Japan Advanced Institute of Science and Technology

Information Science School

I116 – Fundamental Programing

Final Assignment Report

Student name: **NGUYEN, Tien Minh**

Student ID: **s1810445**

Student email: **minh.nguyen@jaist.ac.jp**

Instructor: **Prof. Armagan Elibol**

Submitted: **June, 2020**

Introduction

I use Jupiter notebook for writing code and presenting my idea for all final assignments

Some of advantages can be listed:

- **Suitable for presenting algorithm implementation**: write code (Python), documents (Markdown), show figures
- **Portable**: Run on browser, user only need a major browser to run and view a notebook
- **Not need to re-interpret source code for grading**

It's better to view the compiled notebook on Github, I have a Github repository that contains all my notebooks: https://github.com/m-inh/i116-final-assignment

Some answers of the teacher's requirements can be shown in notebook. The detailed answer is listed as below:

1. BASIC INFORMATION
This action is done in this report.

2. PROBLEM STATEMENT
This action is explained in all notebooks.

Every notebook has a short paragraph that describes the purpose of the program and contains teacher's requirements.

3. INPUT AND OUTPUT FORMAT
This action is explained in all notebooks.

Every Python function in my notebooks has comments that declare what the input and output are. In some important functions, I have more detailed information why I do that, and how the function works.

4. VARIABLE DESCRIPTION
This action is explained in all notebooks.

Like the 3rd section.

5. ALGORITHM DESCRIPTION
This section is explained in all notebooks.

6. STRUCTURE DIAGRAM
This section is explained in this report.

7. ERROR INDICATIONS
This section is explained in this report. (if needed)

8. HAND TRACE
This action is explained in all notebooks.

I wrote functions in order of flow of my thinking. After each main block in notebooks, I wrote tests to ensure that every function in these blocks runs correctly.

9. EXTRA CREDIT
This section is explained in this report. (if needed)

10. CORRECTNESS ARGUMENT
This section is explained in this report. (if needed)

11. LISTINGS
All files are notebooks, python code and this report.

12. COMPILATION TYPESCRIPT
Not need to conduct compilation. Because all blocks of code are run in previous session and printed out the result if needed.

