# Project1_POW

《Fundamental Data Structure》

Project1 Report

| | | |
|---|---|---|
| Teacher | : | 杨子祺 |
| Date | : | 2023/10/8 |

# Chapter 1: Introduction

There exist a minimum of two distinct algorithm to calculate $X^N$ where $N$ is a positive integer.

The first method multiplies $X$ by itself $N-1$ times.
The second method operates as such: when $N$ is even, $X^N = X^{N/2} \times X^{N/2}$. If $N$ is odd, $X^N = X^{(N-1)/2} \times X^{(N-1)/2} \times X$.

The tasks assigned to me are:

- (1) Code the first method and an **iterative** approach to the second method;
- (2) Delve into the complexities of both methods;
- (3) Evaluate and contrast the efficiencies of the first method and both the iterative and recursive variants of the second method when (X = 1.0001) and (N) ranges from 1000 to 100000 at specified intervals.

# Chapter 2: Algorithm Specification

- Algorithm 1 (**multiplication**) is to use N−1 multiplications.
- Algorithm 2 (**iteration**) works in the following way: if $N$ is even, $X^N = X^{N/2} \times X^{N/2}$; and if N is odd, $XN = X^{(N-1)/2} \times X^{(N-1)/2} \times X$.
- Algorithm 2 (**recursive**) works in the following way: if $N$ is even, $X^N = X^{N/2} \times X^{N/2}$; and if N is odd, $XN = X^{(N-1)/2} \times X^{(N-1)/2} \times X$.

## Algorithm 1 (multiplication):

- **Purpose**: This method calculates $X^N$ by simply multiplying $X$ with itself $N$ times.
- **Implementation**: It initializes a result variable with 1. A loop runs $N$ times, and in each iteration, the value of `x` is multiplied with the result.

```
double multiplication(double x, int n) { // operate (n-1) times of multiplications
    double res = 1;
    for (int i = 0; i < n; i++) {
        res *= x;
    }
    return res;
}
```

## Algorithm 2 (iteration):

- **Purpose**: This method uses the properties of exponentiation to compute $X^N$ in a more efficient manner than the straightforward multiplication. It uses the principle of "exponentiation by squaring" in an iterative manner.
- **Implementation**: A result variable is initialized to 1. A while loop continues as long as $N$ is positive. Inside the loop:
    - If $N$ is even, $X$ is squared (multiplied by itself), and $N$ is halved.

- If $N$ is odd, the result is multiplied with $X$, then $X$ is squared, and $N$ is decreased by 1 before being halved.

```
double iterative(double x, int n) {
    double result = 1; // initialize result

    while (n > 0) {
        if (n % 2 == 0) {  // n is even
            x *= x;
            n /= 2; // X^N = X^(N/2) * X^(N/2) = (X^2)^(N/2)
        } else {  // n is odd
            result *= x;
            x *= x;
            n = (n - 1) / 2; // X^N = X^(N/2) * X^(N/2) = (X^2)^(N/2)
        }
    }

    return result;
}
```

## Algorithm 3 (recursive):

- **Purpose**: This method, like the iterative method, uses the "exponentiation by squaring" principle but does so using a recursive approach.
- **Implementation**:
  - The base case: If $N = 0$, it returns 1 because any number raised to the power of 0 is 1.
  - If $N$ is even, the function recursively calls itself with $X$ squared and $N$ halved.
  - If $N$ is odd, it recursively calls itself with $X$ squared and $(N-1)/2$, then multiplies the result with $X$.

```
double recursive(double x, int n) {
    if (n == 0) { // recursive base case
        return 1;
    } else if (n % 2 == 0) { // n is even
        return recursive(x * x, n / 2); // X^N = X^(N/2) * X^(N/2) = (X^2)^(N/2)
    } else { // n is odd
        return recursive(x * x, (n - 1) / 2) * x; // X^N = X^(N/2) * X^(N/2) =
(X^2)^(N/2)
    }
}
```

# Chapter 3: Testing Results

**ticks difference**

In my computer(MacOS) 1 ticks equals 1,000,000 per second, I read the source lib code and it says:

```
#if __DARWIN_UNIX03
#define CLOCKS_PER_SEC  ((clock_t)1000000)      /* [XSI] */
#else /* !__DARWIN_UNIX03 */
#include <machine/_limits.h>    /* Include file containing CLK_TCK. */
#define CLOCKS_PER_SEC  ((clock_t)(__DARWIN_CLK_TCK))
#endif /* __DARWIN_UNIX03 */

#define __DARWIN_CLK_TCK 100 /* ticks per second */
```

And in my former Windows, I read the source lib code and it(possibly use low version of gcc) says:

```
#define CLK_TCK CLOCK_PER_SEC
#define CLOCK_PER_SEC 1000
```

Thus, ticks differ a lot in different computers.

