THEORETICAL METHOD FOR TESTING AN UNTESTABLE CASES

By

Chang Wing Le

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology

(Kampar Campus)

May 2019

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: _____

_____

_____

Academic Session: _____

I _____

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

_____ _____

(Author's signature) (Supervisor's signature)

Address:

_____

_____ _____

_____ Supervisor's name

Date: _____ **Date**: _____

THEORETICAL METHOD FOR TESTING AN UNTESTABLE CASES

By

Chang Wing Le

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology

(Kampar Campus)

May 2019

DECLARATION OF ORIGINALITY

I declare that this report entitled "**THEORETICAL METHOD FOR TESTING AN UNTESTABLE CASES**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : _____

Name : _____

Date : _____

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Dr Tse Siu Hong Savio. Dr Savio had been given a lot of helping for guiding me on how to progress for this project. His patience helped me to have my time to stepped out from comfort zone. I also need to thank my parents for giving unconditional support for the choice of my education.

ABSTRACT

In the area of software testing, there are many untestable cases due to the lack of oracles. Examples are finding a hotel with minimum price, using a spreadsheet to find sum or average of a large set of numbers, simulation, etc.. In the literature, it is called the Oracle Problem. Oracle Problem cannot be solved completely due to the natures of the requirements of the functions. What we can do is to alleviate the problem by showing more evidence on the correctness of the functions.

We notice that many numerical functions are not testable, like $\sin(29.8^o)$, finding the natural number *e*, finding π, etc., when the correctness is measured to a large number of decimal point digits. For those functions with integer outputs, the scenarios are not much better. There is an example—finding n!, which is not testable when n is large, like 100000. In this project, we develop four methods which show evidences on the correctness of n!

The first method is to apply the Wilson theorem in the literature, which states the property of n! when n+1 is a prime. We use this property for testing n! when the input is prime.

The second method is length-testing. Without computing n!, we only find the length of n!, because it is much easier than n! itself. If a function being tested gives the correct n!, the length of this output n! must perfectly match our finding on the length of n!.

The third method is prime-counting. Without computing n!, we want to match the numbers of factors of "3" (say), "5", "7", etc., inside n!, with our theoretical results.

The fourth method is metamorphic-testing, which is a new technique in software testing proposed within these 10 years. We show that it is possible to apply this on finding n!, and we will also prove its very low probability of errors.

By the above methods, we alleviate the Oracle Problem for testing n!. We consider this project as a milestone in exploring some more general techniques for testing other numerical functions.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Background

Software testing is an important branch in Software Engineering. Since testing a program is basically incomputable, in computational theory; however, it is the gateway to make sure that the systems to be delivered are reasonably error-free. In order to make software testing efficient, oracles are needed. Oracle is referred to the inputs which can be used for verifying (but not proving) the correctness systems efficiently.

Untestable cases in software testing are those programs having no oracles so far. In the literature, it is also called Oracle Problem. In short, oracles are in the format of (input, output); and Untestable cases occur when the output cannot be found efficiently. Often for testing these programs could lead to countless number of test-cases and consuming a lot of time thus untestable cases had been one of the basic challenges for software tester. Difficulty for testing these programs often caused by its error-prone behavior or output results that are too difficult to be verified by human manually. It is also known as “test oracle problem”. Test oracle is collection of information to determine if a test have passed or failed, normally specifying a set of inputs and desired output. A survey had been done for analyzing properties of test oracle and test oracle problems (Barr et al., 2015). In this survey (Barr et al., 2015), according to distinctively properties, these test oracles had been divided into different categories, calling “test oracles can be specified”, “test oracles can be derived”, “test oracles can be built from implicit information” and “no automatable oracle but can reduce human effort”.

In effort to solve these pseudo-oracle problems, many specific software techniques such as metamorphic testing had been introduced by T.Y.Chen (Zhi Quan Zhou et al, 2004). Metamorphic testing originally being invented with the intention to complete testing by reducing number o test-cases compare to general software testing technique. Metamorphic testing is performed but defining metamorphic relation between the program input and output to act as follow-up cases for multiple execution of desired program. According to a study, it suggests metamorphic testing could potentially helping on solving oracle problem to some degree even testing is performed but inexperienced testers (Liu, H., Kuo, F., Towey, D. and Chen, T, 2014). However, for each specific case, even applied with certain testing technique, it needed to be defined their own specific test cases which are highly dependent on tester skills. For untestable cases, the ideal way for testing a

program would be replicating the original program using different programming language and libraries bu different develop team, which called pseudo-oracle (Weyuker, 1982). However, it is too costly on using pseudo-oracle, if there is mismatch output between original and replicate program, it is difficult to determine which one is the correct one and needed for second replication (Weyuker, 1952). Hence, it is better to find some other methods on testing an untestable case.

As an example for untestable cases, the sin function, let's say for testing the output of sin(78), we can use test case sin(78)= sin(45+33) = sin(45)cos(33) + sin(45)cos(33), where sin(45)=$\frac{1}{\sqrt{2}}$, cos(45) = $\frac{1}{\sqrt{2}}$. But in this case, the value of sin(33) and cos(33) could be difficult to tested out the result even for further extending (sin(33))=sin(30+3) and cos(33) = cos(30+3.)

Another good example of untestable cases would be the pi number, π that a program is done for showring all digits for pi 3.14159265… that could be infinite long, normally the result it self automatically admitted as true result for further used in pi analysis. However, to confirm the pi program truly perform desired behavior, the whole output needed to be retest again, thus showing the importance of testing the cases what is considered as "untestable".

Factorial function is another untestable function, and we focus on this in this project. There are many applications for factorial function such as combinatorics. Moreover, since factorial function could only give discrete values, efforts had been done one extending factorial function for wider appliances, one of them is the notable Gamma function. General factorial function n! can only receive positive integer arguments. Gamma function extends the range of argument to fractions and negative arguments. Unlike factorial function, Gamma function could give a continuous graph for further analysis, hence it could be said n! is a special case for Gamma function. General definition for Gamma function is $\Gamma(z+1) = \int_0^\infty x^{z-1}e^{-x}dx = (z)!$. There is also several derivation of Gamma formula done in use of n! such as $\Gamma(1+n) = \frac{(2n)!}{4^n n!}\sqrt{\pi}$,, hence we can insert n! to test Gamma function. This means that n! is needed to be determined first. Another usage of factorial function is exponential function e^x, which is equal to $1 + (x/1!) + (x^2/2!) + (x^3/3!) + \ldots$ We believe that a correct value of n!, for large n, is useful in scientific computing. For example, n particles will have n! permutations and under the model of uniform probabilities, each permutation is 1/n!. A wrong value of n! will give a wrong probability, which may affect a lot especially in the mathematics of quantum mechanics.