13. TEST DESCRIPTION
This action is explained in all notebooks.

14. TEST TYPESCRIPTS
This action is explained in all notebooks.

15. KNOWN BUGS
This section is explained in this report. (if needed)

16. POSSIBLE IMPROVEMENTS
This section is explained in this report. (if needed)

17. COMMENTS
This section is explained in this report. (if needed)

I. General idea

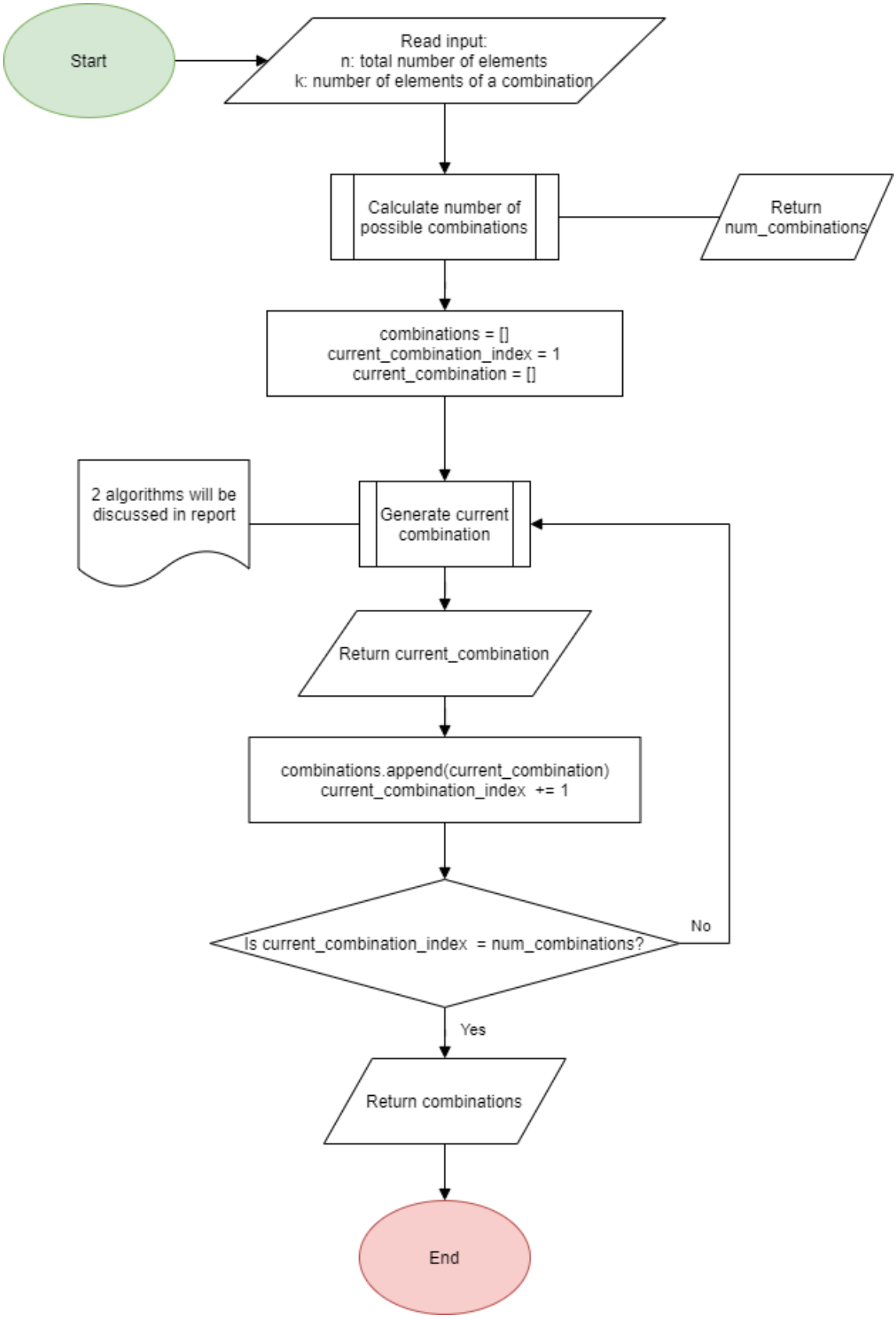

Figure 1. Flow chart of general program

The program in this assignment is aimed for generating and searching a bunch of combinations.

Suppose we have an array contained many elements that need to be re-arranged into k-elements combinations. Because some arrays may have different elements but same length, so instead of generating many times for these arrays, we just need to generate all combinations of an index array (length k), after that, mapping the index array to original arrays. Doing this way can reduce the complexity of the algorithm and re-use the generated combinations.

By example, suppose we need to generate all 2-elements combinations of 2 arrays: [10, 20, 30], [5, 4, 6]. First, we choose an index-array: [0, 1, 2] to conduct generating on it. Second, we generate all possible combinations of the index-array, which have to be: [0, 1], [0, 2], [1, 2]. In the third stage, we map the generated array to original arrays, which are: [10, 20], [10, 30], [20, 30] and [5, 4], [5, 6], [4, 6].

To simplify the implementation, in this report, we just use index-array to generating. In the real case, the mapping stage needs to be implemented.

In the generating phase, we assign two numbers as inputs of the function:

- **n**: Number of elements in the array
- **k**: Number of elements in a combination

The number of all possible combinations can be calculated as fomula:

$$C_k^n = \binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

The output combinations can be served to another purposes (eg: Testing, Gaming, Lottery, etc.)