> ⚠ **TO RUN IT Please Use Command Line, and Unix System environment is preffered.**
>
> Place the 3 source code in the **same directory**, and input **make run** in your command line at the directory path to automatically generate executable files and run them.

## Shell Output

> I use MacOS's shell and WSL (ubuntu) to test it

```
$ make run
gcc -o project1_1000 project1_POW.c -O0 -DN=1000
gcc -o project1_5000 project1_POW.c -O0 -DN=5000
gcc -o project1_10000 project1_POW.c -O0 -DN=10000
gcc -o project1_20000 project1_POW.c -O0 -DN=20000
gcc -o project1_40000 project1_POW.c -O0 -DN=40000
gcc -o project1_60000 project1_POW.c -O0 -DN=60000
gcc -o project1_80000 project1_POW.c -O0 -DN=80000
gcc -o project1_100000 project1_POW.c -O0 -DN=100000
Running this time: project1_1000
result = 1.105165
iteration = 1000, ticks = 6566, total time = 0.006566(s), duration = 0.0000065660(s)
result = 1.105165
iteration = 1000, ticks = 43, total time = 0.000043(s), duration = 0.0000000430(s)
result = 1.105165
iteration = 1000, ticks = 60, total time = 0.000060(s), duration = 0.0000000600(s)
Running this time: project1_5000
result = 1.648680
iteration = 1000, ticks = 33719, total time = 0.033719(s), duration = 0.0000337190(s)
result = 1.648680
iteration = 1000, ticks = 49, total time = 0.000049(s), duration = 0.0000000490(s)
result = 1.648680
iteration = 1000, ticks = 71, total time = 0.000071(s), duration = 0.0000000710(s)
Running this time: project1_10000
```

```
result = 2.718146
iteration = 1000, ticks = 59676, total time = 0.059676(s), duration = 0.0000596760(s)
result = 2.718146
iteration = 1000, ticks = 52, total time = 0.000052(s), duration = 0.0000000520(s)
result = 2.718146
iteration = 1000, ticks = 81, total time = 0.000081(s), duration = 0.0000000810(s)
Running this time: project1_20000
result = 7.388317
iteration = 1000, ticks = 109197, total time = 0.109197(s), duration = 0.0001091970(s)
result = 7.388317
iteration = 1000, ticks = 57, total time = 0.000057(s), duration = 0.0000000570(s)
result = 7.388317
iteration = 1000, ticks = 87, total time = 0.000087(s), duration = 0.0000000870(s)
Running this time: project1_40000
result = 54.587232
iteration = 1000, ticks = 210980, total time = 0.210980(s), duration = 0.0002109800(s)
result = 54.587232
iteration = 1000, ticks = 61, total time = 0.000061(s), duration = 0.0000000610(s)
result = 54.587232
iteration = 1000, ticks = 95, total time = 0.000095(s), duration = 0.0000000950(s)
Running this time: project1_60000
result = 403.307791
iteration = 1000, ticks = 310821, total time = 0.310821(s), duration = 0.0003108210(s)
result = 403.307791
iteration = 1000, ticks = 63, total time = 0.000063(s), duration = 0.0000000630(s)
result = 403.307791
iteration = 1000, ticks = 96, total time = 0.000096(s), duration = 0.0000000960(s)
Running this time: project1_80000
result = 2979.765922
iteration = 1000, ticks = 405566, total time = 0.405566(s), duration = 0.0004055660(s)
result = 2979.765922
iteration = 1000, ticks = 65, total time = 0.000065(s), duration = 0.0000000650(s)
result = 2979.765922
iteration = 1000, ticks = 98, total time = 0.000980(s), duration = 0.0000000980(s)
Running this time: project1_100000
result = 22015.456049
iteration = 1000, ticks = 513670, total time = 0.513670(s), duration = 0.0005136700(s)
result = 22015.456048
iteration = 1000, ticks = 66, total time = 0.000066(s), duration = 0.0000000660(s)
result = 22015.456048
iteration = 1000, ticks = 100, total time = 0.000100(s), duration = 0.0000001000(s)
```

