YAŞAR UNIVERSITY
GRADUATE SCHOOL

MASTER THESIS

RIGOROUS ANALYSIS OF EFFICIENCY

TECHNIQUES OF SOFTWARE ALGORITHMS

ATABARIŞ AYAYDIN

THESIS ADVISOR: PROF. DR. MEHMET SÜLEYMAN ÜNLÜTÜRK

COMPUTER ENGINEERING MASTER PROGRAM

PRESENTATION DATE: 15.6.2020

BORNOVA / İZMİR
JUNE 2020

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Jury Members:	**Signature:**
Prof. Dr. Mehmet Süleyman ÜNLÜTÜRK Yaşar University	.....
Dr. Korhan KARABULUT Yaşar University	.....
Assoc. Prof. Dr. Gökhan DALKILIÇ Dokuz Eylül University	.....

Prof. Dr. Cüneyt GÜZELİŞ
Director of the Graduate School

ABSTRACT

RIGOROUS ANALYSIS OF EFFICIENCY TECHNIQUES OF SOFTWARE ALGORITHMS

Ayaydın, Atabarış
Msc, Computer Engineering
Advisor: Prof. Dr. Mehmet Süleyman ÜNLÜTÜRK
June 2020

Efficiency, in programming, generally treated as a concept of "on-demand" rather than an integral part of the programming. However, as it is a part of the software quality measurements, the programmer also responsible to write a program that will meet the requirements. Since there is no known technique to find the least time or space complexity for the problem on the hand, augmenting the programmer's knowledge with the known techniques is essential. As the meaning of efficiency changes throughout the time, these mentioned techniques must be reevaluated to adapt to current necessities. This thesis address, the categorization of the mentioned techniques as well as the expansion of them. The runtime comparison between the different versions of the solutions states that efficiency is not a lesser subject to deal with, instead, it requires more attention than it gets.

Key Words: efficiency techniques, performance, optimization

ÖZ

YAZILIM ALGORİTMALARININ VERİMLİLİK TEKNİKLERİNİN TİTİZ ANALİZİ

Ayaydın, Atabarış
Yüksek Lisans Tezi, Bilgisayar Mühendisliği
Danışman: Prof. Dr. Mehmet Süleyman ÜNLÜTÜRK
Haziran 2020

Programlama da verimlilik, genellikle programlamanın ayrılmaz bir parçası olarak görülmek yerine ihtiyaç üzerine yapılan bir kavram olarak görülmektedir. Halbuki, yazılım kalite ölçütlerinin bir parçası olması nedeniyle, programcı aynı zamanda gerekli ihtiyaçları karşılayacak bir program yazmakla da yükümlüdür. Bir problem için var olan en az zaman ya da alan karmaşıklığını bulabilecek bir teknik var olmaması nedeniyle, programcının, bilgisini var olan teknikler ile arttırması gerekmektedir. Verimliliğin anlamı yıllar içerisinde değişiklik gösterdiği için, bahsi geçen teknikler güncel ihtiyaçlara uygulanabilmek adına yeniden değerlendirilmelidir. Bu tezde, bahsi geçen tekniklerin gruplandırılması ve ek olarak genişletilmesi ele alınmaktadır. Çözümlerin farklı sürümlerinin çalışma süreleri arasındaki karşılaştırmalar, verimliliğin hafife alınacak bir konu olmadığını, aksine, olduğundan daha fazla dikkat gerektirdiğini ifade etmektedir.

Anahtar Kelimeler: verimlilik teknikleri, performans, optimizasyon

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor MEHMET SÜLEYMAN ÜNLÜTÜRK for his guidance and patience during this study.

I would like to express my enduring love to my family, who are always supportive in every possible way in my life.

I would like to thank all my research assistant friends at Yaşar University who always supported me and helped me with the best of their abilities.

It was my decision to choose software engineering, it was my fortune to study at Yaşar University.

Atabarış Ayaydın

İzmir, 2020

TEXT OF OATH

I declare and honestly confirm that my study, titled “RIGOROUS ANALYSIS OF EFFICIENCY TECHNIQUES OF SOFTWARE ALGORITHMS” and presented as a Master’s Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Atabarış Ayaydın

Signature

……..

July 14, 2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SYMBOLS AND ABBREVIATIONS

ABBREVIATIONS:

WBG White Black Gray

JVM Java Virtual Machine

JDK Java Development Kit

ASCII American Standard Code for Information Interchange

RPS Rock, Paper, Scissor

SYMBOLS:

µs/op unit to represent milliseconds

CHAPTER 1
INTRODUCTION

Programming is a concept that has many challenges to deal with. These challenges range from the definition of a problem to the implementation of the solution and more. To cope up with these challenges, many metrics, methods, and techniques are defined, however, agreeable ones tend to be questionable as the challenge becomes more fundamental.

Since the programming concept has a very large scope related to many other topics, the methods, definitions, etc. may need to be updated, extended, or rewritten from scratch. The "efficiency" of a program as one of these topics is addressed in this chapter.

1.1. Definitions

The general idea of a program to be "efficient" is, it must use little time or space. Boundaries for these usage limits are generally drawn with complexity analysis on the problem, however, it is not possible to know exact limits for the least time or space usage on all problems that can be encountered. Therefore, the usage of "little" time or space depends on the computation power of a machine with the chosen algorithm so, it has arguable validity. As an example, assume that a program runs in 0.02 milliseconds. From our perception of understanding the time, it is faster than the blink of an eye so, it may seem efficient enough. However, if another program with a different approach can achieve the same thing with 0.0002 milliseconds, statistically the first program cannot be defined as "efficient" since the second program is a hundred times faster than the original. Another example is, assume that a program has $O(2^n)$ complexity. The asymptotic notation of algorithm indicates, this algorithm is classified in exponential complexity, therefore, trying to solve this problem will not be "efficient" in the Turing machine. However, the program itself, maybe using the best algorithm that can be achievable (due to the nature of the problem) so, it is "efficient" but the implementation on the machine is not suitable to apply so, it becomes inefficient.

1.2. Limits

The efficiency of a program can be improved in different design levels of computer systems independently. This degree of freedom may provide various potential ways for improvements, however, changes in design levels such as hardware or system software may become too specific so that executing the program in another environment may not be possible at all. A program that uses processor-specific instructions or operating system specific procedures for efficiency will have less portability to other application environments. For this reason, such changes in a program should meet specific conditions and, should be considered as a last resort except for programs with specific target environment requirements.

Programs are generally written with the same programming language for their entire lifecycle. Programming languages tend to be updated from time to time due to accommodate current needs, new technologies, bug fixes, etc. So, some of the functionalities that have been used for efficiency may deprecate in time, which will be a problem for a program aimed to be used in long term.

1.3. Psychology

In general, when the efficiency of the program is addressed by articles, books, etc. it is evaluated or mentioned as a subject solely related to the machine. However, we should not forget that every program is written by a programmer(s). Therefore, the psychology of the programmer is also a dimension that should be considered along with other topics related to efficiency.

The limitations mentioned in the upper section are about topics that can be encountered while dealing with the efficiency of a program. However, the programmer's limitation to himself always will be in this process. Hence, the programmer's knowledge about the problem, selected algorithm, programming language, operating system, etc. are important as much as any methodology or technique produced to deal with efficiency.

As an example, undergraduate students were given the task of writing a C program, that prints an integer on the form of binary string and one of the simplified (without error or boundary checking) solution given in below:

```
void intToBinary(int number) {
    int bit_count = log2(number) + 1;
    int bits[ bit_count];
    int index = bit_count - 1; //last index
    while (number > 0) {
        bits[index] = number % 2;
        number = number / 2;
        index = index - 1;
    }
    for (index = 0; index < bit_count; index++) {
        printf("%d", bits[index]);
    }
}
```

Figure 1.1. Integer to Binary in C

The same task is given in another undergraduate lesson with Java language and solutions generally had the same structure (looping through the array to print indexes). A representative code can be written as:

```
public static String intToBinaryV1(int number) {
    ArrayList<Integer> bits = new ArrayList<>();
    String binaryString = "";
    while (number > 0) {
        bits.add(number % 2);
        number = number / 2;
    }
    for (int index = bits.size() - 1; index >= 0; index--) {
        binaryString = binaryString + bits.get(index).toString();
    }
    return binaryString;
}
```

Figure 1.2. Integer to Binary in Java Version 1

Another version of the code written as:

```
public static String intToBinaryV2(int number) {
    String binaryString = "";
    while (number > 0) {
        binaryString = Integer.toString(number % 2) + binaryString;
        number = number / 2;
    }
    return binaryString;
}
```

Figure 1.3. Integer to Binary in Java Version 2

Comparing both versions of Java codes, it can be easily deduced that version 2 is more efficient since it does not create extra space to store digits. At that time, students had enough knowledge to use necessary data structures and string rules in Java language, but they did not implement the left string concatenation. The concatenation rule in their daily language also in English, adds new letters to the right side of the existent word, even if Java offers concatenation from both sides of the string, most of them could not think this simple change, which resulted with a program that uses more space and time.