The general idea is presented in the *Figure 1.*

II. The first method – Sequential generation

The first method is an easy strategy to generating all possible combinations. We start with a combination, which has k elements in the left-most (or right-most) side of an given index-array. For the next $C_k^n - 1$ steps, we move the right-most element, which doesn't have maximum value of its position. For each step of moving, we achive a combination.

The algoritm can be seen in *Table 1, 2.*

PROCEDURE:		**generate_all_combinations**
Input:		**n:** number of elements **k:** number of elements in a combination
Output:		**combinations:** all combinations will be generated
Algorithm:	1	*/* Calculate total of possible combinations can be
	2	generated from n, k
	3	*/*
	4	**SET** number_of_combinations := nck(n, k)
	5	
	6	**SET** combinations := []
	7	
	8	*/* The first combination is an empty list */*
	9	**SET** current_combination := []
	10	
	11	**FOR each** i **IN** number_of_combinations:
	12	**SET** next_combination := generate_next_combination(n, k,
	13	current_combination)
	14	
	15	**STORE** next_combination **IN** combinations
	16	**SET** current_combination := next_combination
	17	**END FOR**

Table 1. Algorithm of generate_all_combinations()

PROCEDURE:		**generate_next_combination**
Input:		**n:** Number of elements **k:** Number of elements in a combination **current_combination:** A combination of previous step
Output:		**successor:** The next combination
Algorithm:	1	** Check if the current combination is the last combination*
	2	*that can be generated, then return NULL*
	3	**/*
	4	**SET** first_element := current_combination[0]
	5	
	6	**IF** first_element = n+1-k **THEN:**
	7	**RETURN** NULL
	8	**END IF**
	9	
	10	*/* Copy all elements of current combination to successor */*
	11	**SET** successor := current_combination
	12	
	13	**FOR** i **TO** k:
	14	*/* Use reversed element of i instead of I */*
	15	**SET** ri := k-1-i
	16	
	17	**IF** successor[ri] < n-i **THEN:**
	18	**SET** successor[ri] := successor[ri] + 1
	19	
	20	**IF** ri < k-1 **THEN:**
	21	**FOR** j=ri+1 **TO** k:
	22	successor[j] := successor[j] + j - ri
	23	**END FOR**
	24	**END IF**
	25	
	26	**BREAK**
	27	
	28	**END IF**
	29	**END FOR**
	30	

Table 2. Algorithm of generate_next_combination()

For implementation, instead of using recursion, we develop a class to contain the main logic of the generating function, *figure 2*. When calling *next()* function, a next-combination is returned.

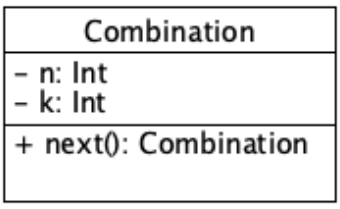

Figure 2. Class diagram of Combination

The first method can be implemented easily, but it has some trouble in performance when the n, k are huge. Due to the main idea of this method is sequential generation, we cannot generate a combination without the previous combination. It seems hard to divide the generating process into smaller pieces, so it may not take an advantage of distributed computing or parallel computing.

Tools and Experiment

Tools:

- OS: **CentOS 7**
- CPU: **Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz**
- Cores: **16**
- Language: **Python 3.5**
- Algorithm: **Sequential generation**

File listing:

- Jupiter notebooks: **notebooks/1st-method.ipynb**
- Python code: **src/1st_method.py**
- Output log of running Python code: **1_phi.log**

This method uses just one core for processing so we cannot take the whole power of the computer, *Figure 3*.