It is common for social media on interpreting how to solve a large factorial number is simplifying the factorial number based on division by a very large factorial number with relatively smaller factorial number. For example, (51!-50!)/(50!-49!) = (50! x (51-1))/(49!x(50-1)) = 2550/49. However, this kind of solution is inadequate because the problem isn't really related for finding actual large factorial number and instead applying art of problem solving onto simpler visible number due to the nature of division. It could be a very difficult task for verifying correctness of actual factorial number n! when n is larger. This project would bring contribution to advanced scientific calculator for finding actual factorial number but not the approximation.

1.2 Problem Statements

Definition for factorial function is n! = 1* 2 * 3* … * (n-1) * n, value for n could only receive positive integer argument. One of the properties for factorial functions is that its value rises greatly on increasing the value of n, which makes it be difficult to be verified correctness of factorial number program output manually or automatically. When n is large, n! is considered as an untestable function. It is not surprising that one may not agree to test n! due to the simplicity of its iterative and recursive algorithms as shown below:

```
factorial = 1;
For i = 1 to n
    factorial = factorial * i ;
return factorial;
```

```
Algorithm factorial(n)
if n = 1
   return 1;
return n*factorial(n-1);
```

However, the above algorithms are correct and perfect only when n is small. For large n, the integer or long integer type in any language will encounter overflow. Programmers must construct an extremely long integer type by himself. When the compiler platform or even the computer memory cannot withstand such long integer, programmers need to use secondary storage for storing any single integer. The multiplication of integers of such type is certainly error prone, and therefore, systems which output n! are needed to be tested.

Why n! is not testable? Even when n is as small as 20, from our experience, we do not have any way to verify 20! = 3,715,891,200, except re-computing it, or performing division with divisors 1, 2, 3, …, 20. Both methods are not preferred in software testing, because they are not oracles, but simply another program for n! which should be tested too. In this sense, n! is not testable.

In this project, exploration on methods for testing n! is the main objective. The immediate method on testing factorial program is using the famous Stirling formula. Stirling formula is approximation to factorial formula which could give results closely to actual factorial number. The formula is stated below:

$$n! = \sqrt{2\pi}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right)$$

As shown at the above, error rate for Stirling Approximation to actual factorial numbers n! would be 1/n, so the higher the value of n, the lower the error rate. By taking an example, let's say when n = 200, the error rate for Stirling Approximation would be just 1/200 = 0.005. While one may be satisfied with this low error rate, error rate 1/n means that only the first Θ(log n) bits can be ensured its correctness.

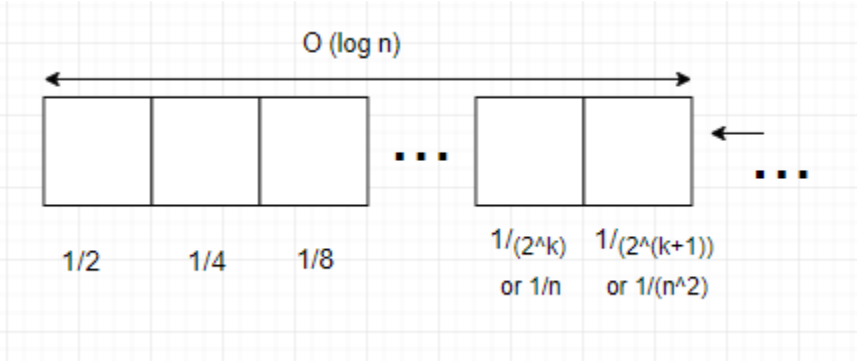


Figure 1.2(a) Bits calculation for Stirling Formula

Is Θ(log n) bit accuracy acceptable? It depends on the whole length of n!. We know that the length of a function f(n) is $(\log 2\lfloor f(n)\rfloor)+1$. For n!, it would be $(\log_2\lfloor n!\rfloor)+1 = (\log_2 n!)+1$.

For upper bound,

$$\log(n!) + 1 = \log(1) + \log(2) + \log(3) + \cdots + \log(n) + 1$$
$$\leq \log(n) + \log(n) + \log(n) + \cdots + \log(n) + 1 = nlog(n) + 1$$

For lower bound,

$$\log(n!) + 1 = \log(1) + \log(2) + \log(3) + \cdots + \log(n) + 1$$
$$\geq \log\left(\frac{n}{2}\right) + \log\left(\frac{n}{2}+1\right) + \log\left(\frac{n}{2}+2\right) + \ldots + \log(n) + 1 = \frac{n}{2}log\left(\frac{n}{2}\right) + 1$$

Hence, the length of n! is O(n log n), and therefore, the correctness of O(log n) leading digits contribute little to the correctness of the whole n!.

Could we count the trailing zeroes? We only need to consider all factors 2*5 to get the result from the factorial numbers, each 2*5 factor would mean there is one trail zero for the number. Since in factorial number higher value of prime factor always have lower occurrence than lower value of prime factor, hence only calculated for factors of 5 equals to the number of trailing zeroes.

Calculation is as follow:

Let's say

$$5^{k-1} < n \leq 5^k$$

Then

$$5n > 5^k > n$$

$$n! \leq 5^k!$$

From section 2.3 later, it would be reviewed that $p^k!$ has $((p^k - 1) / (p-1))$ factors of p, where p is a prime.

Hence,

$$Factor\ of\ 5 \leq \frac{5^k - 1}{5 - 1} = \Theta(5^k) = \Theta(n)$$

Since the trailing zeros used up would be Θ(n) of spaces which is still insignificant compare to Θ(n log n). It can be concluded that Θ(log n) bits guaranteed by Stirling formula and Θ(n) trailing zeros together are still not significant enough to be used as test case for actual factorial program output. For further increasing testable digits, trailing zeros in factorial number is found. There would be always some digits in the middle of factorial number that are untestable, the ratio of these untestable digits to factorial number needed to be find out too in order to know efficiency of this testing method.

1.3 Contribution (Why choose factorial number as test case)

For scientific calculator, especially those used by researcher who are going on a real-life large project which highly dependent on real environment needed to get true result as accurate as possible when doing calculation. This is because on large project, such as launching a rocket toward the space, a slightly deviation from the true calculation result could lead to the rocket launch towards a very different direction. Factorial number as a common quantity number, had its many application on combinatorics and approximation of other function such as exponential and logarithm. In this project, it assumed factorial number would be an important property for incoming future scientific works and needed to be computed as accurate as possible.

As illustrate at section 1.2, factorial number result is difficult to test and consume high computation power. In this project, several methods are explored and give its importance on testing factorial number at different condition. Methods such as testing using Wilson theorem and length testing could compute fast and can be used as earlier testing for factorial number. Although methods mentioned above suggested to be used as an earlier testing, these methods are considered as high efficiency for finding faults on the program too. Other than these two, this project also included methods so called prime-counting and metamorphic-testing, these computes result relatively slower than methods mentioned earlier, but could show more irregular pattern on the program faults.

This project may contribute effort for building advanced scientific calculator which want to get actual factorial number. Algorithm for factorial number had been studied carefully at the past, with both approximation or getting the actual factorial number. Nevertheless, output of n! program could be a difficult task for testing its correctness, hence contents in this project could be used to relieving the problem thus reducing the error might got when developers try to build a factorial number program.