1.4. Techniques

Writing an efficient program is not an issue that can be directly related to experience, intelligence, or programming language. Their impact on coding is not negatable however, there are also other topics that should be considered. As mentioned before, the exact limits for efficiency are not known for all problems that can be encountered. Therefore, augmenting a person's knowledge with the techniques make the program more effective and channeling the thoughts about the problem, to more specific points rather than heuristic decisions is a good starting point in this way. Throughout the thesis, these techniques explained on example problems with different implementation or variation of solutions. Each initial solution that is not referenced to someone else's work written with the idea of solving the problem without the consideration of efficiency as its primary objective. Even though both the initial and the altered versions of the solutions written by the same person, the initial solutions are in a generic form that can found in various online resources. A couple of problems with the application of these techniques are presented in this section.

1.4.1. Roman Numerals

In a second-year undergraduate software engineering course, the instructor gave the task of writing a Java program that converts given integer to Roman Numerals. To keep it simple and focusing on the efficiency of the solution, assume that given input is always a valid integer between 1 and 999. After some time, he shared his solution and talked about why some of the students were not able to produce a fully functional code or any piece of code at all. His solution did not have any advanced level structure or language-specific instruction, moreover, it was so simple and clear that it could be converted to any language with a few line changes. Example code was written by a student is given below.

```
1   public static String convertToRomanV1(int number) {
2       int i = 0;
3       ArrayList<String> digits = new ArrayList<>();
4       String[][] romans =
5               {
6                       {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"},
7                       {"", "X", "XX", "XXX", "XL", "L", "LX", "LXXX", "XC"},
8                       {"", "C", "CC", "CCC", "CD", "D", "D", "DC", "DCC", "DCCC", "CM"}
9               };
10      String result = "";
11      while (number > 0) {
12          digits.add(romans[i][(number % 10)]);
13          number = (number - (number % 10)) / 10;
14          i++;
15      }
16      for (int j = digits.size() - 1; j >= 0; j--) {
17          result = result + digits.get(j);
18      }
19      return result;
20  }
```

Figure 1.4. Integer to Roman Numeral in Java Version 1

In addition to this solution, much fewer effective solutions can be found in (Java & Junior, 2020). Based on the analysis of the problem, there are a couple of topics that should be considered:

Core problem: From the relationship between Roman numerals and integer numbers, it can be implied that finding an efficient algorithm for the conversion of numbers will determine the general complexity of the given problem. Since the conversion of numbers is a digit based, a loop (lines 11-15) written, that takes the last digit of the number and reduces the number by a factor of ten for shifting the next digit to be last. Is there a better way to accessing digits in a number?

Data Structure: Each digit in an integer has a unique equivalent in Roman numerals, so any key-based structure can be selected for mapping integer to Roman numeral, however, since the key value is a one-digit integer, it is more convenient to choose a structure that uses integer numbers as its index for accessing required data. Therefore, a simple array structure can provide constant access time. The written code is built upon a logic of shifting the last digit in each iteration, so instead of a three-piece one-dimensional string array, it uses a more complex two-dimensional one (lines 4-9). Is this approach sufficient enough for the problem, or can it be adjusted to increase its efficiency?

Algorithm Design: If it is possible, finding an algorithmic approach that is not a limitation for other decisions is a good one to start with. In the above solution, the

approach used for the core problem enforces accessing to digits from rightmost to left. The problem is, proceeding with that logic will not be able to handle the construction of the result with the access of digits in the same iteration (thought so). For this reason, another array structure is used to store results for each digit (line 3) then, traversed the array in reverse order (lines 16-18) to construct the desired output. Can this extra array be removed with another approach?

1.4.1.1 Adjustments

Restricting to the thought of "traversing from right to left and constructing result in reverse order cannot be achieved in the same iteration" is an example of a programmer's limitation to himself. For a more efficient program without extra space usage, all that needs to be done is to concatenate digits to the right side of the result variable, so the loop inline 11-15 is replaced with:

```
10      while (number > 0) {
11          result = romans[i][(number % 10)] + result;
12          number = (number - (number % 10)) / 10;
13          i++;
14      }
```

Figure 1.5. Roman Numerals Adjustment 1

With this change, the extra loop between lines 16-18 and storage inline 3 for constructing the result is removed. Still, the code looping through number to access each digit in the number. Using a while loop may seem logical since the digits of inputs may vary between one and three, but it would be better if there was a way to accessing the Nth digit of a number without iterating through it. That type of approach would also allow a programmer to use a simple three-piece one-dimensional array instead of two dimensional. Appling numerical analysis on this problem shows that starting from rightmost digit, any desired digit on a given integer number can be accessed with $(number\%10^{k}) \ / \ (10^{k-1})$, where k represents the position of digit starting from the rightmost digit with initial value 1.

As mentioned before, the upper formula allows to use one-dimensional arrays, so the lines between 4-9 are replaced with:

```
2       String[] units = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"};
3       String[] tens = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXXX", "XC"};
4       String[] hundreds = {"", "C", "CC", "CCC", "CD", "D", "D", "DC", "DCC", "DCCC", "CM"};
```

Figure 1.6. Roman Numerals Adjustment 2

In addition to these changes, one of the most important advantages of the upper data structure is the first element of arrays (“”). Being able to replace 0 with an empty character allows using conversion exactly 3 times (since the highest possible integer input is 999) instead of digit count based while loop. So, the lines between 11-15 were replaced with:

```
5       return hundreds[(number % 1000) / 100] + tens[(number % 100) / 10] + units[(number % 10)];
```

Figure 1.7. Roman Numerals Adjustment 3

The final code can be written as:

```
public static String convertToRoman(int number) {
    String[] units = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"};
    String[] tens = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXXX", "XC"};
    String[] hundreds = {"", "C", "CC", "CCC", "CD", "D", "D", "DC", "DCC", "DCCC", "CM"};
    return hundreds[(number % 1000) / 100] + tens[(number % 100) / 10] + units[(number % 10)];
}
```

Figure 1.8. Instructor solution of Roman Numerals

As a result, throughout all changes, the initial program is speed up more than 10 times. The detail of these speedups with corresponding adjustments can be found in Table 1.4.1 3.2 in Appendix 3.

Prime Numbers

Prime numbers are a sequence of numbers that have very interesting properties and due to these properties, they influenced most of the mathematicians and programmers throughout history. Since there is no exact formula for finding all prime numbers, writing a program to find all primes in a range between 2 and n (a positive integer number to set upper bound for range) is an interesting topic to deal with.

Without putting too much thought or formal knowledge about prime properties, a straightforward solution to get all primes in a range will be a brute-force method, which is to try to divide every number by 2 through n-1. If a number is evenly divisible by any other number that is smaller than it, that number is skipped and the next one

checked with the same method. An application of this algorithm in Java can be represented as:

```java
public static ArrayList<Integer> primesV1(int limit) {
    ArrayList<Integer> prime_list = new ArrayList<>();
    boolean isPrime;
    for (int number = 2; number < limit; number++) {
        isPrime = true;
        for (int divider = 2; divider < number; divider++) {
            if (number % divider == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            prime_list.add(number);
        }
    }
    return prime_list;
}
```

Figure 1.9. Prime Numbers in Java Version 1

This code supplies desired result for various limit values, however, the complexity analysis on this solution shows that the code complexity is $O(limit^2)$, which indicates that for the higher values of limit, the necessary time for finding primes will increase quadratically.

To make this program more efficient, there is no need to try to deal with even numbers since every prime except 2 is odd. Also, the range of numbers for divisibility can narrow down to the square root of a number, due to prime factorization properties. With the mentioned changes above, the lines between 4-6 were replaced with

```java
prime_list.add(2);
    for (int number = 3; number < limit; number = number + 2) {
        isPrime = true;
        int divider_limit = (int) (Math.sqrt(number) + 1);
        for (int divider = 3; divider < divider_limit; divider = divider + 2) {
```

Figure 1.10. Prime Numbers Adjustment 1

And the final code can be written as:

```
public static ArrayList<Integer> primesV2(int limit) {
    ArrayList<Integer> prime_list = new ArrayList<>();
    boolean isPrime;
    prime_list.add(2);
    for (int number = 3; number < limit; number = number + 2) {
        isPrime = true;
        int divider_limit = (int) (Math.sqrt(number) + 1);
        for (int divider = 3; divider < divider_limit; divider = divider + 2) {
            if (number % divider == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            prime_list.add(number);
        }
    }
    return prime_list;
}
```

1.4.2.1 Figure 1.11. Final Code of Prime NumbersSieve of Eratosthenes

All the changes up to this point are made over an idea of finding primes by checking every possible number one by one, which resulted in a very inefficient program, even with all the speedups that have been added due to its complexity. This approach may seem reasonable since there is no known mathematical formula to implement and analysis on primes may consume enormous time without any decent result. In a more fundamental base, when coding for a problem, the main purpose is getting results that can lead to a feasible solution. These results may be sub-solutions for the problem or something else that can be relatable with the main solution. No one wants to code for the wrong answers since there is no benefit. However, from set theory, we know that the universal set is the union of a set and the complement of it, which can be represented as $A + A' = U$. In other words, removing all undesired results from the solution space will let the remaining subset to be the answer.

