numbers: a with 30 digits and b with 40 digits. Then, with the `vedicmultiply` function, we efficiently compute and present their product in the output. The full demonstration of the code, including output, can be found in the **figure** 5.

Figure 5 (**Line1-2**), we use function `myrandi` generate two random input strings for function `vedicmultiply`.

$$a =' 382642825609132600080066017327'(30 - digit).$$

$$b =' 722715680337118637238029800195930886577 9'(40 - digit).$$

(**Line3**) apply the function `vedicmultiply` with two inputs a and b, we received the output

$$ab =' 2765419700362217091952928383747807930261048503348310798996406131352733'.$$

```
1  a = myrandi(30)
2  b = myrandi(40)
3  ab = vedicmultiply(a,b)
4
5  a =
6
7      '382642825609132600080066017327'
8
9
10 b =
11
12     '7227156803371186372380298001959308865779'
13
14
15 ab =
16
17     '2765419700362217091952928383747807930261048503348310798996406131352733'
```

Figure 5: MATLAB code: Random 30-digit number a × random 40-digit number b, including output.

2.5 Task 5

In the last part of the report, we will create a well-labeled graph that illustrates the relationship between N (number of digits in the integer), on the horizontal x-axis, and T (average time required to perform 100 multiplications) using two randomly generated positive integers, each having N digits, on the vertical y-axis. Ideally, choose values of N from 10 to 50 in increments of 10.

In **figure** 6 (**Line2-3**), `N_values` is an array containing values from 10-50 in increments of 10, and `T_values` is initialised as an array of zeros with the same size as `N_values`. (**Line5-6**) set `num_runs` to 100 for 100 runs of a specific operation timed, and `times` is preallocated as a 2D array with `num_runs` rows and the same numberof columss as the length of `N_values`, it was used to sore the execution times of operations for different values of `N` [4]. The loop in (**Line8**) is used to check if a parallel pool is exists, if not then create one[5]. (**Line12-13**) generate two random integers `int1` and `int2` in array format with the help of `myrandi` function from section 2.1.

(**Line15-23**), we create a parallel loop `parfor` [6] that iterates over the values inside `N_values`, and each value of N follows the following steps:

- 1. Inner loop (for `num_runs` times) to execute for the current `N` value with `vedicmultiply` function from section 2.3.
- 2. `tic` starts the timer, and `toc` stops the timer, measuring the time taken for each run, and the length of time is stored in `times` array.

After the runs are completed for each value of `N`, the code in (**Line24**) calculates the mean execution time and stores in the `T_values` array.

From the graph in **figure** 7, it is easy to tell the data values show an exponential pattern between the number of digits and the time taken for multiplication. The increase in average time taken is larger for every 10 digits more. The data shows the average time raised from 20-digit to 30-digit is 0.3s, from 30-digit to 40-digit is around 0.4s and from 40-digit to 50-digit is around 0.6s. In summary, as the number of digits increases, the time required for computation also increases.

The full demonstration of the code, including output, can be found in the **figure** 6 and **figure** 7.

```
% Initialize arrays to store values of N and average time T
N_values = 10:10:50;
T_values = zeros(size(N_values));
% Preallocate the times array
num_runs = 100;
times = zeros(num_runs, length(N_values));
% Create a parallel pool (if it doesn't exist)
if isempty(gcp('nocreate'))
    parpool;
end
% Generate random integers for all N values in one go
int1 = arrayfun(@(N) myrandi(N), N_values, 'UniformOutput', false);
int2 = arrayfun(@(N) myrandi(N), N_values, 'UniformOutput', false);
% Iterate over values of N
parfor k = 1:length(N_values)
    N = N_values(k);
    for i = 1:num_runs
        % Time the multiplication operation using vedicmultiply
        tic;
        vedicmultiply(int1{k}, int2{k});
        times(i, k) = toc;
    end
end
T_values = mean(times); % Calculate and store the average time
% Plot the results
figure;
plot(N_values, T_values, '-o');
xlabel('N (Number of digits)');
ylabel('T (Average time for 100 multiplications [s])');
title('Vedic Multiplication Performance');
grid on;
```

Figure 6: MATLAB code: 100 multiplications using two positive random integers.