CHAPTER 2: FOUR METHODS

2.1 Applying Wilson's theorem

Wilson's theorem (Andrew Ohana, 2009) stated that integer p > 1, then (p-1)! + 1 is divisible by p if and only if p is prime.

$$(n-1)! \equiv -1 (\text{mod } n)$$

By giving the condition if and only if, if n is a prime, Wilson's theorem can be used for testing the factorial function. It is not difficult to determine whether an integer is a prime, because we can apply Agrawal et al's polynomial time algorithm (Manindra Agrawal et al, 2004)。

In the literature, this problem is called primality testing. Though we have the primality testing done in P-time, it is still difficult to find a prime within a range of integers because of its scarcity. This is the short-coming for applying Wilson's Theorem.

2.2 Length-testing

We count the number of base-10 digits in n!. Number of digits could be a good testing method as early testing for factorial number. The number of digits for a number x is always as $1 + \lfloor logx \rfloor$. In the case of factorial number, the number of digits is:

$$1 + \lfloor \log_{10} n! \rfloor = 1 + \lfloor \log_{10}(1 * 2 * 3 * 4 * \dots * n - 1 * n \rfloor$$

$$= 1 + \lfloor \log_{10} 1 + \log_{10} 2 + \log_{10} 3 + \log_{10} 4 + \dots + \log_{10}(n-1) + \log_{10} n \rfloor$$

The formula above list an appropriate property of factorial number since the time complexity for addition above is only O(n log log n) and if computed correctly it could be 100% effectiveness on confirming fault on the program when numbers of digits for the output is not matching with formula above. Before on confirming the 100% effectiveness, there would be some aspect need to be concern with the method, in order to achieve effectiveness mentioned above.

In this project, we do not assume the reliability of log function in any built-in library, due to the unopened source of truncation or round off techniques. Our method requires the software tester to write his own function of logarithm and at which point the truncation takes place depends on an input parameter. No matter how well the precision is, the effect of many truncations may affect the integer part of the final sum.

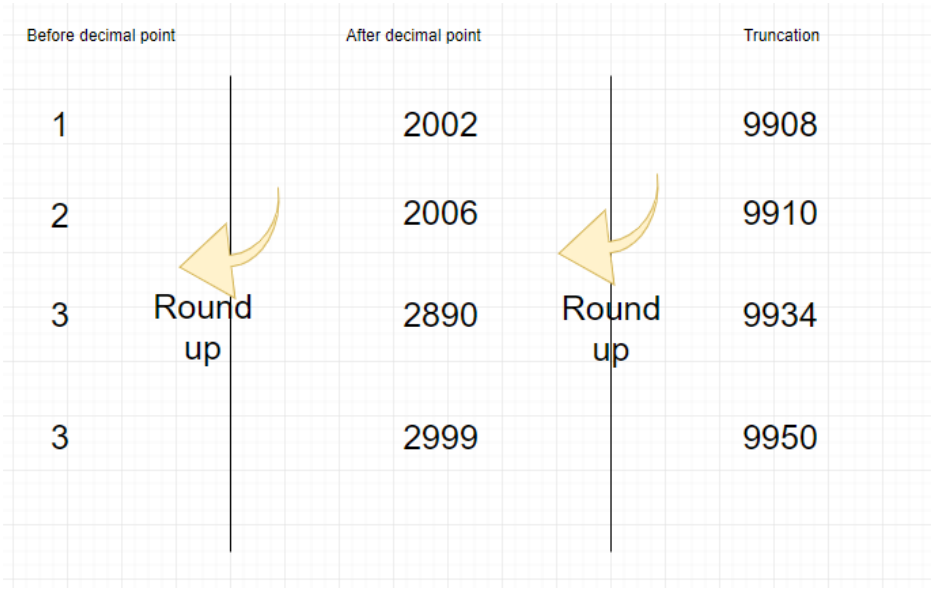


Figure 2.2.1 Concept of missing one value before decimal point due to truncation

Truncation occurs at each $\log_{10} i$, i=1, 2, …, n. Suppose all digits after the pth decimal point will be truncated. The error for each truncation will be at most 10^{-p}, and totally, the total error will be at most $n*10^{-p}$. This will be at most 1 unit at the $(p - \lfloor log\ n \rfloor)$th decimal point. However, this value may cause one number missing from the $1 + \lfloor log\ n! \rfloor$ and becoming $\lfloor \log n! \rfloor$ which is not accurate for the properties of actual factorial number. When this occurs, the format of log n! must have "9" for each digit from the 1st to the (p-⌊log n⌋-1)th decimal point. However the missing-one effect will not occur when we try m! where m=n+1 and m is not a power of 10. In case n+1 is a power of 10, then we set m=n+2. We explain the reason below:

Lets say $10^K + 1 \le m < 10^{K+1}$, for m!

Claim: For $m = 10^K + 1$, $\log(10^K + 1) = K + 0.000 \ldots 000x \ldots$, where x is any non-zero digit, and the number of "0"s in the decimal points before x is no more than $\lfloor logn \rfloor$.

Proof:

$$\log(10^K + 1) = \log(10^K)\left(1 + \frac{1}{10^K}\right) = \log 10^K + \log(1 + \frac{1}{10^K})$$

$$= K + \frac{1}{ln10} \ln\left(1 + \frac{1}{10^K}\right) \le K + \frac{1}{ln10}(\frac{1}{10^K})$$

Q.E.D.

From $K + \frac{1}{ln10}(\frac{1}{10^K})$, it become $\ln(1 + y) \le y$ for small y, to illustrate it graphically it would be as below:

100000000	8
100000001	8.000000004
100000002	8.000000009
100000003	8.000000013
100000004	8.000000017
100000005	8.000000022
100000006	8.000000026
100000007	8.00000003
100000008	8.000000035
100000009	8.000000039
100000010	8.000000043

100000011	8.000000048
100000012	8.000000052
100000013	8.000000056
100000014	8.000000061
100000015	8.000000065
100000016	8.000000069
100000017	8.000000074
100000018	8.000000078
100000019	8.000000083
100000020	8.000000087
100000021	8.000000091
100000022	8.000000096

Figure 2.2.1 Value of $\log_{10}10^K$+t Figure 2.2.1 Value of $\log_{10}10^K$+t

Let's say K=8, then there must be always 8 zeroes between the decimal point and non-zero value after it. As the increasing value of $n \geq 10^K$, it also could be seen that the zeroes in between becoming lesser. However, with one column having K zeroes in between, it is already sufficient for doing truncation after K space without affecting the value before the decimal point.