## Time Measurement Table

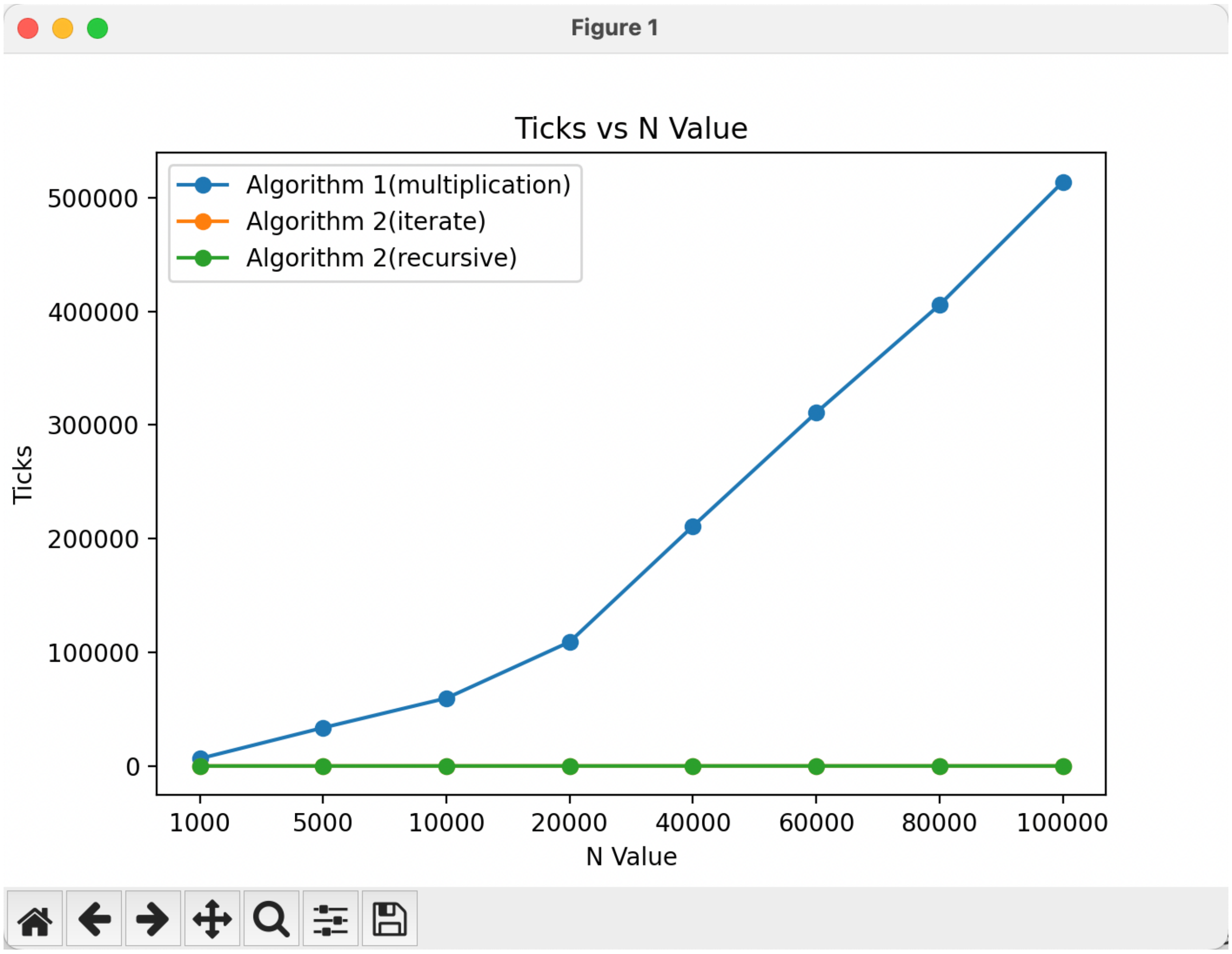
| **-** | **N** | **1000** | **5000** | **10000** |
|---|---|---|---|---|
| Algorithm 1 | Iteration(K) | 1000 | 1000 | 1000 |
| - | Ticks | 6566 | 33719 | 59676 |
| - | Total Time(sec) | 0.006566 | 0.033719 | 0.059676 |
| - | duration(sec) | 0.0000065660 | 0.0000337190 | 0.000059676 |
| Algorithm 2(iterate) | Iteration(K) | 1000 | 1000 | 1000 |
| - | Ticks | 43 | 49 | 52 |
| - | Total Time (sec) | 0.000043 | 0.000049 | 0.000052 |
| - | duration (sec) | 0.0000000430 | 0.0000000490 | 0.0000000520 |
| Algorithm 2(recursive) | Iteration (K) | 1000 | 1000 | 1000 |
| - | Ticks | 60 | 71 | 81 |
| - | Total Time (sec) | 0.000060 | 0.000071 | 0.000081 |
| - | duration (sec) | 0.0000000600 | 0.0000000710 | 0.0000000810 |