Most of the time this type of approach is not feasible due to the enormous size of solution space or the cost of time to cover it, however, in this specific problem the cost of elimination for undesired results can be decreased to a very small amount of time. Every multiple of a prime is not a prime, to be able to separate primes from the non-primes, all that is needs to be done is, sum up prime with itself till to the end of the limit. While doing that, a “remember” option should be implemented for them in some sense, so that removing non-primes from the solution space can be possible. This

remembering introduces a new dimension to the problem, but it brings new possibilities to combine or simplify the existing challenges in the hand. To summarize, a to-do list with four objectives can be listed as:

- Create a sequence of positive integer numbers from 2 to n (limit)
- Find an approach to detect non-prime numbers
- Find an approach to "remember" non-prime numbers
- Remove them from solution space to get the desired result

Since remembering is a binary option (being prime or not) it can be implemented as a variable and the creation of an integer number sequence that can be related to it. At this point, a person who is not familiar with data structures should search in-built data structures that exist in the programming language with desired properties or write his/her custom structure. However, most of the time "integer sequence" can be related to an "array" and binary with "Boolean". So, using a Boolean array allows combining two objectives that will mostly increase the coding efficiency. Indexes of the Boolean array are used as numbers and "remembered" by changing their value between "True" and "False". Since the default values of the Java Boolean variable is False, the indexes that have True value will be treated as a non-prime number.

This approach is based on the algorithm of "Sieve of Eratosthenes" and a Java implementation for it is given below. It is faster than traditional algorithms that are not using mathematical properties to the utmost limits, however, information about more efficient algorithms and prime number generation can be found in (Atkin & Bernstein, 2003).

```
public static ArrayList<Integer> sieveOfEratosthenes(int sequence_limit) {
    ArrayList<Integer> primes = new ArrayList<>();
    boolean[] integers = new boolean[sequence_limit];
    int prime_limit = (int) (Math.sqrt(sequence_limit) + 1);
    for (int index = 2; index < prime_limit; index++) {
        if (integers[index] == false) {
            //np_index => non prime index
            for (int np_index = index * 2; np_index < sequence_limit; np_index = np_index + 2) {
                integers[np_index] = true;
            }
        }
    }
    for (int index = 2; index < sequence_limit; index++) {
        if (integers[index] == false) {
            primes.add(index);
        }
    }
    return primes;
}
```

Figure 1.12. Implementation of the "Sieve of Eratosthenes" in Java

Most of the efficiency gain acquired by simply reducing the iteration of inner loop in with the first adjustment. However, the sieving technique shows increased efficiency compared to the first version with the adjustments as the range of numbers increase. The details for run time difference with related implementations can be found in Table A3.3 in Appendix 3.

1.4.3. Tower of Hanoi

A programmer must have comprehensive knowledge about the problem and be able to consider it from various aspects, to be able to produce the desired program. That is why algorithmic thinking is one of the essentials in programming. One way to improve/develop this ability is solving puzzles, thus, "Tower of Hanoi" with n disk and three peg variation is chosen to present different implementation of algorithms and their efficiency using Java.

The first implementation is a very straight forward recursive algorithm with negative logic. The approach built on the idea of imitating peg movements instead of disks. Each time disk moves from the "source" peg to the "destination" peg, instead of moving the disk itself, the program changes the representation of the pegs. So, the pegs named "source", "helper" and "destination" from left to right, and as the disk moves,

the “source” peg may become “destination”, “helper” peg may become “source” etc. The rotation of this representative change is decided based on the analysis of disk movements on various numbers of disks with several movements

```
public static int move = 1;
public static void recursiveHanoi(String source, String helper, String destination, int disk_count) {
    if (disk_count == 1) {
        System.out.println("Move " + move + ": disk in " + source + " to " + destination);
        move++;
    } else {
        recursiveHanoi(source, destination, helper, disk_count - 1);
        System.out.println("Move " + move + ": disk in " + source + " to " + destination);
        move++;
        recursiveHanoi(helper, source, destination, disk_count - 1);
    }
}
```

Figure 1.13. Implementation of “Tower of Hanoi” as a recursive function

The string parameters may be named as “Left”, “Middle”, “Right” for a more readable output or some integer sequence for analytic purposes.

This implementation of the problem is not efficient enough to solve the puzzle for higher values of the disk due to the storage limits of the program space. The limit of this usage can be increased by adjusting necessary settings or changing the implementation to use disk rather than RAM, however, analysis on problem indicates that changing the way of implementation should provide results for a higher number of disks without adjusting storage capacity.

In addition to problems in storage, using a recursive implementation on the problem creates a chain of steps, that will go down to the base case and return the solution after calculating all steps. This type of approach will probably increase the general complexity of the program which is inversely related to efficiency.

The second implementation uses an iterative function with an interesting approach. Representing disk movements in binary form, with respect to rules that are mentioned by Sedgewick in his book “Algorithms in Java, Parts 1-4/3E”.This representation uses 0,1,2 numbers for the pegs from left to right and the source number of m^{th} movement for the odd number of disks can be calculated by the formula $(m\&(m-1))\%3$ where m represent the number of movement starting from 1. Moreover, the analysis of source-destination number pairs indicates that the summation of source and destination numbers is repeating in the sequence of 1,2,3 for even number of disks and 2,1,3 for

odd number of disks. So, the destination number can be calculated by different mathematical or logical implementations, which will be explained in the following implementations.

```
1  public static void iterativeHanoiV1(int disk_count) {
2      int move_count = (1 << disk_count) - 1; // 2 to the power disk_count, minus 1
3      String[] words = {"Left", "Middle", "Right"};
4      int source, destination;
5      int even_odd = disk_count % 2;
6      for (int move = 1; move <= move_count; move++) {
7          source = (move & (move - 1)) % 3;
8          destination = 3 - (move % 3) - source;
9          source = ((1 - even_odd) * (3 - source) + even_odd * source) % 3;
10         destination = ((1 - even_odd) * (3 - destination) + even_odd * destination) % 3;
11         //System.out.println("Move " + move + ": disk in " + words[source] + " to " + words[destination]);
12     }
13 }
```

Figure 1.14. Implementation of "Tower of Hanoi" as iterative function Version 1

This implementation uses a mathematical calculation to assign values in lines 9 and 10, based on the condition of disk counts. To represent a value with a possible output of "A", when the condition is true or "B" if the condition is false. A general formula can be written as $(1 - c) * A + c * B$, where c is the condition with value 1 or 0 that represents true or false.

This mathematical representation in implementation can be replaced with direct access to conditional values by introducing a relation between disk count and integer sequence. The third version of the implementation can be written as:

```
1  public static void iterativeHanoiV2(int disk_count) {
2      int move_count = (1 << disk_count) - 1; // 2 to the power disk_count, minus 1
3      String[] words = {"Left", "Middle", "Right"};
4      int source, destination, source_value;
5      int even_odd = disk_count % 2;
6      int[][] number_mapping_source = {{0, 2, 1}, {1, 2, 3}};
7      int[][] number_mapping_destination = {{3, 1, 2}, {3, 2, 1}};
8      for (int move = 1; move <= move_count; move++) {
9          source_value = (move & (move - 1)) % 3;
10         source = number_mapping_source[even_odd][source_value];
11         destination = number_mapping_destination[even_odd][move % 3] - source;
12         System.out.println("Move " + move + ": disk in " + words[source] + " to " + words[destination]);
13     }
14 }
```

Figure 1.15. Implementation of "Tower of Hanoi" as iterative function Version 2

The second version of iterative implementation increased runtime of the program abnormally, (details in table A3.4, Appendix 3) mostly because of the clash between compiler optimization and replacement of mathematical computation with a multi-

dimensional array. There is also a possibility of the locality of reference for multi-dimensional array showed a poor performance because of the configuration of the test environment, however, a scenario that can distinguish this theory could not be produced.

The techniques that are mentioned up to this point are already known and some of them even were implemented at the compiler level. But, because of the wide-ranging decisions that can be taken to create a solution, some of these techniques are in the form of “advice”.

The invention of these techniques and different perspectives about the efficiency are presented in the following chapter.

CHAPTER 2
LITERATURE REVIEW

The techniques invented to increase the program efficiency can be thought of as a branch of a tree, which is related to the trunk of a tree, with other branches and roots, where the tree itself is the efficiency. Therefore, this chapter presents a variety of information related to efficiency that utilizes the invention of techniques mentioned above.