This method is used as below:

For any given n,

if n+1 is not 10^k for any k,

let m=n+1

else // for some k, $n + 1 = 10^k$

let m=n+2;

Also let

A = $1 + \lfloor log\ n! \rfloor$

A'= $1 + \lfloor log\ m! \rfloor$

B = length of result from factorial program using input n

B'= length of result from factorial program using input m

Then compare the value of A and B, A' and B' which give conclusion based on the below condition:

If fulfilling any condition 1 combine with condition 2, then report no error. Else, report error.

Condition 1	Condition 2	Output
B=A	B'=A'	Report no error
B=A-1	B'=A'-1	

2.3 Prime counting

2.3.1 The number of times that a prime p appears inside n! where p ≤ n

Fact 2.3.1 For any positive integers, *a* and *b*, and a prime *b*<*p*, we have *ap*+*b* indivisible by *p*. □

Using Fact 2.3.1, we can prove Lemma 2.3.2, below.

Lemma 2.3.2 For any prime p, positive integers s, k, i, j, such that $s \leq k$, $i \leq p-2$, when $j \in [1, p^k]$, p^s can divide $(ip^k + j)$ if and only if p^s can divide j; and when $j \in [1, p^k - 1]$, p^s can divide $((p-1)p^k + j)$ if and only if p^s can divide j. □

Lemma 2.3.3 For any prime p, the number of factors, which are equal to p, in $1 * 2 * ... * p^k$ is $(p^k - 1)/(p-1)$.

Proof: We prove this by mathematical induction. When k=1, the number of factors, which are equal to p, inside $1 * 2 * ... * p^k$ is trivially one, and therefore, the lemma is true for k=1.

Inductive case: Suppose that the number of factors, which are equal to p, in $1 * 2 * ... * p^{k-1}$ is $(p^{k-1} - 1)/(p-1)$. We now prove that the number of factors, which are equal to p, in $1 * 2 * ... * p^k$ is $(p^k - 1)/(p-1)$.

Since $1 * 2 * ... * p^k = 1 * 2 * ... * p^{k-1} * (p^{k-1} + 1) * (p^{k-1} + 2) * ... * p^k$, it suffices to show that the number of factors, which are equal to p, in $(p^{k-1} + 1) * (p^{k-1} + 2) * ... * p^k$ is p^{k-1}.

Rewriting $(p^{k-1} + 1) * (p^{k-1} + 2) * ... * p^k$ into the product of $p-1$ expressions as follows:

$.[(p^{k-1} + 1) * (p^{k-1} + 2) * ... * (p^{k-1} + p^{k-1})] *$

$[(2p^{k-1} + 1) * (2p^{k-1} + 2) * ... * (2p^{k-1} + p^{k-1})] *$

$[(3p^{k-1}+1)*(3p^{k-1}+2)*...*(3p^{k-1}+p^{k-1})]*$

……

$[((p-2)p^{k-1}+1)*((p-2)p^{k-1}+2)*...*((p-2)p^{k-1}+p^{k-1})]*$

$[((p-1)p^{k-1}+1)*((p-1)p^{k-1}+2)*...*((p-1)p^{k-1}+p^{k-1})]$.

We can further rewrite the first $p-2$ expressions as

$$[(ip^{k-1}+1)*(ip^{k-1}+2)*...*(ip^{k-1}+p^{k-1})],$$

where $i \in [1, p-2]$. By Lemma 2.3.2, for any positive integer *s*, p^s can divide $(ip^{k-1}+j)$, where $j \in [1, p^{k-1}]$, if and only if p^s can divide *j*. Hence, the number of factors, which are equal to *p*, in the ith expression is the same as that in $1*2*...*p^{k-1}$, and it equals to $(p^{k-1}-1)/(p-1)$ by the inductive hypothesis.

Consider $\left[\left((p-1)p^{k-1}+1\right)*((p-1)p^{k-1}+2)*...*((p-1)p^{k-1}+p^{k-1})\right]$.

By Lemma 2.3.2, for some integer *s*, p^s can divide $\left((p-1)p^{k-1}+j\right)$, where $j \in [1, p^{k-1}-1]$, if and only if p^s can divide *j*. Note that, the factor $\left((p-1)p^{k-1}+p^{k-1}\right)$ is exactly p^k. Hence, the number of factors, which are equal to *p*, inside $\left[\left((p-1)p^{k-1}+1\right)*((p-1)p^{k-1}+2)*...*((p-1)p^{k-1}+p^{k-1})\right]$ is the same as that inside $1*2*...*p^{k-1}$ plus one. That is, $\frac{(p^{k-1}-1)}{p-1}+1$, by the inductive hypothesis.

Summing up, the number of factors, which are equal to *p*, in $(p^{k-1}+1)*(p^{k-1}+2)*...*p^k$ is $\left[\frac{(p^{k-1}-1)}{p-1}\right](p-2)+\frac{(p^{k-1}-1)}{p-1}+1=p^{k-1}$. □

Lemma 2.3.4 For any prime *p*, and positive integer $a<p$, the number of factors, which are equal to *p*, in $1*2*...*a\cdot p^k$ is $a(p^k-1)/(p-1)$.

Proof: By using Lemmas 2.3.2 and 2.3.3. □

Now we prove for the case that n may or may not be a power of prime p. We first express n as $a_k p^k + a_{k-1} p^{k-1} + \cdots + a_1 p + a_0$, where $a_i < p$, $i \in [0, k]$, for some positive integer k.

We partition $n!$ into $k + 1$ expressions, and then sum up the number of appearance of p inside them.

$[1 * 2 * \ldots * a_k p^k] *$

$[(a_k p^k + 1) * (a_k p^k + 2) * \ldots * (a_k p^k + a_{k-1} p^{k-1})] *$

$[(a_k p^k + a_{k-1} p^{k-1} + 1) * (a_k p^k + a_{k-1} p^{k-1} + 2) * \ldots * (a_k p^k + a_{k-1} p^{k-1} + a_{k-2} p^{k-2})] *$

……

……

$(a_k p^k + a_{k-1} p^{k-1} + \cdots + a_1 p + 1) * (a_k p^k + a_{k-1} p^{k-1} + \cdots + a_1 p + 2) * \ldots * (a_k p^k + a_{k-1} p^{k-1} + \cdots + a_1 p + a_0)]$.

By extending Lemma 2.3.2, the number of appearance of p inside them is the same as that inside the following expressions:

$[1 * 2 * \ldots * a_k p^k]$, $[1 * 2 * \ldots * a_{k-1} p^{k-1}]$, $[1 * 2 * \ldots * a_{k-2} p^{k-2}]$, ……, $[1 * 2 * \ldots * a_1 p]$, and $[1 * 2 * \ldots * a_0]$.

Hence, by Lemma 2.3.4, the number of appearance of p in $n!$ is $a_k(p^k - 1)/(p - 1) + a_{k-1}(p^{k-1} - 1)/(p - 1) + \ldots + a_1(p^1 - 1)/(p - 1) + a_0(p^0 - 1)/(p - 1)$, or $\sum_{i=1}^{k} \frac{a_i(p^i - 1)}{p - 1}$, because the last term is zero.