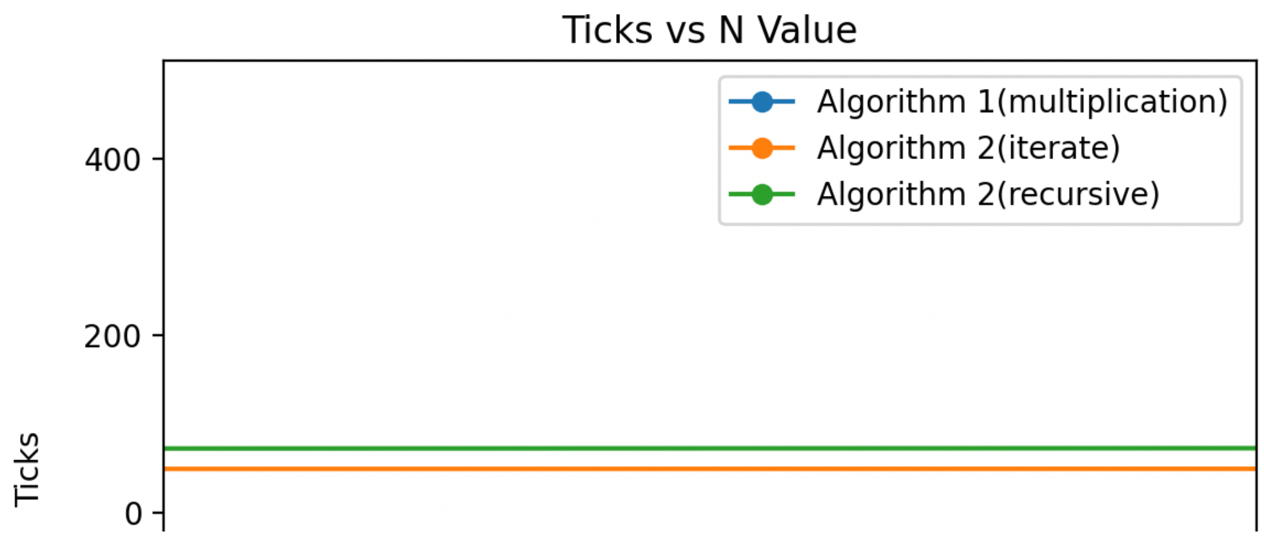
| 20000 | 40000 | 60000 | 80000 | 100000 |
|---|---|---|---|---|
| 1000 | 1000 | 1000 | 1000 | 1000 |
| 109197 | 210980 | 310821 | 405566 | 513670 |
| 0.109197 | 0.210980 | 0.3108210 | 0.405566 | 0.513670 |
| 0.0001091970 | 0.000210980 | 0.0003108210 | 0.000405566 | 0.000513670 |
| 1000 | 1000 | 1000 | 1000 | 1000 |
| 57 | 61 | 63 | 65 | 66 |
| 0.000057 | 0.000061 | 0.000063 | 0.000065 | 0.000066 |
| 0.000000057 | 0.000000061 | 0.000000063 | 0.000000065 | 0.000000066 |
| 1000 | 1000 | 1000 | 1000 | 1000 |
| 87 | 95 | 96 | 98 | 100 |
| 0.000087 | 0.000095 | 0.000096 | 0.000098 | 0.000100 |
| 0.000000057 | 0.000000095 | 0.000000096 | 0.000000098 | 0.000000100 |

## Figure Plotter in Python

**Tester: Plot the run times of the functions.**

In this figure, iteration(K) equals 1000, ticks equals 1,000,000 per second. Algorithm 2 (iterative) and Algorithm 3 (recursive) produce very similar results, so their line graphs overlap to a significant extent. Run the source code and zoom the figure, you can see separated lines of yellow and green.
在这张图片中，循环次数K = 1000，1 tick = 一百万分之一秒，Algorithm 2(iterative) 和 Algorithm(recursive) 两个算法得到的 ticks 非常接近，所以折线图中有所重合，用源 python 代码跑后放大可以看见分开的黄绿色线。





# Chapter 4: Analysis and Comments

## Algorithm 1 (multiplication):

- **Purpose**: This method calculates $X^N$ by simply multiplying $X$ with itself $N$ times.
- **Implementation**: It initializes a result variable with 1. A loop runs $N$ times, and in each iteration, the value of `x` is multiplied with the result.