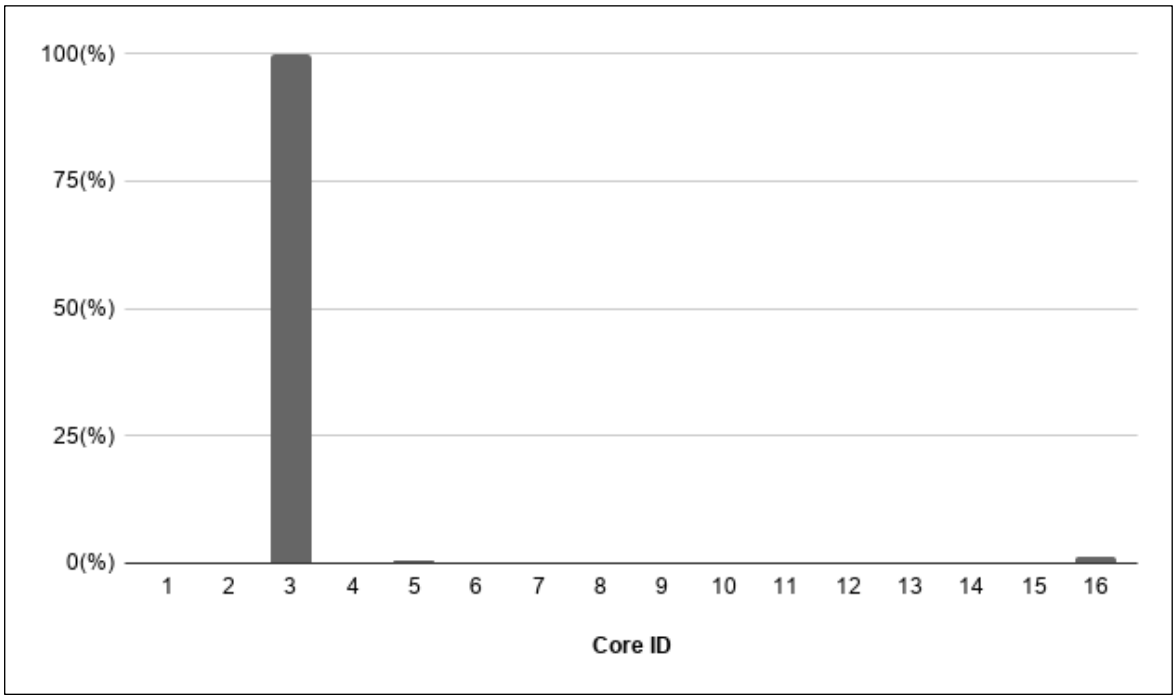

Figure 3. CPU utilization of using 1st method

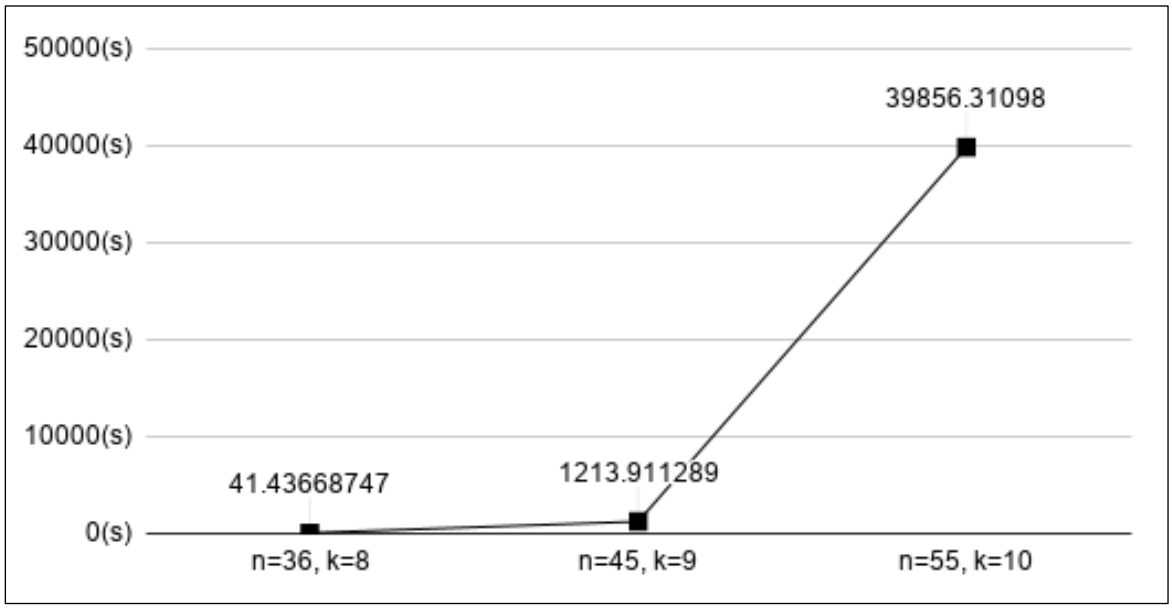

Figure 4 . Execution time of 1st method

III. The second method – By-index generation

The 2nd method is borned to take advantages of parallel computing.

In mathematic, we know that each number in [1, C_k^n] can be presented by an unique sum of combinadics:

$$C = C_{k-1}^{n1} + C_{k-2}^{n2} + \cdots + C_1^{n(k-1)}$$

We consider each number in [1, C_k^n] as an index of a combination, so we can find a combination independently with the others. This feature is suitable for conducting a parallel algorithm.

By example, suppose we have to generate combinations of an index-array [1, 2, 3], so all 2-elements combinations can be generated in *table 3*:

index	combination
1	[1, 2]
2	[1, 3]
3	[2, 3]

Table 3. A combinadics example

When running this method, we need to find the available CPUs (or cores) of the computer, then divide the indexes of combinations into pieces, which fit suitable with the architecture of CPU. For example, some CPUs can reach maximum performance when each process runs per core, some others are 2-processes per core, so it depends on the architecture of a chip.

The algorithm of this method can be seen in *Table 4.*

PROCEDURE:		generate_combination_by_index