Efficiency itself has a shifting structure, definitions, requirements and the need for it changes, as time, and technology advances. In the early 1960s, people were desperate about the lack of hardware that is capable of achieving the desired program with other problems such as the cost of implementation or time to debug, test, etc. So, the efficiency was a must, which generally leads the program under intractable changes that only the programmer himself was able to understand. This is also one of the fundamental reasons behind the trade between "correctness and efficiency". As people start to become aware of the ongoing problem a short chapter was written (Dijkstra, 1965) about the quality and correctness of a program to increase the attention. Another work in this area written by Weissman (1974), that focuses on programmers and effects on the program with the reasons behind the complexity of it. A list of factors defined in the paper contains some of the rules that are required to write quality software, independent from the programming language. A comparison between the "psychological complexity" and cyclomatic complexity is presented by Tanik (1980). He used McCabe's cyclomatic measure on organized data gathered from different FORTRAN and COBOL programs to present statistical data as well as the comparison. His results indicated that the factors used in cyclomatic and psychological complexity models are not identical, with the encouragement of the repetition of the experiment due to the limited number of programs.

In addition to methodologies written for programmers, there are also ways of writing a program in an efficient way. The conference paper written by Darlington and Buxatall (1973), is about a program that converts written program to another program

using the idea of abstract programming. As the programmer writes a code according to rules, the proposed program can replace his/her code with more efficient versions mainly by removing higher function calls.

As the concept becomes more fundamental, its effect on efficiency gets increased considerably and algorithm design and analysis stay on the top. An article was written to decide which design methodology is the "right" one by Parker (1978). His study compares some of the selected design methodologies and compares them by some criteria such as complexity, modularity, etc. His conclusion states that the "Jackson methodology must be the recommended technique", however, the writer also states the importance of module independence in the conclusion. In addition to the theoretical part of algorithm design, application of design changes and their effect on program complexity and run-time was presented in (Bentley, 1979), that also includes a brief explanation of selected problems and other topics related to algorithm design such as algorithm classification, complexity and data structures. One of the most important works related to algorithm design and analysis is written by Levitin (2000), in which, he classified problem-solving in static and dynamic categories. Especially, the dynamic view of algorithmic problem solving was modeled by a diagram in his work, shaped in an iterative process which was also mentioned by the writer himself. The static view contains relatively subjects that should be considered to produce a solution. On the other hand, the dynamic view includes the general structure of problem-solving step by step. The article mainly criticizes the current treatment at that time to point out the problems and propose an alternative way for improvements.

The treatment of efficiency moves towards the concept of "on-demand". Programmers tend to deal with efficiency when they realize it is needed rather than thinking it as another important part of programming. An article was written about performance issues on programming and was thought in their undergraduate course(Debray, 2004), that especially indicates, treating the performance-related topics in the academic courses piece by piece, and let the students make mistakes about the performance-related issues as much as understanding them. Nasehi et al. (2012), made analytical research to find effective code examples and identify the characteristics of them. Their work was especially limited with a single programming language with some other criteria, however, their classification and finding can be relatable with the programmer's limitation as well as program efficiency.

The following works constitute the backbone of this thesis as they specially mention the techniques to improve the program efficiency. The 2nd edition of the book written by Kernighan and Plaguer (1978), has an entire chapter devoted to the efficiency of a program, and the overall of the book contains different rules to improve the software quality. *Writing Efficient Programs* (Bentley, 1982) categorize the design levels in 6 different categories to treat each of them with different techniques that will improve efficiency. The general structure of the book includes sample problems and the technique applied with run time comparisons to illustrate the difference over changes that are also presented in this thesis. Moreover, the name of the techniques used for the categorization and elaboration in the next chapter of this thesis were taken from the book. Another work of the same writer (Bentley, 1986) is a book of the collection of his works throughout the ’80s. Each column was presented in a formed structure that started with a sample problem and the explanation of the intended subject throughout the column with the closure of problems to solve for the reader and the suggestions for other related reading materials. The 2nd edition of the same book (Bentley, 2000) presents very interesting results in the term of efficiency. After updating some of his codes to adapt to the current language and other specifications at that time, he indicates that “the times were almost unchanged across fifteen years.” (Bentley,2000, p.281). Even though he took the runtime measurements under the highest level of compiler optimization, the run times show, his techniques were not out of date.

CHAPTER 3
EFFICIENCY TECHNIQUES

The techniques that will be mentioned in this chapter are taken from (Bentley, 1982, p.110) since they are organized and well explained over various problems. The same writer also published another book (Bentley, 2000) which states the efficiency techniques were still effective based on the run time compared with the prior edition (Bentley, 1986). Table 3.1 represents the runtime results of different algorithms written to solve a problem on a vector with the size of n that includes positive and negative integer values. Each algorithm uses a different approach to find the maximum sum in any contiguous sub-vector. These results used as a way of benchmarking to measure the effectiveness of efficiency techniques as well as their relationship with compiler optimizations.

Year		Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
1986	First Edition	$3.4n^3$	$13n^2$	$46n\ log_2\ n$	$33n$
2000	Second Edition	$1.3n^3$	$10n^2$	$47n\ log_2\ n$	$48n$
2020	Thesis Results	$0.22n^3$	$1.14n^2$	$43.74n\ log_2\ n$	$3.81n$

Table 3.1. Run time results of algorithms with the addition of thesis results. Thesis run times results are extended to nanoseconds in order to display coefficients with less fraction.

Even though coefficients are getting smaller, the application of techniques still proves their effectiveness, since the algorithms stated in the table are using almost the same configuration over the years and the implementation of the techniques did not clash with compiler optimizations.

3.1. Categorization of Techniques

This thesis is specifically written with the aim of expending a selected efficiency technique to make it more structural and, relating the technique with a potential decision criterion for the ease of use. However, over the last 30 years, the applicational usage of some techniques has changed, therefore, a pre-categorization of the techniques is necessary to find a suitable technique for working with. The “WBG” (acronym for White, Black, Gray) categorization uses three distinct categories to separate existing techniques based on characteristic criteria that every member in the category must-have.

3.1.1. White

“White” is a synonym used for the techniques that are implemented at the compiler level for applicational uses. The selected techniques are collected from the C and Java compiler information. The detailed optimization parameters with their explanation for the C compiler can be found in (GCC Team, 2020). The Java compiler information is collected by running the java command, the detailed information can be found in the appendix section.

3.1.2. Black

“Black” is a synonym used for the techniques that the implementation of the technique on the problem is affected by various decision criteria and the changes of it vary on large scale.

3.1.3. Gray

“Gray” is a synonym used for the techniques that neither belong to the “Black” or “White” category. In other words, the techniques in this category are possible candidates for other categories as the related works continue. The categorization of these techniques is represented below:

*Data structure augmentation		
*Coroutines		
*Store precomputed results		
*Exploit common cases		
*Exploit algebraic identities		
*Precompute logical functions		
*Compiler time initialization		
*Code motion out of loops	*Caching	*Packing
*Combining tests	*Interpreters	
*Loop unrolling	*Exploit word parallelism	
*Transfer-driven loop unrolling	*Parallelism	
*Unconditional branch removal	*Short-circuiting monotone functions	
*Loop fusion	*Reordering tests	
*Boolean variable elimination	*Collapsing procedure hierarchies	
*Common subexpression elimination	*Transformation of recursive procedures	
*Pairing computation	*Lazy evaluation	

Table 3.2. WBG Categorization of Rules

The majority of the techniques categorized under the white category have an incomplete implementation since the techniques in this thesis prove their effectiveness even under the compiler optimization. The optimization done by the compiler may decrease the run-time up to some factor however the characteristics of this "optimization" can vary due to the nature of the problem, programming language, platform etc.

The researches on compilers show that the underlying structure of compiler should be adjusted to increase the optimization performance or the accuracy of applied

optimizations (Li et al., 2019; Pape, Bolz & Hirschfeld, 2017). Therefore, exploiting the advantage of efficiency techniques can be more useful than that of compiler optimization for the programmer.

3.2. Enhancement of Techniques

Even though most of the mentioned techniques above have abstract book definitions, it is possible to derive structured methodologies from their underlying meaning. As an example, the book definition for “store pre-computed results” states the cost of recomputing an expansive repetitive function can be reduced significantly by storing the result of the function in a lookup table. This definition can also be used to define the implementation content for the optimal solution of sub-problems in “dynamic programming”. Another example is from the definition of “exploit common cases” which states the procedures should treat “common cases” (the case represents the behavior of the program with/to a data that has nonmutated properties) efficiently. In an object-oriented paradigm, if we treat the nonmutated data as an object, a design pattern called “singleton” used to return the same instance of the object for every call, that increases storage, and time efficiency, since the creation of a new instance costs more. A similar approach for the “data structure augmentation” technique is used to construct a more structured methodology in this section.

As it categorized in the “black” category, the implementation of the technique itself may vary, however, some of the problem characteristics can be helpful to make decisions for the implementation, due to the relationship between problem variables and the nature of selected data structure.