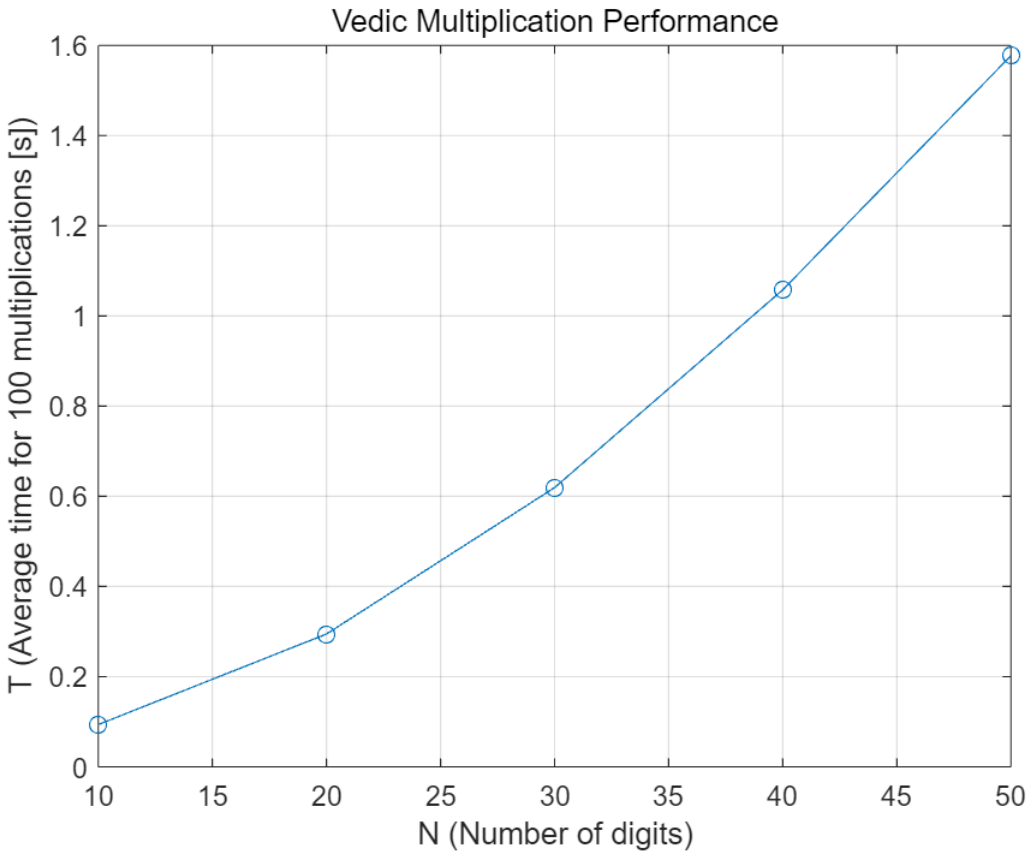

Figure 7: Output of MATLAB code: 100 multiplications using two positive random integers.

References

[1] Upavidhi Vedic Maths. Vedic Mathematics: Sūtra 3. ūrdhva tiryagbhyāṃ. [online]
Available: `https://www.upavidhi.com/sutra/urdhva-tiryagbhyam`

[2] Gupta, M. (2012). Vedic Math - Multiplication of any numbers. [online]
Available: `https://www.vedantatree.com/2012/08/vedic-math-multiplication-of-any-numbers.html`

[3] uk.mathworks.com. Format data into string or character vector - MATLAB sprintf - MathWorks United Kingdom. [online]
Available: `https://uk.mathworks.com/help/matlab/ref/sprintf.html`

[4] Mathworks.com. (2014). Create Line Plot. [online]
Available: `https://uk.mathworks.com/help/matlab/ref/plot.html`

[5] uk.mathworks.com. Get current parallel pool - MATLAB gcp - MathWorks United Kingdom. [online]
Available: `https://uk.mathworks.com/help/parallel-computing/gcp.html`

[6] uk.mathworks.com. Execute for-loop iterations in parallel on workers - MATLAB parfor - MathWorks United Kingdom. [online]
Available: `https://uk.mathworks.com/help/parallel-computing/parfor.html`