Input:		**n:** Number of elements **k:** Number of elements in a combination **index:** Index of a combination
Output:		**combination:** A combination is generated
Algorithm:	1	*/* Calculate total of possible combinations can be
	2	generated from n, k
	3	*/*
	4	**SET** num_combinations := nck(n, k)
	5	
	6	**SET** m := num_combinations - 1 - index
	7	
	8	**SET** guess := n - 1
	9	
	10	**FOR each** i **IN** k:
	11	**SET** ri := k - i
	12	**SET** take := nck(guess, ri)
	13	
	14	**WHILE** take > m:
	15	**SET** guess := guess - 1
	16	**SET** take := nck(guess, ri)
	17	**END WHILE**
	18	
	19	**SET** m := m - take
	20	**SET** current_element := n-guess
	21	
	22	**STORE** current_element **IN** combination
	23	**END FOR**

Table 4. Algorithm of generate_ combination_by_index()

Tools and Experiment

Tools:

- OS: **CentOS 7**
- CPU: **Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz**
- Cores: **32**
- Language: **Python 3.5**
- Libraries: **Multiprocessing**
- Algorithm: **By-index Generation**

File listing:

- Jupiter notebooks: **notebooks/2nd-method.ipynb**
- Python code: **src/2nd_method.py**
- Output log running Python code: **2_vpcc_long.log, 2_vpcc_long_huge.log**

This method uses all cores for processing, *Figure 5.* Because we just use 30 cores in 32 cores of the experiment computer, so the core 23th and 27th don't reach the highest performance.