3.2.1. Abstraction

The abstraction is used for many improvements in the implementations for problems in this thesis. In the “Roman Numerals” problem, the first version used a two-dimensional array structure by relating the first dimension as the digit index of the value and the second dimension as the Roman Numeral equivalent for the digit itself. On the other hand, in the second adjustment of the problem, the programmer abstracts the digit index by replacing necessary calls to the augmented data structure, which allows further improvements, such as the loop elimination as its last adjustment.

"Prime Number" problem, is one of the perfect examples to demonstrate the mentioned relationship for problems that mainly uses integer sequence as a variable set. While the first implementation with adjustments focuses on mathematical division properties, the sieving method uses the index-based nature of array structure and stores the Boolean information of the number (being prime or not) in the structure itself. This augmentation allows the usage of mathematical properties to sieve numbers much more efficiently compared to the first version, and the second version as the number limit increase.

3.2.2. Relation Based Storage

For a decision problem that requires extensive comparison to relate two concepts, a relation function can be used to reduce the number of elimination or eliminate them completely. The creation of this type of relation function can be produced from the built-in properties of the system or using mathematical equations by exploiting the concepts. A common example is, most of the programming languages have a built-in function to return the ASCII value (an integer value corresponding to each letter that exists in the computer environment). By using these values, a programmer can create a function to sort words in alphabetical order (since each letter has a unique integer value, the overall comparison of the letters from left to right will determine the position of the word in a list type structure). The second example is the implementation of a well-known game "Rock, Paper, Scissor" in Java. The first implementation uses String word input (under the assumption of user input is always valid) comparison to display the result as "Tie"," Player 1 win" or "Player 2 win". The first implementation uses String-based comparison and multiple decision statements based on the pre-determined rules of movements and wins conditions to return a human-readable result, whereas, using human-readable values may make the program harder to comprehend or create a relationship between variables, or vice versa.

```
public static String winner(String player_1, String player_2) {
    String result;
    if (player_1.equals(player_2)) {
        result = "Tie";
    } else {
        if (player_1.equals("Rock")) {
            if (player_2.equals("Scissor")) {
                result = "Player 1 win";
            } else {
                result = "Player 2 win";
            }
        } else if (player_1.equals("Paper")) {
            if (player_2.equals("Rock")) {
                result = "Player 1 win";
            } else {
                result = "Player 2 win";
            }
        } else { //Player 1 "Scissor"
            if (player_2.equals("Paper")) {
                result = "Player 1 win";
            } else {
                result = "Player 2 win";
            }
        }
    }
    return result;
}
```

Figure 3.1. Implementation of RPS game

However, using the abstraction, we can use “1”,”2”,”3” integer values instead of “Rock”, “Paper” and “Scissor” words. The representative change for values allows the programmer to represent values on primitive type array structure, rather than using multiple branches of comparison by storing the win conditions that simplifies reduces total comparison to three. The second implementation is written as:

```
public static String winner1(int player_1, int player_2) {
    // Relationship Structure, representation of movements with integer numbers
    // Rock->0 , Paper->1 , Scissor->2
    String result;
    int[] win_conditions = {2, 0, 1};
    if (player_1 == player_2) {
        result = "Tie";
    } else {
        // variable = (condition) ? ( value if condition true) : (value if condition false)
        result = (win_conditions[player_1] == player_2) ? "Player 1 win" : "Player 2 win";
    }
    return result;
}
```

Figure 3.2. Relation-based RPS game

Even though this example is too simple to measure time or space complexity changes, the underlying idea of relation-based storage is heavily used in hash-tables as a key generation function, using a relation function to store values in the table, the information about the existence of a value in a table can be obtainable in constant time rather than searching extensively. (Collusion handling is not accounted for this example).

The methodologies mentioned in this section are not a problem specific examples, instead, programmers may apply them instinctively without the consideration of them as a technique to various problems. Mainly because of “on-demand” treatment to these techniques, the lack of a structured definition even caused some people to refer to them as “tricks”. On the contrary, to this day, their influence on program efficiency is too firm to be simplified as a “trick” or remaining as “advice”. The possible ways to enhance the techniques, as well as the improvement of their definition, are remaining as a future work for researches intended to work on these subjects.

CHAPTER 4
CONCLUSION AND FUTURE WORK

Throughout the thesis, a handful of problems and solutions are provided, to compare their effectiveness in the terms of mainly time, and space complexity. Various techniques are used for improvements and their effects are unignorable based on the collected results. For the task of converting integer numbers to Roman numerals, each improvement speeds up the program approximately by the factor of 10 (from 160 μs/op to 0.014 μs/op) mainly by merging the job of two-loop into the single one (Loop fusion) and changing the structure of array for convenient access to numeral values (Data structure augmentation). Finding prime numbers up to n includes a speed up from the factor of 6 up to 300 and more (for values of n large enough to calculate). Even though most of the speedup comes from the mathematical exploit to decrease the range of testing numbers, the idea of using Boolean array as a "remember" option still affects 25% speedup for the program (Exploit common cases, Data structure augmentation). A different implementation of the Hanoi Tower problem has extraordinary run time differences, which clearly show the perception of time for us and efficiency needs should not be the key factor for the decision making. With the run time difference from 3190178.365 μs/op to 0.001 μs/op (nanosecond result values are not included due to the precision loss) the trade-off between time and space, and vice versa, as well as the implementation of techniques, may not always yield desired results. The precision and accuracy of results in this thesis are carried with the utmost caution, however, the runtime result of Algorithm3 ($43.74\, n\, log_2\, n$) includes a coefficient value with loss of precision due to the unknown cause that could not be isolated during the experiment. For this reason, the experiment configuration and the code used to test this algorithm requires extra work to supply more precise results. Moreover, the idea of using selected algorithms over decades to test the effectiveness of the applied techniques can be standardized as a supportive idea for the works that may deal with similar subjects in the future.

The enhancement of the “data structure augmentation” technique, clarify the part of the relation between the structure and the problem variables that can be altered to improve performance by introducing “abstraction” and “relation-based storage” terms. These terms are referring to the application of some of the changes a programmer can perform as well as the characteristic properties of a problem that may lead the programmer to exploit the technique. Even though these terms increase the ease of use compared to the book definition of the technique, additional researches required to standardize the definition to take advantage of the technique completely.

Each technique listed in the WBG categorization is a potential subject to work on with various applications. A person may choose to work on a compiler level to increase the effectiveness of “White” techniques or theoretical work on techniques in other categories to shape them in a more structural way for ease of use. Even though these techniques are developed to make the program more efficient, they are generally applied by the programmer. Therefore, experimental work that includes the effect of these techniques to programmers as well as to program may yield better results.

REFERENCES

Atkin, A., & Bernstein, D. (2003). Prime sieves using binary quadratic forms. Mathematics Of Computation, 73(246), 1023-1031. doi: 10.1090/s0025-5718-03-01501-1

Bentley. (1979). Special Feature An Introduction to Algorithm Design. Computer, 12(2), 66-78. doi: 10.1109/mc.1979.1658620

Bentley, J. (1986). Programming Pearls (1st ed.). NewYork: Association for Computing Machinery.

Bentley, J. (2000). Programming Pearls (2nd ed.). NewYork: ACM Press/Addison-Wesley Publishing Co.

Bentley, J. (1982). Writing efficient programs. Englewood Cliffs, NJ: Prentice-Hall.

Darlington, J., & Buxatall, A. (1973). A system which automatically improves programs. In IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence (pp. 479–485). San Francisco: Morgan Kaufmann Publishers Inc.

Debray, S. (2004). Writing efficient programs. ACM SIGCSE Bulletin, 36(1), 275. doi: 10.1145/1028174.971396

Djikstra, E. (1979). Programming considered as a human activity. In E. Yourdon, Classics in software engineering (pp. 1-9). NewYork: Yourdon Press.

Java, C., & Junior, B. (2020). Converting Integers to Roman Numerals - Java. Retrieved 11 May 2020, from https://stackoverflow.com/questions/12967896/converting-integers-to-roman-numerals-java

Kernighan, B., & Plauger, P. (1978). The elements of programming style (2nd ed.). New York: McGraw-Hill.

Levitin, A. (2000). Design and analysis of algorithms reconsidered. ACM SIGCSE Bulletin, 32(1), 16-20. doi: 10.1145/331795.331802

Li, X., Gao, G., Zhang, J., Jiang, H., & Ren, Z. (2019). Selection of compiler-optimization sequences. *SCIENTIA SINICA Informationis*, *49*(10), 1267-1282. doi: 10.1360/n112019-00050

McGeoch, C. (2012). A guide to experimental algorithmics (pp. 50-83). Cambridge: Cambridge University Press.