Let us take an example — 338!.

Express 338 in base 5 would be $2323_{(\text{base5})}$ or $2(5^3) + 3(5^2) + 2(5^1) + 3(5^0)$.

Let us say Fi5(x) is the number of factors of 5 for x.

$$\begin{aligned}\text{Fi5}(338!) &= Fi5(2(5^3)!) + FiP(3(5^2)!) + FiP(2(5^1)!) + FiP(3(5^0)!) \\ &= 2(Fi5(5^3!)) + 3(FiP(5^2)!) + 2FiP(5^1)!) + 3(FiP(5^0)!) \\ &= 2\left(\frac{5^3-1}{5-1}\right) + 3\left(\frac{5^2-1}{5-1}\right) + 2\left(\frac{5^1-1}{5-1}\right) + 3\left(\frac{5^0-1}{5-1}\right) \\ &= 62 + 18 + 2 + 0 = 82\end{aligned}$$

2.3.5 Intuition of the proof

$$\text{if n} = p^k, \text{then the number of factors of p} = 1 + \text{p} + p^2 + \cdots + p^{k-1} = \frac{p^k - 1}{p - 1}$$

Let's say:

$$factors\ in\ n!\ are\ 1,2,3,\dots,p,p+1,\dots,2\text{p},\dots,p^2,\dots,p^3,p^k$$

Terms that consist of factor p are $\text{p}, 2\text{p}, 3\text{p}, \dots, p^2, p^2 + p, p^2 + 2p, \dots, p^3, \dots, p^k$

By taking one p from all the factors, number of p taken would be p^{k-1}.

After taking the p, the factors would become: $1,2,3,\dots,\text{p},\text{p}+1,\dots p^{k-1}$.
The factors above would be the whole factors of p^{k-1}!.
From this we can see that next time number of factor p would be taken is p^{k-2}.
And the next time number of factor p would be taken is p^{k-3}.

From this, we can see that the total number for factors of p being factored out along the progress would be $\text{p}^{\text{k-1}} + \text{p}^{\text{k-2}} + \text{p}^{\text{k-3}} + \dots + \text{p}^2 + \text{p} + 1$, or it could be stated as following:

$$\text{For factorial function n!, when n} = p^k, \text{the occurrence for factor p in n! would be } \frac{p^k - 1}{p - 1}$$

From the result about, we can concluded that when $\text{n} = \text{p}^{\text{k}}$ for factorial function n!, the factorial program output could be tested by extracted all factors of p to see if the number of factors p would be match the number $((\text{p}^{\text{k}} - 1) / (\text{p}-1))$. Since n! in the form of p^{k}! can

produce very huge number with just small value of p and k, hence it would just suitable for huge factorial number.

Let's say $5^k! < n! < 5^{k+1}!$,
then $n = a_0(5^k) + a_1(5^{k-1}) + a_2(5^{k-2}) + a_3(5^{k-3}) + \cdots + a_{k-1}(5^1) + a_k(5^0)$
after finding value of coefficients a_0, a_1, a_2, …, a_{k-1}, a_k,
writing to these coefficients together is convert n onto a number in base 5.

For an example, if q = 407 in q! , then $5^3! < q! < 5^4!$, $q = 3(5^3) + 1(5^2) + 1(5^1) + 2(5^0)$,
q written in base 5 is writing coefficients together = $3112_{(\text{base } 5)}$.

Moreover, 5^k and 5^{k+1} in base 5 would always be 0100…000 and 1000…000, like in example above 5^3 and 5^4 in base 5 would be 01000 and 10000.

In example above, range of q would be 01001 to 04444, then we can see that in the case of $5^k! < n! < 5^{k+1}!$, the range of n could be 0100…001 to 0444…444.

In n!, for finding number of factors of p when $p^k! < n! < p^{k+1}!$, it is needed to find relationship between n in base p (coefficients for a_0, a1, a2, …, ak-1 in the above) and number of factors of p.

Initially it is intended to use property for $p^k!$, let's say property(i), $p^k!$ have $\frac{p^k-1}{p-1}$ number of factors of p and add it with number of factor of p from $p^k!$ to n!, however it may suggest that there is no such need for this. After determining relationship between n in base p (coefficients for a0, a1, a2, …, ak-1 in the above) and number of factors of p, it shows the relationship is an extension to property(i).

For factorial number n!, start from $000…000_{(\text{base } p)}$, and incremental of digit in p^0 means add 1 value to n, which means in denary there is increase of one more multiplying term for

n!. However incremental in digits (or coefficient) of p^0 (digits until p-1) gives no multiplying term that contain factor of p.

Each incremental in digits for p^1(until p-1) give p multiplying terms for n! in denary where one term of these term would contain one p factor.

Each incremental in digits for p^2(until p-1) give p^2 multiplying terms for n! in denary where p-1 term of these term would contain one factor of p and one of these terms contain two factors of p.

The illustration is as following:
$1*2*3*\ldots*p*\ldots.*2p*\ldots*3p*\ldots* p^2 * \ldots* 2p^2 * \ldots * p^3 * \ldots * 2p^3*\ldots$

For every p multiplying terms (1 to p, p to 2p, 3p to 4p,…, with not reaching p^2) there contain 1 factor of p.

When counting factors of p from term 1 until p^2, where would be p(1) + 1 factors of p where the term p^2 give (one extra) factor of p. Hence when incrementing one digit of p^2 give p+1 factors of p.

When counting factors of p from term 1 until p^3, where there would be p(p+1) +1 factors of p where the term p^3 give (one more extra factor) of p. Hence when incrementing one digit of p^3 give $p^2 + p + 1$ factors of p.
As a result, it could be seen that incrementing one digit of p^4 give $p^3 + p^2 + p + 1$ number of factors of p.

Incrementing one digit of p^5 give $p^4 + p^3 + p^2 + p + 1$ number of factors of p.
We can conclude that an incremental of digit in p^k given n in base p results in $p^{k-1} + p^{k-2} + \ldots + p + 1 = (p^k - 1)/ (p - 1)$ number of factors of p which is the similar result of properties(i).

Below are some extra properties that we also can conclude:

Let's say FiP(x) is finding number of factors of p for x.

Then

$$\text{FiP(n!)} = FiP(a_0p^k!) + FiP(a_2p^{k-1}!) + FiP(a_3p^{k-2}!) + \cdots + FiP(a_{k-1}p^1!) + FiP(a_kp^0!),$$

$$\text{where n} = a_0p^k + a_2p^{k-1} + a_3p^{k-2} + \cdots + a_{k-1}p^1 + a_kp^0$$

And

$$\text{FiP}(\text{a}(p^k)!) = \text{a}\Big(\text{FiP}(p^k!)\Big), \text{given that } 0 \leq \text{a} < \text{p}$$

2.3.6 For the result in Lemma 2.3.4, stating how can it be verified by experiment

Basic logic flows of program:

```
count = 0;
x, p , k  << input;
while (x % p == 0){
        x = x / p;
        ++count;
}
```

If count == $((p^k - 1) / (p-1))$

```
    return true;
else
    return false;
```

First input value of n in the form of p^k to get output from the factorial program shows, then the rest modulus and division is the only mathematical operations for performing this experiment. Since it need to be taken out all the factors of p, so just

1) Initially taking the output of factorial program modulus by p to see if it equals to zero (means the output contain factor of p.). If equal to zero go to step 2, otherwise go to step 3.
2) Divide the output by p and increase the counting of factor p by one. Return to step 1 again using divided output.