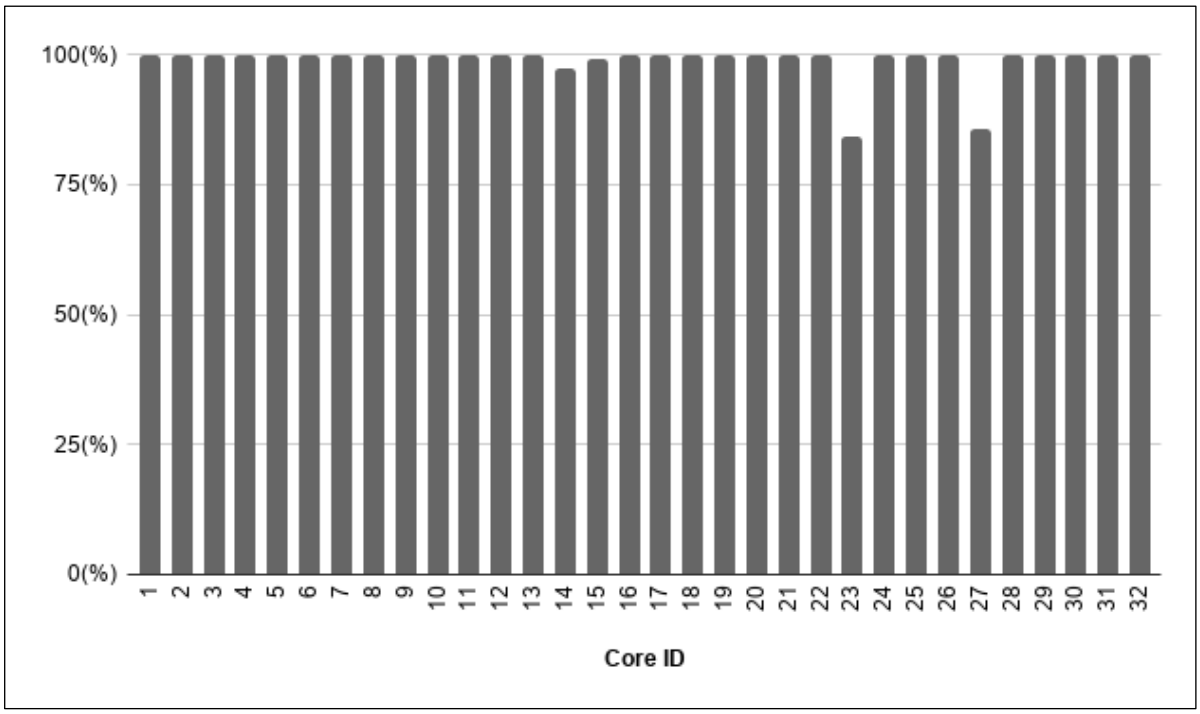

Figure 5. CPU utilization of using 2nd method

When using more cores for processing, the total of execution time of 2nd method is decreased (Figure 6).