Nasehi, S., Sillito, J., Maurer, F., & Burns, C. (2012). What makes a good code example?: A study of programming Q&A in StackOverflow. 2012 28Th IEEE

International Conference On Software Maintenance (ICSM). doi: 10.1109/icsm.2012.6405249

Pape, T., Bolz, C., & Hirschfeld, R. (2017). Adaptive just-in-time value class optimization for lowering memory consumption and improving execution time performance. *Science Of Computer Programming*, *140*, 17-29. doi: 10.1016/j.scico.2016.08.003

Parker, J. (1978). A comparison of design methodologies. ACM SIGSOFT Software Engineering Notes, 3(4), 12-19. doi: 10.1145/1010741.1010743

Tanik, M. (1980). A comparison of program complexity prediction models. ACM SIGSOFT Software Engineering Notes, 5(4), 10-16. doi: 10.1145/1010884.1010888

Weissman, L. (1974). Psychological complexity of computer programs. ACM SIGPLAN Notices, 9(6), 25-36. doi: 10.1145/953233.953237

APPENDIX 1 – Experiment Environment

All the run time results are taken on a 64-bit Win10 operating system with minimal startup option. (8 GB RAM, Intel i5-8300H). Each version of the C programs runs a hundred-times in a loop (each of the individual) and their results are stored in a file. This entire process is done by running a Windows bash script that can be found in the next appendix. As the C compiler, MinGW (MinGW.org GCC-8.2.0-5) was used to compile the C programs. To be able to get accurate and precise time results "Query Performance Counter" was used with a microsecond time unit respect to measurement criteria as mentioned in (McGeoch, 2012).

Java program results were taken by using a Java benchmark tool named as "JMH" (Version 1.23) with Maven (Version 4.0.0) and JVM (JDK 13.0.2 Java HotSpot(TM) 64-Bit Server VM, 13.0.2+8). For the benchmark configuration, default parameters are not changed since the main purpose of the Java benchmark experiment was to measure the run time difference between the updated version of the same code, rather than optimizing the code itself. While using a benchmark, any printing function such as "System.out.println()" etc. are comment out from the code to prevent interference with the benchmark output as well as the time measurement.

APPENDIX 2 – CODE

1. Windows script file (timing.bat) with parameter explanations

%1 => filename for C file
%2 => filename (unoptimized / optimized)
%3 => inputSize
%4 => optimization param (-O3)

```
ECHO OFF
if not exist "some_path\Thesis\Experiments\%1" (mkdir " some_path
\Thesis\Experiments\%1")
if not exist " some_path \Thesis\Experiments\%1\%2" (mkdir " some_path
\Thesis\Experiments\%1\%2")
gcc %4 some_path \%1.c -o some_path \Thesis\ThesisCodes\%1.exe
FOR /L %%A IN (1,1,100) DO (
ECHO %%A
some_path \Thesis\ThesisCodes\%1.exe %3 >> some_path
\Thesis\Experiments\%1\%2\%3.txt
)
PAUSE
```

2. C program to print an integer number as a bit string (intToBinary.c)

```
// Written by Atabarış Ayaydın
#include <stdio.h>
#include <math.h>


void intToBinary(int number){
	int bit_count = log2(number)+1;
	int bits[bit_count];
	int index = bit_count-1; //last index
	while(number > 0){
		bits[index]=number %2;
		number =  number /2;
		index = index-1;
	}
	for(index=0;index<bit_count;index++){
		printf("%d",bits[index]);
	}
}


void main(){
	int number = 65;
	intToBinary(number);

}
```

3. C program for Algorithm 1 (maximumSumCubic.c)

```
// Copyright (C) 1999 Lucent Technologies
// Originally taken from Bentley's "Prograamming Pearls 2nd
Edition", Pearls 8-3
// Modified by Atabarış Ayaydın
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>


int main(int argc, char **argv) {
    srand(time(NULL));
    int n = atoi(argv[1]);
    float x[n];
    int i, j, k;
    float sum, maxsofar = 0;

    /* Fill x[n] with reals uniform on [-1,1] */
    for (i = 0; i < n; i++) {
        x[i] = 1 - 2 * ((float) rand() / RAND_MAX);
    }
    LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
    LARGE_INTEGER Frequency;

    QueryPerformanceFrequency(&Frequency);
    QueryPerformanceCounter(&StartingTime);

    for (i = 0; i < n; i++){
        for (j = i; j < n; j++) {
            sum = 0;
            for (k = i; k <= j; k++){
                  sum += x[k];
                  }
            if (sum > maxsofar){
                  maxsofar = sum;
                  }
        }
      }

    QueryPerformanceCounter(&EndingTime);
    ElapsedMicroseconds.QuadPart = EndingTime.QuadPart -
StartingTime.QuadPart;
    ElapsedMicroseconds.QuadPart *= 1000000;
    ElapsedMicroseconds.QuadPart /= Frequency.QuadPart;
    printf("MaxSum : %f ",maxsofar);
      printf("%lli \n", ElapsedMicroseconds.QuadPart);
    return ElapsedMicroseconds.QuadPart;
}
```

4. C program for Algorithm 2 (maximumSumQuadratic.c)

```
// Copyright (C) 1999 Lucent Technologies
// Originally taken from Bentley's "Prograamming Pearls 2nd
Edition", Pearls 8-4
// Modified by Atabarış Ayaydın
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>


int main(int argc, char **argv) {
    srand(time(NULL));
    int n = atoi(argv[1]);
    float x[n];
    int i, j, k;
    float sum, maxsofar = 0;

    /* Fill x[n] with reals uniform on [-1,1] */
    for (i = 0; i < n; i++) {
        x[i] = 1 - 2 * ((float) rand() / RAND_MAX);
    }

    LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
    LARGE_INTEGER Frequency;


    QueryPerformanceFrequency(&Frequency);
    QueryPerformanceCounter(&StartingTime);

    for (i = 0; i < n; i++) {
        sum = 0;
        for (j = i; j < n; j++) {
            sum += x[j];
            maxsofar = max(maxsofar,sum);
        }
    }

    QueryPerformanceCounter(&EndingTime);
    ElapsedMicroseconds.QuadPart = EndingTime.QuadPart -
StartingTime.QuadPart;
    ElapsedMicroseconds.QuadPart *= 1000000;
    ElapsedMicroseconds.QuadPart /= Frequency.QuadPart;
    printf("MaxSum : %f ",maxsofar);
      printf("%lli \n", ElapsedMicroseconds.QuadPart);
    return ElapsedMicroseconds.QuadPart;
}
```

5. C program for Algorithm 3 (maximumSumLinearithmic.c)

```
// Copyright (C) 1999 Lucent Technologies
// Originally taken from Bentley's "Prograamming Pearls 2nd
Edition", Pearls 8-6
// Modified by Atabarış Ayaydın
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>
float *x;

float maxsum3(int l, int u) {
    int i, m;
    float lmax, rmax, sum;

    if (l > u) {/* zero elements */
        return 0;
    }
    if (l == u) {/* one element */
        return max(0, x[l]);
    }
    m = (l + u) / 2;
    /* find max crossing to left */
    lmax = sum = 0;
    for (i = m; i >= l; i--) {
        sum += x[i];
        lmax = max(lmax,sum);
    }
    /* find max crossing to right */
    rmax = sum = 0;
    for (i = m + 1; i <= u; i++) {
        sum += x[i];
        rmax = max(rmax,sum);
    }
    return max(lmax + rmax, max(maxsum3(l, m), maxsum3(m + 1, u)));
}

float maxsum(int n){
      return maxsum3(0, n - 1);
}

int main(int argc, char **argv) {
    srand(time(NULL));
    int n = atoi(argv[1]);
      x = malloc(n * sizeof(float));
    int i, j, k;
    float sum, maxsofar = 0;
    /* Fill x[n] with reals uniform on [-1,1] */
    for (i = 0; i < n; i++) {
        x[i] = 1 - 2 * ((float) rand() / RAND_MAX);
    }

    LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
    LARGE_INTEGER Frequency;
```

```
    QueryPerformanceFrequency(&Frequency);
    QueryPerformanceCounter(&StartingTime);
    maxsofar = maxsum(n);

    QueryPerformanceCounter(&EndingTime);
    ElapsedMicroseconds.QuadPart = EndingTime.QuadPart -
StartingTime.QuadPart;
    ElapsedMicroseconds.QuadPart *= 1000000;
    ElapsedMicroseconds.QuadPart /= Frequency.QuadPart;
    printf("MaxSum : %f ",maxsofar);
    printf("%lli \n", ElapsedMicroseconds.QuadPart);
    free(x);
    return ElapsedMicroseconds.QuadPart;
}
```