This algorithm uses N-1 multiplications to calculate X raised to the power of N. Therefore, the **time complexity is O(N)**.
The algorithm only requires storing an additional variable to hold the multiplication result, so the **space complexity is O(1)**.

## Algorithm 2 (iteration):

- **Purpose**: This method uses the properties of exponentiation to compute $X^N$ in a more efficient manner than the straightforward multiplication. It uses the principle of "exponentiation by squaring" in an iterative manner.
- **Implementation**: A result variable is initialized to 1. A while loop continues as long as $N$ is positive. Inside the loop:
  - If $N$ is even, $X$ is squared (multiplied by itself), and $N$ is halved.
  - If $N$ is odd, the result is multiplied with $X$, then $X$ is squared, and $N$ is decreased by 1 before being halved.

In this algorithm, the value of N is divided by 2 at each step until it reaches 0. For each division by 2, a multiplication operation is performed. Hence, the number of iterations is determined by the number of times N can be divided by 2, which is equal to the logarithm of N to the base 2, denoted as log(N). Therefore, the **time complexity** of Algorithm 2 (iteration) is **O(log(N))**. The algorithm overwrite the variable evrytim, so the **space complexity is O(1)**.

## Algorithm 3 (recursive):

- **Purpose**: This method, like the iterative method, uses the "exponentiation by squaring" principle but does so using a recursive approach.
- **Implementation**:
  - The base case: If $N = 0$, it returns 1 because any number raised to the power of 0 is 1.
  - If $N$ is even, the function recursively calls itself with $X$ squared and $N$ halved.
  - If $N$ is odd, it recursively calls itself with $X$ squared and $(N-1)/2$, then multiplies the result with $X$.

In this algorithm, the value of N is divided by 2 at each step until it reaches 0. For each division by 2, a multiplication operation is performed. Hence, the number of iterations is determined by the number of times N can be divided by 2, which is equal to the logarithm of N to the base 2, denoted as log(N). Therefore, the **time complexity** of Algorithm 2 (iteration) is **O(log(N))**. Every recursive time algorithm requires storing an additional variable to hold the result, so the **space complexity is O(log(N))**.

# Chapter 5: Source Code in C, Makefile and Python with Comments

## C Souce Code (Decide K = 1000)

```
#include <stdio.h>
#include <math.h>
#include <time.h>
clock_t start, stop;
double duration;
```

```c
double total;
int ticks;

enum alg{
    multip, // 0
    iterat, // 1
    recurs, // 2
};

double multiplication(double x, int n);
double recursive(double x, int n);
double iterative(double x, int n);
void timemeasure(int alg, double x, int n, int k);

int main() {
    double x = 1.0001;
    int n = N; // If you use makefile I provided, you don't need to change this;
    // If you want to run the testcases yourself, change N to the exponent number
    int k = 1000;

    timemeasure(multip, x, n, k);
    timemeasure(iterat, x, n, k);
    timemeasure(recurs, x, n, k);

    return 0;
}

void timemeasure(int alg, double x, int n, int k) {
    double res = 0; //initialize result
    start = clock(); // start time ticking

    for (int i = 0; i < k; i++) { // operate k times function
        switch(alg) {
            case multip: res = multiplication(x, n); break;
            case iterat: res = iterative(x, n); break;
            case recurs: res = recursive(x, n); break;
        }
    }

    stop = clock(); // stop time ticking
    ticks = (int)(stop - start);
    total = (double)ticks/CLOCKS_PER_SEC;// replace "CLK_TCK" with "CLOCK_PER_SEC" in
case of OS difference
    // in Apple Computer's "time.h" and "limits.h": if not UNIX 03, #define
__DARWIN_CLK_TCK 100 /* ticks per second */
    // if is UNIX 03, #define CLOCKS_PER_SEC  ((clock_t)1000000)        /* [XSI] */
    // in low version of gcc: #define CLK_TCK CLOCKS_PER_SEC, #define CLOCKS_PER_SEC 1000
    duration = total / k;

    printf("result = %lf\n", res); // print result
    printf("iteration = %d, ticks = %d, total time = %lf(s), duration = %.10lf(s)\n", k,
ticks, total, duration);
}

double multiplication(double x, int n) { // operate (n-1) times of multiplications
    double res = 1;
    for (int i = 0; i < n; i++) {
        res *= x;
    }
    return res;
}

double iterative(double x, int n) {
    double result = 1; // initialize result

    while (n > 0) {
        if (n % 2 == 0) {  // n is even
            x *= x;
            n /= 2; // X^N = X^(N/2) * X^(N/2) = (X^2)^(N/2)
        } else {  // n is odd
            result *= x;
            x *= x;
            n = (n - 1) / 2; // X^N = X^(N/2) * X^(N/2) = (X^2)^(N/2)
        }
    }

    return result;
}

double recursive(double x, int n) {
    if (n == 0) { // recursive base case
        return 1;
    } else if (n % 2 == 0) { // n is even
        return recursive(x * x, n / 2); // X^N = X^(N/2) * X^(N/2) = (X^2)^(N/2)
    } else { // n is odd
        return recursive(x * x, (n - 1) / 2) * x; // X^N = X^(N/2) * X^(N/2) =
(X^2)^(N/2)
    }

}
```