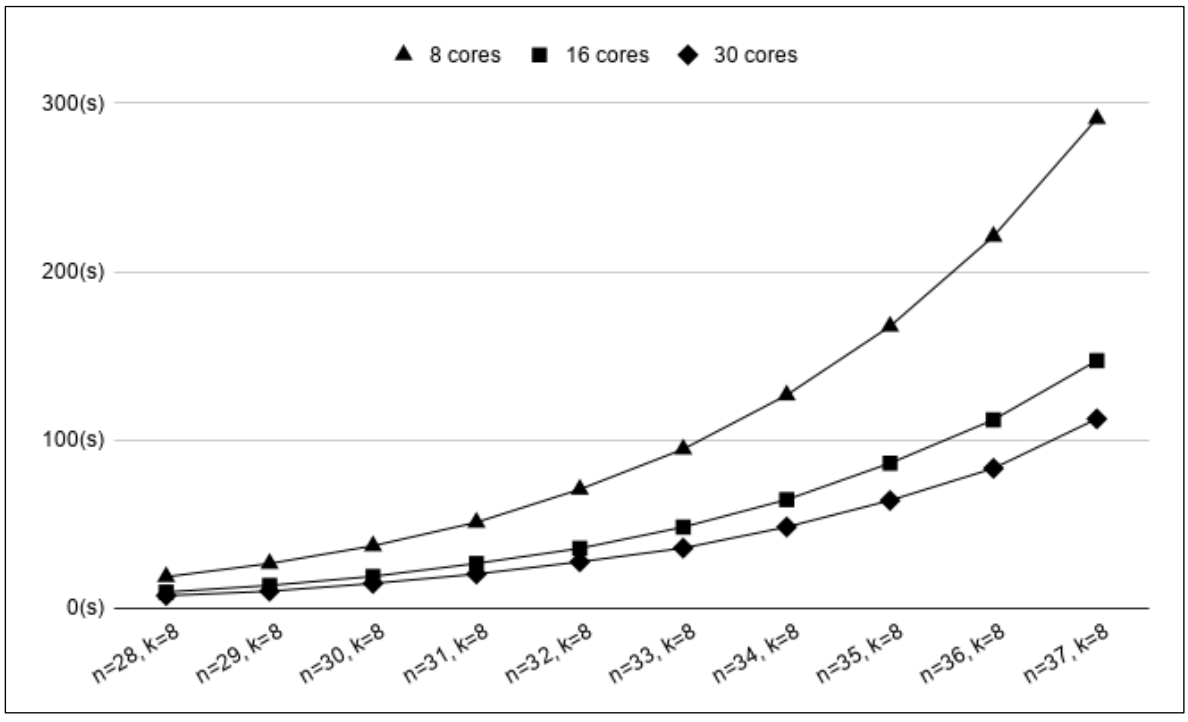

Figure 6. Execution time of 2nd method

Figure 7, 8 show the execution time per core of CPU. Those seem be equal, so we can conclude that the jobs divided to each core is fair.