6. C program for Algorithm 4 (maximumSumLinear.c)

```
// Copyright (C) 1999 Lucent Technologies
// Originally taken from Bentley's "Prograamming Pearls 2nd
Edition", Pearls 8-4
// Modified by Atabarış Ayaydın
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>


int main(int argc, char **argv) {
    srand(time(NULL));
    int n = atoi(argv[1]);
    float x[n];
    int i, j, k;
    float sum, maxsofar = 0;

    /* Fill x[n] with reals uniform on [-1,1] */
    for (i = 0; i < n; i++) {
        x[i] = 1 - 2 * ((float) rand() / RAND_MAX);
    }

    LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
    LARGE_INTEGER Frequency;


    QueryPerformanceFrequency(&Frequency);
    QueryPerformanceCounter(&StartingTime);

    for (i = 0; i < n; i++) {
        sum = 0;
        for (j = i; j < n; j++) {
            sum += x[j];
            maxsofar = max(maxsofar,sum);
        }
    }

    QueryPerformanceCounter(&EndingTime);
    ElapsedMicroseconds.QuadPart = EndingTime.QuadPart -
StartingTime.QuadPart;
    ElapsedMicroseconds.QuadPart *= 1000000;
    ElapsedMicroseconds.QuadPart /= Frequency.QuadPart;
```

```
    printf("MaxSum : %f ",maxsofar);
      printf("%lli \n", ElapsedMicroseconds.QuadPart);
    return ElapsedMicroseconds.QuadPart;
}
```

7. Maven pom.xml file for JMH Benchmark

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>JMHTest</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <jmh.version>1.23</jmh.version>
    <maven.compiler.source>13</maven.compiler.source>
    <maven.compiler.target>13</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-core</artifactId>
      <version>${jmh.version}</version>
    </dependency>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-generator-annprocess</artifactId>
      <version>${jmh.version}</version>
    </dependency>
  </dependencies>

</project>
```

8. Java Benchmark runnable class

```
package classes;

import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.util.concurrent.TimeUnit;

public class Main {
```

```
    public static int move = 1;

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(Main.class.getSimpleName())
                .forks(1)
                .build();

        new Runner(opt).run();
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.MICROSECONDS)
    public void runnableMethod() {
      //some parameters
      functionToMeasure(parameters);
    }

    public static functionToMeasure(parameters){
      //....
    }

}
```

9. Java program that prints integer number as a bit string (intToBinary.java)

```
// Written by Atabarış Ayaydın

import java.util.ArrayList;

public class IntBinary {

    public static String intToBinaryV1(int number) {
        ArrayList<Integer> bits = new ArrayList<>();
        String binaryString = "";
        while (number > 0) {
            bits.add(number % 2);
            number = number / 2;
        }
        for (int index = bits.size() - 1; index >= 0; index--) {
            binaryString = binaryString + bits.get(index).toString();
        }
        return binaryString;
    }

    public static String intToBinaryV2(int number) {
        String binaryString = "";
        while (number > 0) {
            binaryString = Integer.toString(number % 2) + binaryString;
            number = number / 2;
        }
        return binaryString;
    }
}
```

10. Java program to solve 3 peg Hanoi Tower for n disk (Hanoi.java)

```
//Written by Atabarış Ayaydın

public class Hanoi {
   public static int move = 1;

   public static void recursiveHanoi(String source, String helper, String destination, int
disk_count) {
      if (disk_count == 1) {
         System.out.println("Move " + move + ": disk in " + source + " to " + destination);
         move++;
      } else {
         recursiveHanoi(source, destination, helper, disk_count - 1);
         System.out.println("Move " + move + ": disk in " + source + " to " + destination);
         move++;
         recursiveHanoi(helper, source, destination, disk_count - 1);
      }
   }

   public static void iterativeHanoiV1(int disk_count) {
      int move_count = (1 << disk_count) - 1; // 2 to the power disk_count, minus 1
      String[] words = {"Left", "Middle", "Right"};
      int source, destination;
      int even_odd = disk_count % 2;
      for (int move = 1; move <= move_count; move++) {
         source = (move & (move - 1)) % 3;
         destination = 3 - (move % 3) - source;
         source = ((1 - even_odd) * (3 - source) + even_odd * source) % 3;
         destination = ((1 - even_odd) * (3 - destination) + even_odd * destination) % 3;
         System.out.println("Move " + move + ": disk in " + words[source] + " to " +
words[destination]);
      }
   }

   public static void iterativeHanoiV2(int disk_count) {
      int move_count = (1 << disk_count) - 1; // 2 to the power disk_count, minus 1
      String[] words = {"Left", "Middle", "Right"};
      int source, destination, source_value;
      int even_odd = disk_count % 2;
      int[][] number_mapping_source = {{0, 2, 1}, {1, 2, 3}};
      int[][] number_mapping_destination = {{3, 1, 2}, {3, 2, 1}};
      for (int move = 1; move <= move_count; move++) {
         source_value = (move & (move - 1)) % 3;
         source = number_mapping_source[even_odd][source_value];
         destination = number_mapping_destination[even_odd][move % 3] - source;
         System.out.println("Move " + move + ": disk in " + words[source] + " to " +
words[destination]);
      }
   }

}
```

11. Java program to convert print integer number as a Roman numeral (Roman.java)

```
//Written by Atabarış Ayaydın

import java.util.ArrayList;

public class Roman {
  public static void main(String[] args) {
    System.out.println(convertToRomanV1(155));
  }

  public static String convertToRomanV1(int number) {
    int i = 0;
    ArrayList<String> digits = new ArrayList<>();
    String[][] romans =
        {
            {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"},
            {"", "X", "XX", "XXX", "XL", "L", "LX", "LXXX", "XC"},
            {"", "C", "CC", "CCC", "CD", "D", "D", "DC", "DCC", "DCCC", "CM"}
        };
    String result = "";
    while (number > 0) {
      digits.add(romans[i][(number % 10)]);
      number = (number - (number % 10)) / 10;
      i++;
    }
    for (int j = digits.size() - 1; j >= 0; j--) {
      result = result + digits.get(j);
    }
    return result;
  }

  public static String convertToRomanV2(int number) {
    int i = 0;
    String[][] romans =
        {
            {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"},
            {"", "X", "XX", "XXX", "XL", "L", "LX", "LXXX", "XC"},
            {"", "C", "CC", "CCC", "CD", "D", "D", "DC", "DCC", "DCCC", "CM"}
        };
    String result = "";
    while (number > 0) {
      result = romans[i][(number % 10)] + result;
      number = (number - (number % 10)) / 10;
      i++;
    }
    return result;
  }

  public static String convertToRomanV3(int number) {
    String[] units = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"};
    String[] tens = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXXX", "XC"};
    String[] hundreds = {"", "C", "CC", "CCC", "CD", "D", "D", "DC", "DCC", "DCCC", "CM"};
```

```
    return hundreds[(number % 1000) / 100] + tens[(number % 100) / 10] + units[(number %
10)];
  }
}
```

12. Java program to find all primes up to n and the implementation of the Sieve of Eratosthenes (Prime.java)

```
//Written by Atabarış Ayaydın

import java.util.ArrayList;

public class Prime {
  public static void main(String[] args) {
    int prime_limit = 1000;
  }

  public static ArrayList<Integer> primesV1(int limit) {
    ArrayList<Integer> prime_list = new ArrayList<>();
    boolean isPrime;
    for (int number = 2; number < limit; number++) {
      isPrime = true;
      for (int divider = 2; divider < number; divider++) {
        if (number % divider == 0) {
          isPrime = false;
          break;
        }
      }
      if (isPrime) {
        prime_list.add(number);
      }
    }
    return prime_list;
  }

  public static ArrayList<Integer> primesV2(int limit) {
    ArrayList<Integer> prime_list = new ArrayList<>();
    boolean isPrime;
    prime_list.add(2);
    for (int number = 3; number < limit; number = number + 2) {
      isPrime = true;
      int divider_limit = (int) (Math.sqrt(number) + 1);
      for (int divider = 3; divider < divider_limit; divider = divider + 2) {
        if (number % divider == 0) {
          isPrime = false;
          break;
        }
      }
      if (isPrime) {
        prime_list.add(number);
      }
    }
    return prime_list;
  }

  public static ArrayList<Integer> sieveOfEratosthenes(int sequence_limit) {
    ArrayList<Integer> primes = new ArrayList<>();
    boolean[] integers = new boolean[sequence_limit];
```