3) Checking the counting of factor p would be same as ((pk – 1) / (p-1)).

4) If the counting result is same then it would verify the statement in (b).

2.3.7 Testing factorial numbers through prime factorization

By using prime factorization to get out first few prime factors and its occurrence for factorial number, the properties of prime factors could give some information that if factorial program output is faulty. When there is some alteration on value of actual factorial number, changes on these properties is significant even for small value prime factors (<13).

The basic flow for prime factorization (from up to down) would be:

```
n = 2;
x << input;
Primefac(x){
        if x < n
                return;
        if (x % n != 0)
                ++n;
        else
                x = x / n;
                output n;
         Primefac (x);
}
```

Below shows some result for prime factorization of factorial numbers (for prime numbers until 13) by program done in section done in section 3.5:

Result 1) Value used: 120!

Prime factorization result: 2 ^ 116 + 3 ^ 58 + 5 ^ 28 + 7 ^ 19 + 11 ^ 10 + 13 ^ 9 + ...

Result 2) Value used: 120! (with one digit changed)

Prime factorization result: 2 ^ 116 + 5 ^ 28 + ...

Result 3) Value used:120! (added value of 3 on third digit)

Prime factorization result: 2 ^ 116 + 3 ^ 1 + 5 ^ 28 + ...

Result 4) Value used:120! (added value of 5 on last 8th digit)

Prime factorization result: 2 ^ 10 + 5 ^ 7 + ...

Result 5) Value used:120! (added value of 2*3*5*7*11*13 or 30030)

Prime factorization result: 2 ^ 1 + 3 ^ 1 + 5 ^ 1 + 7 ^ 1 + 11 ^ 1 + 13 ^ 1 + ...

Result 6) Value used: 120! (added with 2*12 or 18446744073709551616)

Prime factorization result: 2 ^ 64 + ...

Result 7) Value used: 120! (added with 2*3+3*3+5*3+7*3+11*3+13*3 or 123)

Prime factorization result: 3 ^ 1 + ...

Exploration of the testing results and prime number properties for factorial is as below:

2.3.8 Lower value prime factors always have higher occurrence to higher value prime factors

For factor of 2, its number in factorial number incremental by at least once for every incremental value of n in n! starting from 2.

For factor of 3, its number in factorial number incremental by at least once for every incremental value of n in n! starting from 3.

…….

Lower value prime factors always increase its occurrence in factorial number faster than higher value along with n. The significant of this property shown in Result 3 and 5. In result 3, factor of 3 have higher occurrence than 5. In result 5, prime factors from 2 to 13 share the same occurrence.

2.3.9 All prime numbers with value lower than n for n! must be exist

Since n! is divisible by all numbers from one to n, n! must divisible by all prime numbers lower than n. Hence all prime numbers lower than n have to be existed.

The significant of this proper shown property shown in Result 2, 3, 4 ,6 ,7. All these results show the altered factorial number causing missing of prime factors even the factors are in small value.

2.3.10 Changes of prime factors occurrence when value of actual prime number changed

When there is changes on the value of factorial number, there is chances occurrence of prime factors changes. For example, in result 4, occurrence for factor of 2 changed from 116 to 10 and factor of 5 from 28 to 7.

The way for finding occurrence of prime factor for actual prime number is as following:

1) For n! and prime a, divide n by a and get the value (without remainder.) If the value is 0 then go to step 5. Else increment power of a by one and repeat step 1.
2) Add up all the value that done step 1, then it is the occurrence of factor a for n!.

This property is only suitable for testing factorial numbers by just small value prime factors if there is no list of prime factors for referring.

2.3.11 Extra Information

When there is changes value of factorial number n!, it has chances that there would be occurrence of prime factor with higher value than n, however, the computation for testing out this prime factor would be require much effort to done, which is not suitable for performing program testing.

2.4 Metamorphic testing with combination function

The idea of metamorphic testing is to test a function by using the same function. The general combination function is as below:

$$\binom{n}{r} = \frac{n!}{\mathrm{r}!\,(n-r)!}$$

By changing the formula above, we could get a new definition for factorial number.

$$\mathrm{n}! = \mathrm{r}!\,(n-r)!\binom{n}{r}$$

where r = 1,2,3,4 ,… , n-1. We use the function to output n!, r!, and (n-r)!, and then use the above formula for verification. $\binom{n}{r}$ can be computed in O(r) step, but each step is very long integer multiplication or division. This is a method to test the factorial function.

In order to reduce computation time, r is replaced as n/2 as below:

$$\mathrm{n}! = \left\lceil \frac{n}{2} \right\rceil ! \left\lfloor \frac{n}{2} \right\rfloor ! \binom{n}{\frac{n}{2}}$$

With formula above, difference between $\left\lceil \frac{n}{2} \right\rceil$ and $\left\lfloor n - \frac{n}{2} \right\rfloor$ would not larger than 1 which reduce the computation time when applying metamorphic testing.

By "metamorphic", we refer metamorphic testing to a testing technique that using metamorphic relation between inputs and outputs to verify the correctness of an output. In this case, this project is using the factorial number defined inside the formula ($\left\lceil \frac{n}{2} \right\rceil !$ and $\left\lfloor \frac{n}{2} \right\rfloor !$) to test correctness of n! output. Factorial number defined inside the formula ($\left\lceil \frac{n}{2} \right\rceil !$ and $\left\lfloor \frac{n}{2} \right\rfloor !$) could be further defined further through recursive function.

For example, let's say $\left\lceil \frac{n}{2} \right\rceil$ is even number, then it could be further defined onto

$$\left\lceil \frac{n}{2} \right\rceil ! = \left\lceil \frac{n}{4} \right\rceil ! \left\lfloor \frac{n}{4} \right\rfloor ! \binom{\frac{n}{2}}{\frac{n}{4}}$$

And could be further defined again until $\left\lceil \frac{n}{k} \right\rceil$ and $\left\lfloor \frac{n}{k} \right\rfloor$ equal to one. The recursive function could be said being applied using metamorphic testing.

One drawback of this method is the possibility of getting wrong n! and $\left\lceil \frac{n}{2} \right\rceil !$ while the equality $n! = \left\lceil \frac{n}{2} \right\rceil ! \left\lfloor \frac{n}{2} \right\rfloor ! \binom{n}{\frac{n}{2}}$ still holds. For simplicity, we make that such probability is no more than 1/10.