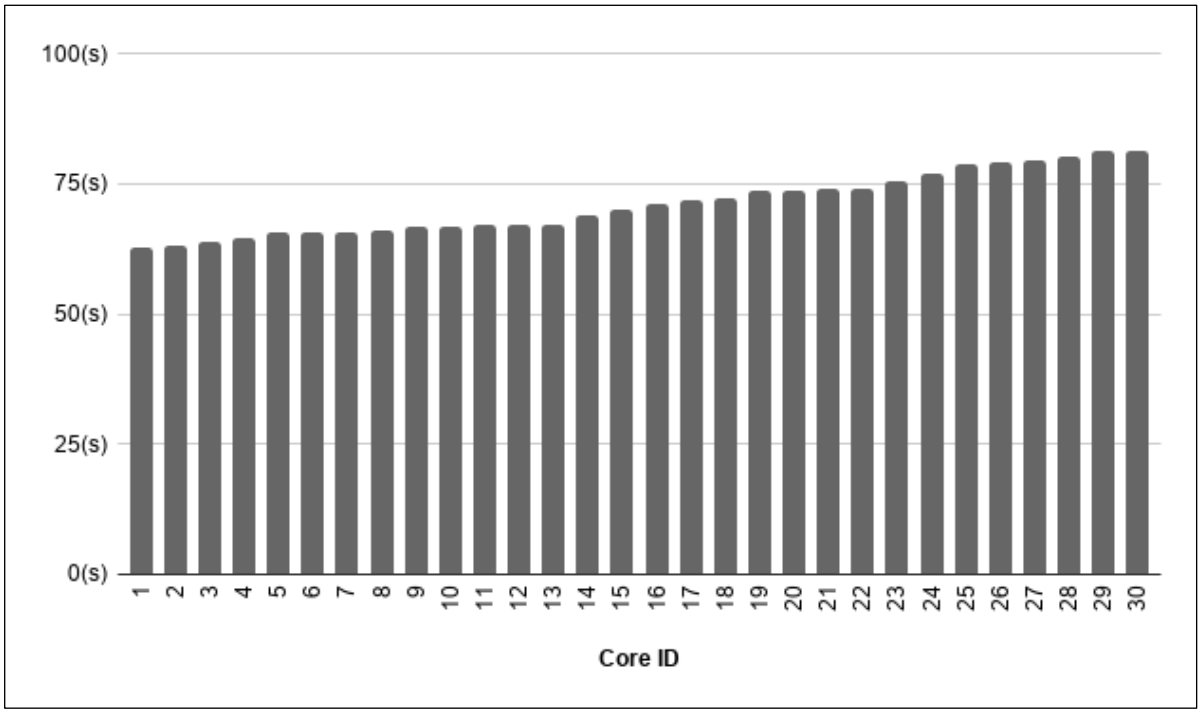

Figure 7. Execution time per each CPU-core of running 2nd method with n=36, k=8

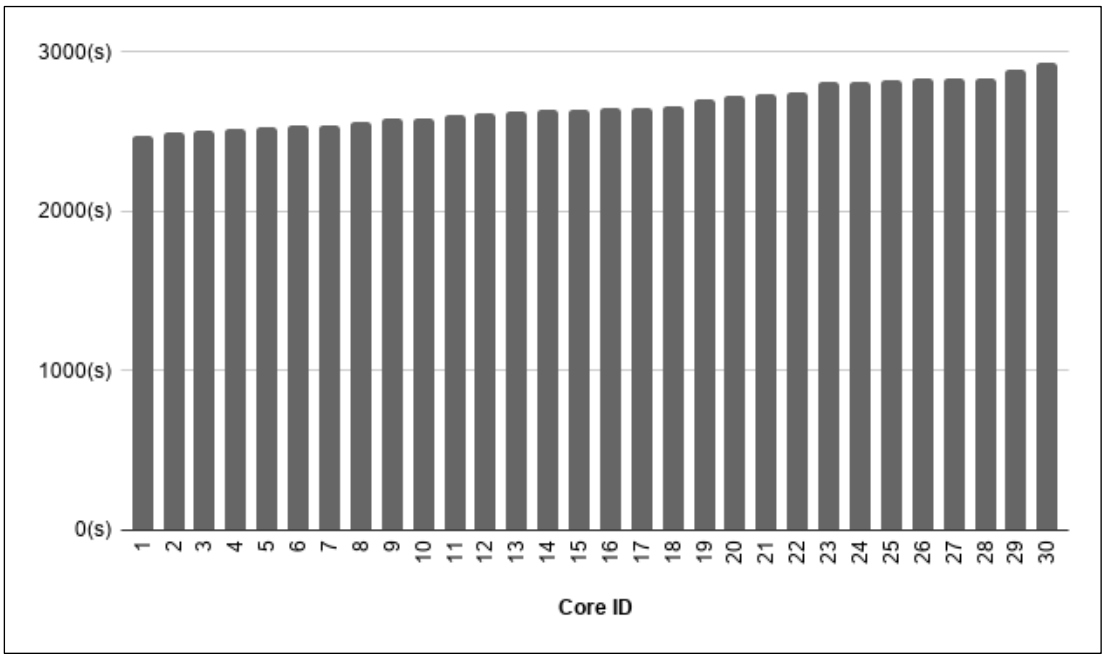

Figure 8. Execution time per each CPU-core of running 2nd method with n=45, k=9

IV. Comment

In this assignment, we develop 2 algorithms to generating combinations, which can be used in different purpose. The first method (Sequential generation) is simple but has a better performance when generating combinations, compare to the 2nd method (By-index generation).

The 2nd method is useful for the searching, when we know exactly the index of a combination, or when we need to generate a small number of combinations in a given inteval of indexes.

The reasonable strategy is to use 1st method for generating all combinations for given n, k and 2nd method for searching, or generating a small number of combinations.

Some works can be done to achive a better performance:

- Port Python code to lower language like C/C++/Rust. Python is a high-level language, so it uses a lot of abstract code to communicate with the lower platform, which can lead to low performance. One of reasons can be told is dynamic typing and dynamic allocating.
- Find a way to divide a job in 1st method into smaller jobs, so this method can be executed in parallel.
- Distribute jobs of 2nd method to many computers. We need to use another libraries like OpenMP or MPI to take both advatages of parallel and distributed computing.

V. References

James McCaffrey (2004), Generating the mth Lexicographical Element of a Mathematical Combination