```
        int prime_limit = (int) (Math.sqrt(sequence_limit) + 1);
        for (int index = 2; index < prime_limit; index++) {
            if (integers[index] == false) {
                //np_index => non prime index
                for (int np_index = index * 2; np_index < sequence_limit; np_index = np_index + 2) {
                    integers[np_index] = true;
                }
            }
        }
        for (int index = 2; index < sequence_limit; index++) {
            if (integers[index] == false) {
                primes.add(index);
            }
        }
        return primes;
    }
}
```

APPENDIX 3 – Runtime Tables

The following tables include run-time results of different implementations and/or variations of programs. The left upper cell includes the name of the program corresponding to the problem and the programming language used, in parenthesis. The other cells on the first row contain the test values used on the experiment and their runtime results are presented in the microsecond unit.

Integer to Binary (Java)	*Test Number (376376376)*
Version 1	*1.318 µs/op*
Version 2	*0.659 µs/op*

Table A3.1. Integer to Binary Results

Integer to Roman Numeral (Java)	*Test Number 376*
Version 1	*0.160 µs/op*
Version 2	*0.130 µs/op*
Version 3	*0.014 µs/op*

Table A3.2. Integer to Roman Numeral Results

Prime Numbers (Java)	*Size Limit 100*	*Size Limit 1000*	*Size Limit 10000*	*Size Limit 100000*
Version 1	*4.345 µs/op*	*297.299 µs/op*	*20822.777 µs/op*	*1700368.140 µs/op*
Version 2	*0.710 µs/op*	*11.072 µs/op*	*265.012 µs/op*	*5805.157 µs/op*
Sieve of Eratosthenes	*0.722 µs/op*	*10.050 µs/op*	*172.537 µs/op*	*4733.575 µs/op*

Table A3.3. Prime Number Results

Hanoi Tower (Java)	*Disk Count 7*	*Disk Count 15*	*Disk Count 30*
Iterative Version 1	*0.001 µs/op*	*0.001 µs/op*	*0.001 µs/op*
Iterative Version 2	*0.422 µs/op*	*97.516 µs/op*	*3190178.365 µs/op*
Recursive	*0.249 µs/op*	*65.231 µs/op*	*1082174.708 µs/op*

Table A3.4. Hanoi Tower Results

APPENDIX 4 – Runtime Table Benchmark Details

Table A3.1 – Version 1

1.318 ±(99.9%) 0.001 us/op [Average]

(min, avg, max) = (1.318, 1.318, 1.318), stdev = 0.001

CI (99.9%): [1.317, 1.318] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.intToBinaryV1 avgt 5 1.318 ± 0.001 us/op

Table A3.1 – Version 2

0.659 ±(99.9%) 0.008 us/op [Average]

(min, avg, max) = (0.656, 0.659, 0.661), stdev = 0.002

CI (99.9%): [0.651, 0.666] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.intToBinaryV2 avgt 5 0.659 ± 0.008 us/op

Table A3.2 – Version 1

0.160 ±(99.9%) 0.005 us/op [Average]

(min, avg, max) = (0.158, 0.160, 0.161), stdev = 0.001

CI (99.9%): [0.155, 0.165] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.convertToRomanV1 avgt 5 0.160 ± 0.005 us/op

Table A3.2 – Version 2

0.130 ±(99.9%) 0.001 us/op [Average]

(min, avg, max) = (0.130, 0.130, 0.130), stdev = 0.001

CI (99.9%): [0.130, 0.130] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.convertToRomanV2 avgt 5 0.130 ± 0.001 us/op

Table A3.2 – Version 3

0.014 ±(99.9%) 0.001 us/op [Average]

(min, avg, max) = (0.014, 0.014, 0.014), stdev = 0.001

CI (99.9%): [0.014, 0.014] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.convertToRomanV3 avgt 5 0.014 ± 0.001 us/op

Table A3.3 – Version 1 – Size Limit 100

4.345 ±(99.9%) 0.006 us/op [Average]

(min, avg, max) = (4.344, 4.345, 4.347), stdev = 0.001

CI (99.9%): [4.340, 4.351] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 4.345 ± 0.006 us/op

Table A3.3 – Version 1 – Size Limit 1000

297.299 ±(99.9%) 2.710 us/op [Average]

(min, avg, max) = (296.564, 297.299, 298.178), stdev = 0.704

CI (99.9%): [294.588, 300.009] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 297.299 ± 2.710 us/op

Table A3.3 – Version 1 – Size Limit 10000

20822.777 ±(99.9%) 23.969 us/op [Average]

(min, avg, max) = (20818.122, 20822.777, 20833.486), stdev = 6.225

CI (99.9%): [20798.808, 20846.746] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 20822.777 ± 23.969 us/op

Table A3.3 – Version 1 – Size Limit 100000

1700368.140 ±(99.9%) 2188.020 us/op [Average]

(min, avg, max) = (1699708.000, 1700368.140, 1701158.367), stdev = 568.222

CI (99.9%): [1698180.120, 1702556.160] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 1700368.140 ± 2188.020 us/op

Table A3.3 – Version 2 – Size Limit 100

0.710 ±(99.9%) 0.001 us/op [Average]

(min, avg, max) = (0.710, 0.710, 0.710), stdev = 0.001

CI (99.9%): [0.710, 0.711] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV2 avgt 5 0.710 ± 0.001 us/op

Table A3.3 – Version 1 – Size Limit 100

4.345 ±(99.9%) 0.006 us/op [Average]

(min, avg, max) = (4.344, 4.345, 4.347), stdev = 0.001

CI (99.9%): [4.340, 4.351] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 4.345 ± 0.006 us/op

Table A3.3 – Version 1 – Size Limit 1000

297.299 ±(99.9%) 2.710 us/op [Average]

(min, avg, max) = (296.564, 297.299, 298.178), stdev = 0.704

CI (99.9%): [294.588, 300.009] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 297.299 ± 2.710 us/op

Table A3.3 – Version 1 – Size Limit 10000

20822.777 ±(99.9%) 23.969 us/op [Average]

(min, avg, max) = (20818.122, 20822.777, 20833.486), stdev = 6.225

CI (99.9%): [20798.808, 20846.746] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 20822.777 ± 23.969 us/op

Table A3.3 – Version 1 – Size Limit 100000

1700368.140 ±(99.9%) 2188.020 us/op [Average]

(min, avg, max) = (1699708.000, 1700368.140, 1701158.367), stdev = 568.222

CI (99.9%): [1698180.120, 1702556.160] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV1 avgt 5 1700368.140 ± 2188.020 us/op

Table A3.3 – Version 2 – Size Limit 100

0.710 ±(99.9%) 0.001 us/op [Average]

(min, avg, max) = (0.710, 0.710, 0.710), stdev = 0.001

CI (99.9%): [0.710, 0.711] (assumes normal distribution)

Benchmark Mode Cnt Score Error Units

Main.primesV2 avgt 5 0.710 ± 0.001 us/op

Table A3.3 – Sieve of Eratosthenes – Size Limit 10000

172.537 ±(99.9%) 2.983 us/op [Average]

(min, avg, max) = (171.554, 172.537, 173.282), stdev = 0.775

CI (99.9%): [169.554, 175.520] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.sieveOfEratosthenes	avgt	5	172.537	± 2.983	us/op

Table A3.3 – Sieve of Eratosthenes – Size Limit 100000

4733.575 ±(99.9%) 8.497 us/op [Average]

(min, avg, max) = (4730.912, 4733.575, 4736.872), stdev = 2.207

CI (99.9%): [4725.077, 4742.072] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.sieveOfEratosthenes	avgt	5	4733.575	± 8.497	us/op

Table A3.4 – Iterative Version 1 – Disk Count 7,15,30

0.001 ±(99.9%) 0.001 us/op [Average]

(min, avg, max) = (0.001, 0.001, 0.001), stdev = 0.001

CI (99.9%): [0.001, 0.001] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.iterativeHanoiV1	avgt	5	0.001	± 0.001	us/op

Table A3.4 – Iterative Version 2 – Disk Count 7

0.422 ±(99.9%) 0.024 us/op [Average]

(min, avg, max) = (0.414, 0.422, 0.430), stdev = 0.006

CI (99.9%): [0.398, 0.447] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.iterativeHanoiV2	avgt	5	0.422	± 0.024	us/op

Table A3.4 – Iterative Version 2 – Disk Count 15

97.516 ±(99.9%) 0.514 us/op [Average]

(min, avg, max) = (97.426, 97.516, 97.742), stdev = 0.134

CI (99.9%): [97.001, 98.030] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.iterativeHanoiV2	avgt	5	97.516	± 0.514	us/op

Table A3.4 – Iterative Version 2 – Disk Count 30

3190178.365 ±(99.9%) 661.947 us/op [Average]

(min, avg, max) = (3189966.950, 3190178.365, 3190445.500), stdev = 171.905

CI (99.9%): [3189516.418, 3190840.312] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.iterativeHanoiV2	avgt	5	3190178.365	± 661.947	us/op

Table A3.4 – Recursive – Disk Count 7

0.249 ±(99.9%) 0.005 us/op [Average]

(min, avg, max) = (0.248, 0.249, 0.252), stdev = 0.001

CI (99.9%): [0.245, 0.254] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.recursiveHanoi	avgt	5	0.249	± 0.005	us/op

Table A3.4 – Recursive – Disk Count 15

65.231 ±(99.9%) 0.056 us/op [Average]

(min, avg, max) = (65.212, 65.231, 65.245), stdev = 0.015

CI (99.9%): [65.175, 65.287] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.recursiveHanoi	avgt	5	65.231	± 0.056	us/op

Table A3.4 – Recursive – Disk Count 30

1082174.708 ±(99.9%) 1253.577 us/op [Average]

(min, avg, max) = (1081778.370, 1082174.708, 1082519.280), stdev = 325.550

CI (99.9%): [1080921.131, 1083428.285] (assumes normal distribution)

Benchmark	Mode	Cnt	Score	Error	Units
Main.recursiveHanoi	avgt	5	1082174.708	± 1253.577	us/op