I decide K = 1000 here, if you want to manually execute it, change your N in the source C code, and use lines like these in command line to run it (or you can just use GUI):

```
$ gcc -o project1_1000 project1_1000.c -O0; ./project1_1000
# e.g. N = 1000
```

## Makefile Source Code

**Compile and Run Test Cases**

Use `make run` to automate the compilation and execution of the test code and `make clean` to delete former target files. It is recommended to use the WSL (Windows Subsystem for Linux) or macOS terminal with GCC. If you are not familiar with the command line, please modify the value of `n` in the C source code manually and compile it for execution.

在终端中使用 `make run` 来自动化编译 c 源码，`make clean` 来清除之前生成的目标文件。推荐使用 wsl 或者 macOS 的终端和 gcc 来编译，如果你不熟悉命令行或 makefile，可以修改 c 源码中 `n` 的值手动编译执行，得到测试结果。

```
# List of values for the variable N
# N 变量的取值列表
N := 1000 5000 10000 20000 40000 60000 80000 100000

# Generate the target names using the values in N
# 使用 N 中的值生成目标文件名
TARGETS := $(foreach n,$(N),project1_$(n))

.PHONY: all clean run

# Default target: build all targets
# 默认目标: 构建所有目标
all: $(TARGETS)

# Pattern rule for building project1_N target
# 用于构建 project1_N 目标的模式规则
project1_%: project1_POW.c
	gcc -o $@ $< -O0 -DN=$*

# Run all the targets
# 运行所有目标
run: $(TARGETS)
	@for target in $(TARGETS); do \
		echo "Running this time: project1_$${target#*_}" && ./$$target ; \
	done

# Clean all the target files
# 清理所有目标文件
clean:
	rm -f $(TARGETS)
```

To run this, you simply need:

```
$ make run # automatically compile and execute the code
$ make clean # delete all the target file
```

## Time Plotter in Python

```python
import matplotlib.pyplot as plt

### data = { # data
    '1000': [6566, 43, 60],
    '5000': [33719, 49, 71],
    '10000': [59676, 52, 81],
    '20000': [109197, 57, 87],
    '40000': [210980, 61, 95],
    '60000': [310821, 63, 96],
    '80000': [405566, 65, 98],
    '100000': [513670, 66, 100]
}

x_values = []
y_values = [[] for i in range(3)]
labels = ['Algorithm 1 (multiplication)', 'Algorithm 2 (iterative)', 'Algorithm 3
(recursive)']

for key, value in data.items(): # Extract x and y values from the data dictionary
    x_values.append(key)
    for i, val in enumerate(value):
        y_values[i].append(val)

for i in range(3): # Plot the data
    plt.plot(x_values, y_values[i], marker='o', label=labels[i])

plt.xlabel('N Value') # Set labels and title and display legend
plt.ylabel('Ticks')
plt.title('Ticks vs N Value')
plt.legend()

plt.show() # Show the plot
```

In command line use this line to run the code:

```
$ python3 timemeasure.py
```

## Declaration

I hereby declare that all the work done in this project titled "Project1" is of my independent effort.