Precisely, the probability that the program is faulty and test result is $n! = \left\lceil \frac{n}{2} \right\rceil ! \left\lfloor \frac{n}{2} \right\rfloor ! \binom{n}{\frac{n}{2}}$ would be 1/10, then given the program is faulty and $n! \neq \left\lceil \frac{n}{2} \right\rceil ! \left\lfloor \frac{n}{2} \right\rfloor ! \binom{n}{\frac{n}{2}}$ is 9/10, for one test case.

$$\text{if } p((\text{LHS} = \text{RHS}) | \text{wrong program}) = \frac{1}{10},$$

$$\text{then } p((\text{LHS} \neq \text{RHS}) | \text{wrong program}) = \frac{9}{10},$$

Because we can use r=1, 2, …, n-1, for the test cases, we randomly choose log n numbers, out of n-1 possible values of r, where base-10 logarithm is applied. Now, p((LHS=RHS in all log n trials)|wrong program) = $(1/10)^{\log n}$ = 1/n. The probability 1/n is considered low in the literature. However this method still considered using moderate high computation time.

CHAPTER 3: CONCLUSION

Theoretically untestable cases are usually impossible to have a method that could confirmed the correctness of output with ease and efficiently. However, further improvements on the test case could be done for relieving the problem. Factorial number is not only difficult to test due to its large output but also its high computation complexity making efficient alternative testing method mostly unavailable.

Testing by applying Wilson theorem could be used as earlier testing due to its fast computing time. It is important to note that when applying Wilson theorem value of n in n! for testing is defined randomly in order to achieving fast computation time. The random n value is gone through primality test to find its primality which only need polynomial time, conversely if n is defined intentionally along with its primality property, a high computation time is needed.

For length-testing, it also could be used as an earlier test case due to its low computation time. One advantage of length-testing is this method would not be interfering with the factorial number program output. The only thing that need to do after extracting output result is counting length of it (number of digits) and compare with expected length value. Since formula for calculating the expected length would give long decimal number before getting the lower floor, truncation had to be done during calculation. In this project show until which point after the decimal point had to be truncated in order not to interfering with the final result (given wrong length value).

In this project also show prime counting (through prime factorization) could give distinct result to the faulty output. Through prime counting, it is easily to see the deviation on occurrence of prime factors on the program output with actual occurrence of prime factors based on n defined for the n!. With faulty output, there could be some other properties changes on the prime factors, included missing prime factor and lower prime factors have lower occurrence on higher prime factors.

For metamorphic-testing, this new technique in software testing could help to relieving the difficulty of black box testing on untestable cases. Metamorphic relation is done between input and output, which in this project the relation of factorial number is defined through the definition of combination function. What make the definition unique is through the definition of combination function, the new definition of factorial formula also contain another factorial number, and can be used for further testing.

BIBLIOGRAPHY

[1] T barr et al.. 2015. The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering. 41(5), pp. .

[2] Liu, H., Kuo, F., Towey, D. and Chen, T. (2014). How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?. IEEE Transactions on Software Engineering, [online] 40(1), pp. 422. Availableat:http://vuir.vu.edu.au/33046/1/TSEmt.pdf.

[3] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, August 2001. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

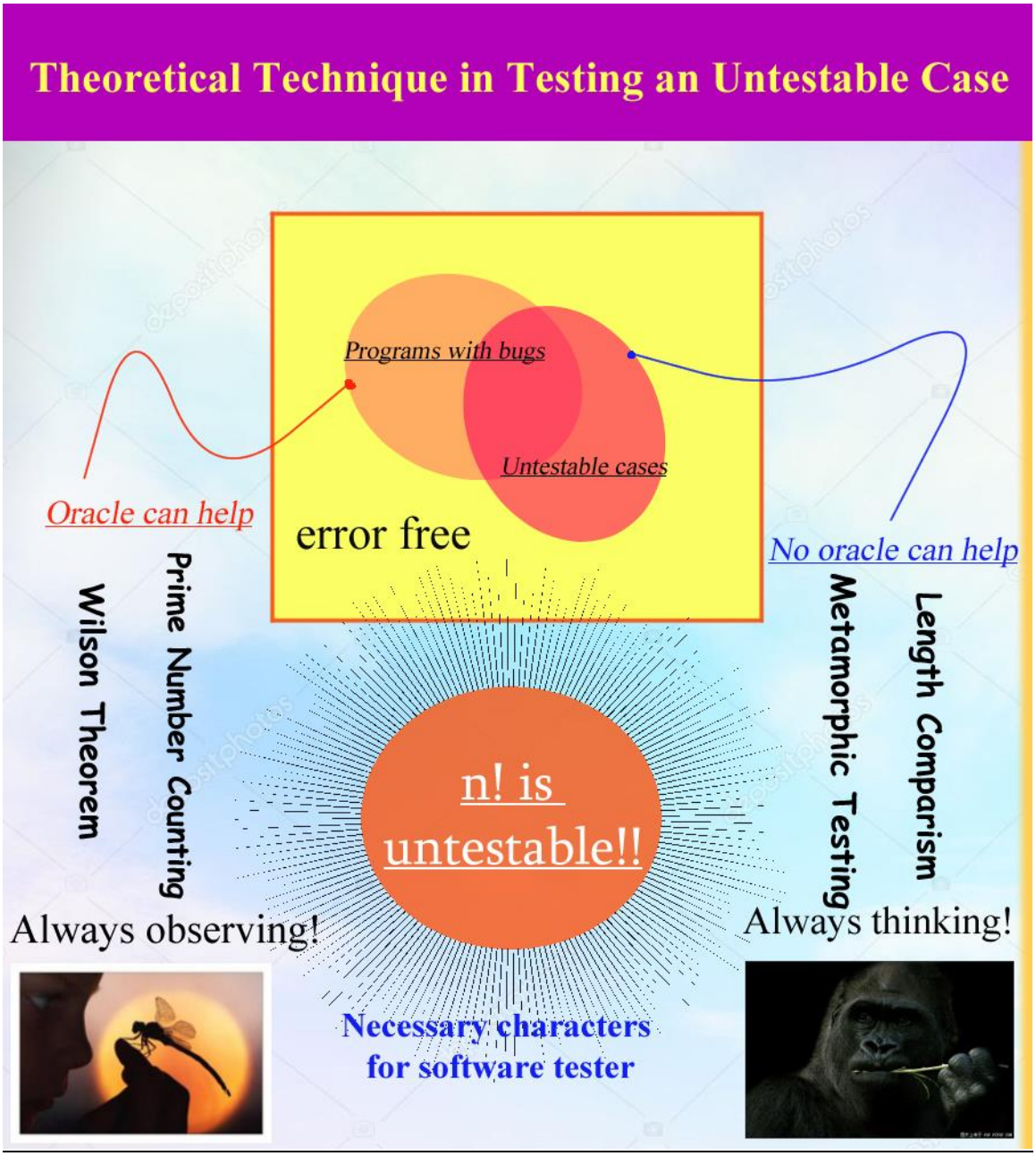
[4] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In Proceedings of the IASTEDInternational Conference on Software Engineering, page, 1998.

[5] Elaine J. Weyuker. On testing non-testable programs. The Computer Journal, 25(4):465–470, November 1982.

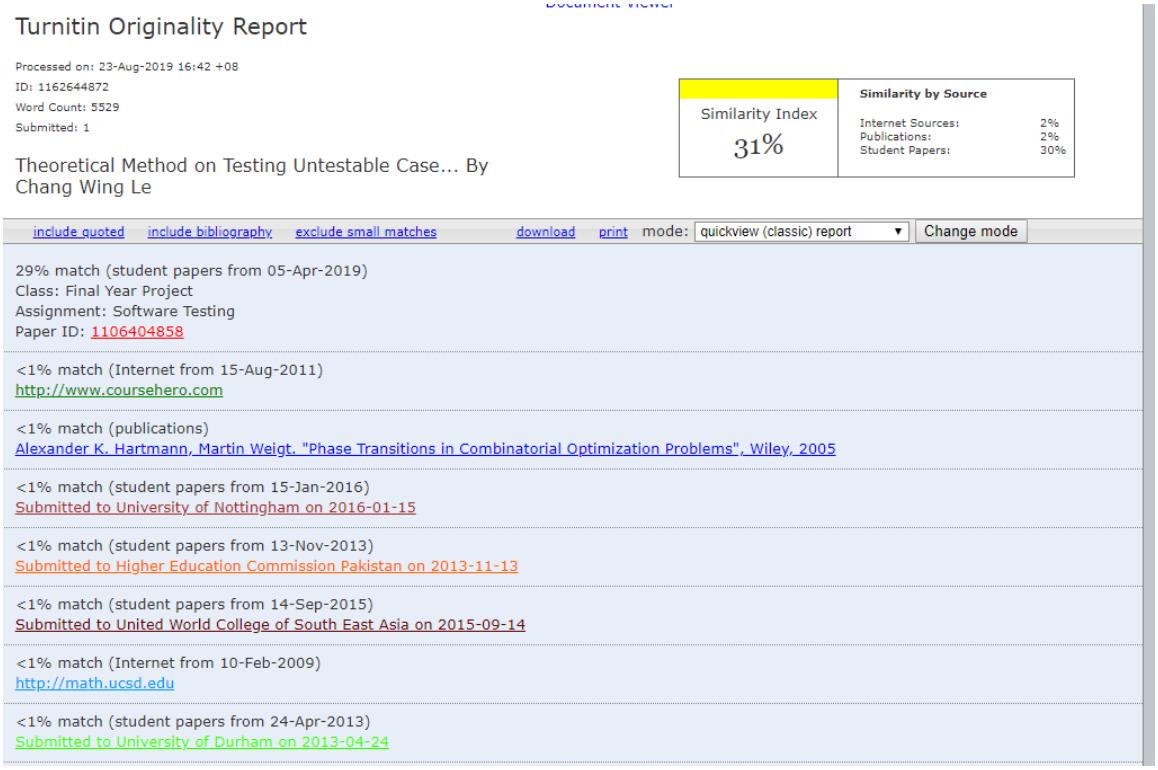
[6] Maninda Agrawal, Neeraj Kayal and Nitin Saxena.(2004). PRIMES is in P. Available at:https://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf

[7] R. Andrew Ohana. (2009). A Generalization of Wilson's Theorem. Available at: https://sites.math.washington.edu/~morrow/336_09/papers/Andrew.pdf

Poster


Theoretical Technique in Testing an Untestable Case
Programs with bugs
Untestable cases
error free
Oracle can help
No oracle can help
Wilson Theorem
Prime Number Counting
Metamorphic Testing
Length Comparism
n! is untestable!!
Always observing!
Always thinking!
Necessary characters for software tester

Turnitin Result



Turnitin Originality Report

Processed on: 23-Aug-2019 16:42 +08
ID: 1162644872
Word Count: 5529
Submitted: 1

Theoretical Method on Testing Untestable Case... By Chang Wing Le

Similarity Index	Similarity by Source	
31%	Internet Sources:	2%
	Publications:	2%
	Student Papers:	30%

include quoted | include bibliography | exclude small matches | download | print | mode: quickview (classic) report | Change mode

29% match (student papers from 05-Apr-2019)
Class: Final Year Project
Assignment: Software Testing
Paper ID: 1106404858

<1% match (Internet from 15-Aug-2011)
http://www.coursehero.com

<1% match (publications)
Alexander K. Hartmann, Martin Weigt. "Phase Transitions in Combinatorial Optimization Problems", Wiley, 2005

<1% match (student papers from 15-Jan-2016)
Submitted to University of Nottingham on 2016-01-15

<1% match (student papers from 13-Nov-2013)
Submitted to Higher Education Commission Pakistan on 2013-11-13

<1% match (student papers from 14-Sep-2015)
Submitted to United World College of South East Asia on 2015-09-14

<1% match (Internet from 10-Feb-2009)
http://math.ucsd.edu

<1% match (student papers from 24-Apr-2013)
Submitted to University of Durham on 2013-04-24

The similarity result is high is due to my FYP report had high similarities.

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY (KAMPAR CAMPUS)

CHECKLIST FOR FYP1 THESIS SUBMISSION

Student Id	
Student Name	
Supervisor Name	

TICK (√)	DOCUMENT ITEMS Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
	Title Page
	Signed form of the Declaration of Originality
	Abstract
	Table of Contents
	List of Figures (if applicable)
	List of Tables (if applicable)
	List of Symbols (if applicable)
	List of Abbreviations (if applicable)
	Chapters / Content
	Bibliography (or References)
	All references in bibliography are cited in the thesis, especially in the chapter of literature review
	Appendices (if applicable)
	Poster
	Signed Turnitin Report (Plagiarism Check Result – Form Number: FM-IAD-005)

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report. _____ (Signature of Student) Date:	Supervisor verification. Report with incorrect format can get 5 mark (1 grade) reduction. _____ (Signature of Supervisor) Date:

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY (KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	
Student Name	
Supervisor Name	

TICK (√)	DOCUMENT ITEMS Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
	Front Cover
	Signed Report Status Declaration Form
	Title Page
	Signed form of the Declaration of Originality
	Acknowledgement
	Abstract
	Table of Contents
	List of Figures (if applicable)
	List of Tables (if applicable)
	List of Symbols (if applicable)
	List of Abbreviations (if applicable)
	Chapters / Content
	Bibliography (or References)
	All references in bibliography are cited in the thesis, especially in the chapter of literature review
	Appendices (if applicable)
	Poster
	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report. _____ (Signature of Student) Date:	Supervisor verification. Report with incorrect format can get 5 mark (1 grade) reduction. _____ (Signature of Supervisor) Date: