Department of

Software Engineering

BRAUDE College of Engineering, karmiel

Capstone Project Phase B

Algorithms for generating permutations

(22-2-R-3)

Students:

ID	Last name	First name	e-mail
205962137	Idov	Yaniv	Yanividov12@gmail.com
204586507	Cohen	Omri	Omric12@gmail.com

Supervisor:

Prof.Shmuel Zaks

Table of content

Abstract

In our project we will explore different algorithms for generating permutations and discuss the complexities of their runtime. For each algorithm, in addition to generating all n! permutations, we will present how to get the appropriate index given the permutation from the list, and how to get the right permutation given its index. In addition, in writing this book, we will refer to the software products we created and explain how to use them and how they provide a solution and an integral part in understanding the entire research process.

Introduction

A permutation is a mathematical technique that determines the number of possible arrangements in a set when the order of the arrangements matters. Common mathematical problems involve choosing only several items from a set of items in a certain order. Permutations are frequently confused with another mathematical technique called combinations. However, in combinations, the order of the chosen items does not influence the selection. In other words, the arrangements a,b and b,e in permutations are considered different arrangements, while in combinations, these arrangements are equal. The general permutation formula is expressed in the following way:

$$p(n,k) = \frac{n!}{(n-k)!}$$

Where: n – the total number of elements in a set, k – the number of selected elements arranged in a specific order. Permutations are a very powerful technique for counting the number of ways things can be done or arranged in a sequence. Permutations are used in almost every branch of mathematics, and in many other fields of science. In computer science, they are used for analyzing sorting algorithms; in quantum physics, for describing states of particles; and in biology, for describing RNA sequences. The number of permutations of n distinct objects is n factorial, usually written as n!, which means the product of all positive integers less than or equal to n.(In this section, in order to give a precise definition of the concept of permutations, we used the website https://corporatefinanceinstitute.com/.)

Background and Related Work

Since in the world of science there is a great use of the whole subject related to permutations, this subject intrigues many scholars. Over the years a lot of algorithms have been invented that deal with creating a list of all the n! permutations, finding the permutation that corresponds to a specific index and finding the index that corresponds to the relevant permutation. In some of the algorithms we can see very similar mechanisms with an attempt to improve the runtime of the algorithm as much as possible. For example, the algorithms of the most familiar names in the field - Jhnson, Trotter, Ehrlich and Dershowits generate each successive permutation by transposing two adjacent items of the preceding permutation - these four algorithms differ only in the manner in which they determine the candidate pair of elements to be exchanged. On the other hand, there are algorithms that work on a different mechanism. For example, in the algorithms invented by our supervisor for the project Professor Shmuel Zaks, each successive permutation is generated by reversing a suffix of the preceding permutation.

Project Review and Process Description

Description of the research and development process

To conduct our research, the first thing we did was to understand what the motivation for the topic was. We did not know that the topic of permutations is relevant to so many areas and uses. After realizing that there are many algorithms that give an answer to the creation and use of permutations, we chose in consultation with our supervisor several algorithms that we think are the most interesting. We came to know that in order to understand how some of the algorithms work we needed a certain order of work. For example, we found that to understand why Nachum Dershowitz's loopless algorithm was more efficient than many others, we had to first understand how the algorithm of Johnson and Trotter's works and what its time complexity. One of the major challenges encountered when conducting the study was to understand how the stages of generating the list of all n! permutations and how exactly finding the index given the permutation in the article "Generation of all permutations by adjacent transpositions" work. Another challenge we had to face is how to implement the GUI software. We have not previously experimented with the implementation of a GUI of this form. After searching and researching for tools that can help us, we decided to use OpenCV. From the side of building the software, we decided to implement 2 separate software in order to transfer in an optimal way what we did in our project. Thoughts we had at the beginning were in which languages to write each of them. We decided that for the experience we want to ty as many languages as possible, we chose 2 languages and not 1 in order to gain additional experience in writing and implementing algorithms in several languages. The testing process included hands-on experience with the system ourselves, making sure all the buttons work and perform the functionality we wanted them to do in the first place. Regarding the system tests, we made sure that the system works as it should by testing it ourselves, and made sure that all the buttons work and perform the functionality we wanted them to perform in the first place.

Product:

As we explained earlier, in the following pages of this book you will find the product of our research - an in-depth and thorough analysis while providing as easy running examples as possible to understand the group of algorithms we have chosen to study. The explanations are accompanied by details that we think were missing in the articles themselves, and also accompanied by an analysis of the time complexity of the phases of the algorithm - a detail that in most articles did not appear at all, and not always easy and trivial to understand. In addition, in terms of the software, our product is 2 separate software that together help the reader of the study to visually understand all the work we have done. The first program is an implementation of all the algorithms which also includes the ability to compare running times between the different stages of all the algorithms. The second program is a GUI program, in this program we actually allow the user to run our algorithms step by step, where each step is accompanied by explanations of the essence of the operations and a visualization, so that even a user who does not have a background in computer science can understand how the algorithms work. The user has full control in terms of running the software, he can move steps forward and backward as he wishes as many times as he likes.

Lexicographic order

The material that formed the basis of our findings for this algorithm is taken from the article of Edward M.Reingold, Jurg Nievergelt and Narsingh Deo [1]. In addition, in order to give a precise definition to the concept of a factorial representation, we used the Wikipedia website. When applied to numbers, lexicographic order is increasing numerical order, (numerical order is a way of arranging a sequence of numbers. This could be in ascending or descending order. For example, if you ordered a set of numbers into ascending order such as 2, 55, 103, 256, 802. ... Or order a set of numbers from lowest to highest.), i.e., increasing numerical order (numbers read left to right). For example, the permutations of {1,2,3} in lexicographic order are 123, 132, 213, 231, 312, and 321.

To be exact , If $\varphi = (\varphi_1, \varphi_2, \dots, \varphi_n)$ and $\tau = (\tau_1, \tau_2, \dots, \tau_n)$ we say that φ lexicographically less then τ if and only if, for some $k \geq 1$, we have $\varphi_j = \tau_j$ for all $j < k$ and $\varphi_k < \tau_k$.

As a preview to our explanation of how the algorithm works, we will refer to the representation of numbers by factorial representation. The algorithm uses this representation in steps we will expand on later - finding the permutation given the index and vice versa.

The factorial number system is a mixed radix numeral system: the I - th digit from the right has base i, which means that the digit must be strictly less than i, and that (taking into account the bases of the less significant digits) its value to be multiplied by (i − 1)! (its place value).

From this it follows that the rightmost digit is always 0, the second can be 0 or 1, the third 0, 1 or 2, and so on. The factorial number system is sometimes defined with the 0! place omitted because it is always zero.

In this explanation, a factorial number representation will be flagged by a subscript "!", so for instance $3{:}4{:}1{:}0{:}1{:}0_!$ stands for $3_5 4_4 1_3 0_2 1_1 0_0$, whose value is

$$= 3\times5! + 4\times4! + 1\times3! + 0\times2! + 1\times1! + 0\times0!$$
$$= ((((3\times5 + 4)\times4 + 1)\times3 + 0)\times2 + 1)\times1 + 0$$
$$= 463_{10}.$$

(The place value is one less than the radix position, which is why these equations begin with 5!.)

General properties of mixed radix number systems also apply to the factorial number system. For instance, one can convert a number into factorial representation producing digits from right to left, by repeatedly dividing the number by the radix (1, 2, 3, ...), taking the remainder as digits, and continuing with the integer quotient, until this quotient becomes 0.

For example, 463_{10} can be transformed into a factorial representation by these successive divisions:

463 ÷ 1 = 463, remainder 0
463 ÷ 2 = 231, remainder 1
231 ÷ 3 = 77, remainder 0
77 ÷ 4 = 19, remainder 1
19 ÷ 5 = 3, remainder 4
3 ÷ 6 = 0, remainder 3

The process terminates when the quotient reaches zero. Reading the remainders backward gives $3{:}4{:}1{:}0{:}1{:}0_!$.

In our explanation of the algorithm, we will address three aspects:

1. Create all the n! permutations in a lexicographic order.

2. Given an index, we will explain how to get to the relevant permutation.

3. Given permutation, we will explain how to get its position in the order of creation, i.e., what its index is.

Given a permutation, we will explain how to get to the relevant permutation

The index of the permutation $\delta = (\varphi_1, \varphi_2, \dots, \varphi_n)$ is defined by:

$$\mathrm{index}_1[(1)] = 0$$

$$\mathrm{index}_n(\delta) = (\varphi_1 - 1)(n-1) + \mathrm{index}_{n-1}(\delta')$$

where $\delta' = (\varphi'_1, \varphi'_2, \dots, \varphi'_{n-1})$ is the permutation obtained from δ by deleting φ_1 and reducing by one all the elements greater then φ_1

For example, suppose we have the permutation 35421 and we are interested in knowing its position in our creation series, i.e., what its index is.

$$\begin{array}{lr}
(3-1)(5-1)! & 35241 \\
+(4-1)(4-1)! & 4231 \\
+(2-1)(3-1)! & 231 \\
+(2-1)(2-1)! & 21 \\
+\mathrm{index}_1[(1)] = 0 & 1
\end{array}$$

We have obtained that the index of the given permutation is 69

Given an index, we will explain how to get to the relevant permutation

Note that from the calculation we did in the previous phase, we have the representation of the permutation index in the form of factorial representation:

$$\begin{array}{lcc}
(3-1)(5-1)! & & 2 * 4! \\
+(4-1)(4-1)! & & 3 * 3! \\
+(2-1)(3-1)! & = & 1 * 2! \\
+(2-1)(2-1)! & & 1 * 1! \\
+\mathrm{index}_1[(1)] = 0 & & 0 * 0!
\end{array}$$

We will use this representation of the index to get to the permutation.

Looking only at the coefficients multiplied, we get an I = 2: 3: 1: 1: $0_!$

Our goal now is to get back the value of our permutation, which was 35241

We are going to look at the digits from right to left.

The process is going to work as follows - we are going to go through the digits of I from right to left, at each step we will copy the digit we are looking at plus 1 to the leftmost place in our array and then chain to the right the previous content that was inside the array, adding 1 to all numbers that were larger than the figure we are looking at from I.

Example:

First, we are looking at the right most digit of I (From the factorial representation we know that the right most digit is always 0),

Consider you have an empty array. the process is first checking if there is any element bigger then 0. Since the array is empty, the answer Is no. so what we do now is writing 0+1 in our array and moving to look at the next digit of I from the right which is 1. Result from this phase:

$$\boxed{1}$$

Like before, first we are scanning our array to search if there are any elements bigger then the number we are currently looking at from I (1) the answer is no. so we just writing the number we are looking at from I, adding 1 to it at the left most place in our array and concatenate to the right the previews content. Result from this phase:

$$\boxed{21}$$

Now we are looking at the third digit from the right in I – > 1.

Looking at the content of the array from the previous phase, we see that the left most digit is 2 which is greater the 1, so when concatenating the content, we will add 1 to it. Result from this phase:

$$\boxed{231}$$

Now we are looking at the fourth digit from the right in I – >3

there are no bigger numbers then 3 in our previous array, so we just concatenating the previous content. Result from this phase:

$$\boxed{4231}$$

Finally, we are looking at the left most digit of I - > 2

We will remember to add 1 to all the numbers from the previous array that are greater than 2 and concatenate them to the right. Result from this phase:

$$\boxed{35231}$$

Create all the n! permutations in a lexicographic order

Creating all the n! permutations in a lexicographic order done by 3 main steps:

find rightmost place where $\varphi_i < \varphi_{i+1}$

find φ_j the smallest element to the right of φ_i and greater than it

interchange and then reverse $\varphi_{i+1}, \dots, \varphi_n$

The algorithm starts by printing the only permutation that contains all n numbers in ascending order from left to right, and ends when i = 0, which occurs if and only if we get back to our initial permutation of ascending order from left to right (without printing it again).

```
For j=1 to n do $\varphi_j \leftarrow j$
$i \leftarrow 1$
While i ≠ 0 do {
        Output $\delta = (\varphi_1, \varphi_2, \dots, \varphi_n)$
        [[find rightmost place where $\varphi_i < \varphi_{i+1}$ ]]
        i← n-1
        While $\varphi_i > \varphi_{i+1}$ do i ← i -1
        [[find $\varphi_j$ the smallest element to the right of $\varphi_i$ and greater than it]]
        j← n
        while $\varphi_i > \varphi_j$ do j ← j -1
        [[interchange and then reverse $\varphi_{i+1}, \dots, \varphi_n$ ]]
        $\varphi_i \leftrightarrow \varphi_j$
        r ← n
        s ← i+1
        while $r > s$ do { $\varphi_r \leftrightarrow \varphi_s$
                           r ← r-1
                           s ← s+1 }
}
```

Note that the aim of the first phase is the code is to update i, the aim of the second phase is to update j, and the aim of the third phase is to make the interchanges. Let's look at the process of creating all permutations for n = 3:

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6
First phase	Output=**123** I=2	Output=**132** I=1	Output=**213** I=2	Output=**231** I=1	Output=**312** I=2	Output=**321** I=0
Second phase	J=3	J=3	J=3	J=2	J=3	End
Third phase	132	231	231	312	321	End

Complexity analysis

We will analyze the complexity of our algorithm, according to the three steps we have explained before: index → permutation, permutation → index and generating all the permutations.

Index → permutation

First, we need to converter our index into factorial representation - an action performed in O(n).

We then begin the process of creating the permutation from the index represented in the factorial basis. We start from an empty array, and at each stage we increase the array size by 1 by inserting another digit and scanning the values from the previous step to copy its contents to the new array, while making the required comparisons and raising the relevant values by 1. each comparison and increment are done at O(1), note that at each step we are scanning an array bigger by 1 then the size of the array from the previous phase. Let us note that the number of steps is according to the amount of numbers in the factorial representation, starting from an array of size 1, and ending with an array of size n. We then can conclude that to estimate the time complexity of this part of the algorithm we need to refer to the changing size of the array at each step: $\sum_{i=1}^{n} i = O(n^2)$ .overall, given an index, our algorithm generates the corresponding permutation In $\mathrm{Max}(\mathrm{n}, \mathrm{n}^2) = O(n^2)$

Permutation → index

Given a permutation, to get the corresponding index, we must perform a multiplication operation that is performed at 0 (1) and then there is a recursive call for δ', where in creation of δ' we remove the leftmost digit, go over the array *members and decrease in 1 the relevant values, thus δ' production takes 0 (n). With the addition of the recursive call, we get:* $\sum_{i=n}^{1} 1+i = O(n^2)$

Generation of all the permutations

To evaluate the time complexity of this section we would need to address two main operations – the number of interchanges and the number of comparisons between elements of the permutation. We will define I_n– total number of interchanges in our algorithm, C_n – total number of comparisons. Its clear that both of these operations takes O(1), The question is how many times do we perform them. Note that we have 2 comparisons: $\varphi_i > \varphi_{i+1}$, $\varphi_i > \varphi_{i+1}$ and two interchanges $\varphi_i \leftrightarrow \varphi_j, \varphi_r \leftrightarrow \varphi_s$. For the convenient of the explanation, assume that the output of our list of permutations divided into n parts, where in each part the first element in each permutation is the same element, and all n parts have all the possible values for the first element. That is, in the first part are all the permutations whose first element is 1, in the second part are all the permutations whose first element is 2 and so on, up to the nth part where the value of the first element is n.

In each of these n parts we are doing I_{n-1} interchanges to generate all the permutations with the same first element. Note that the transition from the last permutation of a section to the first permutation of the next section cost us $\left\lfloor\frac{n+1}{2}\right\rfloor$. (swapping all the elements in the array, the first one with the last one, the second with the element in the n-1 place and so on.) since there are n sections, n-1 of these operations needs to be done, so: $(n-1) * \left\lfloor\frac{n+1}{2}\right\rfloor$.
from that we conclude that $I_n = nI_{n-1} + (n-1)\left\lfloor\frac{n+1}{2}\right\rfloor$ *and* $I_1 = 0$

We will present the equation differently to make it easier for us to solve:

$$I_n = nI_{n-1} + (n-1)\left\lfloor\frac{n+1}{2}\right\rfloor$$
$$\left\lfloor\frac{n+1}{2}\right\rfloor + I_n = nI_{n-1} + (n-1)\left\lfloor\frac{n+1}{2}\right\rfloor + \left\lfloor\frac{n+1}{2}\right\rfloor$$
$$S_n = nI_{n-1} + (n-1)\left\lfloor\frac{n+1}{2}\right\rfloor + \left\lfloor\frac{n+1}{2}\right\rfloor$$
$$S_n = nI_{n-1} + n\left\lfloor\frac{n+1}{2}\right\rfloor$$
$$S_n = n\left(I_{n-1} + \left\lfloor\frac{n+1}{2}\right\rfloor\right)$$
$$S_n = n(S_{n-1} + \varepsilon_{n-1}), \varepsilon_n = 0 \textit{ if } n \textit{ is odd}, 1 \textit{ if } n \textit{ is even}$$
$$\textit{then } S_n = n!\left(1 + \frac{1}{2!} + \frac{1}{4!} + \frac{1}{6!} + \cdots + \frac{1}{\left(2\left\lfloor\frac{n-1}{2}\right\rfloor!\right)}\right) = n!\sum_{j=0}^{\lfloor(n-1)/2\rfloor}\frac{1}{(2j)!}$$

We defined S_n to be $\left\lfloor\frac{n+1}{2}\right\rfloor + I_n$,therefore we can write that

$$I_n = n!\,(\sum_{j=0}^{\lfloor (n-1)/2 \rfloor} \frac{1}{(2j)!}) - \left\lfloor \frac{n+1}{2} \right\rfloor$$

Note that when i=0 the algorithm generates the very first permutation of numbers in ascending order, without printing it again. It takes $\left\lfloor \frac{n+2}{2} \right\rfloor$ *interchanges. So* $I_n = n!\,(\sum_{j=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \frac{1}{(2j)!}) - \left\lfloor \frac{n+1}{2} \right\rfloor + \left\lfloor \frac{n+2}{2} \right\rfloor$

$$I_n = n!\,(\sum_{j=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \frac{1}{(2j)!}) + \varepsilon_n$$

$$I_n = n!\,cosh1 \approx 1.54308n!$$

$$C_n = \left(\frac{3}{2}e - 1\right) n! \approx 3.07742n!$$

The total complexity is $1.54308n! + 3.07742n! = 4.6205n!$

Suffix based algorithm for generation of permutations

The material that formed the basis of our findings for this algorithm is taken from the article of Prof. Shmuel Zaks [2]. Unlike many different permutation algorithms in which each step changes the order of 2 adjacent members, in the algorithm we will present to you now, we create permutations by changing a certain suffix of the previous permutation, the size of the suffix we are looking at in each step change, as we will explain later.

Generating all the n! permutations:

As stated in the previous paragraph, to produce a permutation we use the suffix of the previous permutation. To know what size of suffix to look at from the previous permutation, we will define s_n - the sequence of sizes of these suffixes. Since there are n! permutations for n and for every n the first permutation is 1,…,n , the length of s_n is always n-1. s_n is defined recursively as follows:

$$\boxed{\begin{array}{ll} s_2 = 2 & \\ s_n = (s_{n-1}n)^{n-1}s_{n-1} & n > 2 \end{array}}$$

– Note that a sequence is written as a concatenation of its elements.

Examples for n =3,4:

n=3:

$s_3 = 23232$

Let's understand how we use this value of s_3 to generate the permutations. Moving from each permutation to next starting from the first permutation of 123, we are reading s_3 from left to right to know which size of suffix we are going to reverse.

From the first permutation to the next we are looking at a suffix size of 2, from the second permutation to the third the size of the suffix is 3, and so on.

$1\underline{23}$ $2\underline{31}$ $3\underline{12}$

$\underline{132}$ $\underline{213}$ 321

n=4:

$s_4 = 23232423232423232423232$

$12\underline{34}$ $23\underline{41}$ $34\underline{12}$ $41\underline{23}$

$1\underline{243}$ $2\underline{314}$ $3\underline{421}$ $4\underline{132}$

$13\underline{42}$ $24\underline{13}$ $31\underline{24}$ $42\underline{31}$

$1\underline{324}$ $2\underline{431}$ $3\underline{142}$ $4\underline{213}$

$14\underline{23}$ $21\underline{34}$ $32\underline{41}$ $43\underline{12}$

$\underline{1432}$ $\underline{2143}$ $\underline{3214}$ 4321

Pseudo code of the algorithm:

```
$p_1p_2 \dots p_n := 1,2,..,n\ (start)$
**repeat**
 I:= next element of $s_n$ (starting with first one)
 $p_{n-i+1} \dots p_n := p_n \dots p_{n-i+1}$  $(reverse\ last\ i\ elements)$

**untill** $i\ is\ the\ last\ element\ of\ s_n$
```

Proof of validity:

We will use induction on n to show that the algorithm generates all the n! permutations starting with the permutation 1,2,…,n and ending with n,n-1,…,1.its clear to see that the assertion holds for n=2 because $s_n = 2$ and starting with the permutation 12 we then get 21. We will prove the assertion holds for n assuming it holds for n-1. To prove for n using the induction hypothesis, we need to look at how s_n is built. Because s_n starts with s_{n-1} we first generate (n-1)! Permutations Starting with 1,then, from the induction hypothesis, the last permutation is 1,n,n-1,…,2. Since the next value in s_n is n, we generate all the permutations starting from 2, starting from 2,3,…,n,1.in the same way we then generate all the permutations starting with 3.Over and above that, since (2,3,..,n,1) is the first permutation starting from 2 and obtained from (1,2,…,n) the first permutation starting from 1 by increasing each element by 1(notice n becomes 1) , and from the fact that all permutations starting from 1 and 2 generated from s_{n-1} the last permutation starting with 2 derived from the last permutation starting with 1 in the same way. We then can say that the first permutation starting with i,1<i≤n is i,i+1,…,n,1,2,…,i-1 and the last one is i,i-1,…,1,n,n-1,…,i+1.

Lemma. The average size of a suffix reversed by the generation is less than 2.8.

Proof. if we apply the sequence of suffix reversals $s_n n$ (starting and eding with permutation 1,2,…,n). in this case n appears in this sequence $n!\ /\ (n-1)!$ = n times. Further, n-1 appears in the positions that are multiples of (n-2)! But not of (n-1)!, namely $n!\ /\ (n-2)!\ -\ n!\ /\ (n-1)!$ Times. With induction the number i, $2 \le i < n\_$,appears in the sequence $n!\ /\ (i-1)!$ – $n!\ /\ i!$ times. Thus, the expected suffix size is : $\frac{1}{n!}\left[\, n \cdot \frac{n!}{(n-1)!} + \sum_{i=2}^{n-1} i\left(\frac{n!}{(i-1)!} - \frac{n!}{i!}\right)\right] = \frac{n}{(n-1)!} + \sum_{i=2}^{n-1} \frac{1}{(i-2)!} = \frac{1+(n-1)}{(n-1)!} + \sum_{i=0}^{n-3} \frac{1}{(i)!} = \sum_{i=0}^{n-1} \frac{1}{(i)!} < e\ < 2.8$

Overall this phase of generating the list of n! permutations cost us O(n!) because the average size of the suffix used is 2.8 and there are n! permutations.

<u>Finding the appropriate index given the permutation</u>

Index(1) =1

Index($p_1, p_2, \dots, p_n) = (p_1 - 1)(n-1)! + index(p_2 - p_1, p_3 - p_1, \dots, p_n - p_1)$

With subtraction mod n

This comes from the fact that index(p) = $(p_1 - 1)(n-1)! +$ *{<u>position of p among permutations starting with</u>* $p_1 = index(p_2 - p_1, p_3 - p_1, \dots, p_n - p_1)$ *}.*

this operation takes O(n^2*).*

<u>Examples:</u>

Index (4321) = $3 \cdot 3! + index(321)$
$= 3 \cdot 3! + 2 \cdot 2! + index(21)$
$= 3 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + index(1)$
$= 3 \cdot 6 + 4 + 1 + 1 = 24$

$\boldsymbol{Index(4123)} = 3 \cdot 3! + index(123)$
$= 3 \cdot 3! + 0 \cdot 2! + index(12)$
$= 3 \cdot 3! + 0 \cdot 2! + 0 \cdot 1! + index(1)$
$= 3 \cdot 3! + 0 \cdot 2! + 0 \cdot 1!\ + 1 = 19$

Index (2413) = $1 \cdot 3! + index(231)$
$= 1 \cdot 3! + 1 \cdot 2! + index(12)$
$= 1 \cdot 3! + 1 \cdot 2! + 0 \cdot 1! + index(1)$
$= 6 + 2 + 0 + 1 = 9$

Index (3124) = $2 \cdot 3! + index(231)$
$= 2 \cdot 3! + 1 \cdot 2! + index(12)$
$= 2 \cdot 3! + 1 \cdot 2! + 0 \cdot 1! + index(1)$
$= 2 \cdot 3! + 1 \cdot 2! + 0 \cdot 1!\ + 1 = 15$

Index (2314) = $1 \cdot 3! + index(132)$
$= 1 \cdot 3! + 0 \cdot 2! + index(21)$
$= 1 \cdot 3! + 0 \cdot 2! + 1 \cdot 1! + index(1)$
$= 1 \cdot 3! + 0 \cdot 2! + 1 \cdot 1! + 1 = 8$

finding the appropriate permutation given the index

To do this we are going to use the factorial representation we have discussed previously in this book.

The first step of finding the permutation belongs to index r is writing the factorial representation of r-1:

r-1 = $a_1 \cdot 1! + a_2 \cdot 2! + \cdots + a_{n-1} \cdot (n-1)!\,, \quad where\ 0 \le a_i \le i$

And then apply the algorithm:

```
$p_1 := 1;$
$for\ j := 1\ to\ n-1\ do$
$\boldsymbol{begin}$
$for\ k := j+1\ downto\ 2\ do\ p_k := 1 + (\,p_{k-1} + a_j)mod(j+1);$
$p_1 := a_j + 1$
$\boldsymbol{end};$
```

Examples:

Finding the 24th permutation for n=4:

$24 - 1 = 1 \cdot 1! + 2 \cdot 2! + 3 \cdot 3!$
$for\ j = 1, a_1 = 1, we\ get\ p_1p_2 = 21$
$for\ j = 2, a_2 = 2, we\ get\ p_1p_2p_3 = 321$
$for\ j = 3, a_3 = 3, we\ get\ p_1p_2p_3p_4 = 4321$

Finding the 19th permutation for n=4

$19 - 1 = 0 \cdot 1! + 0 \cdot 2! + 3 \cdot 3!$
$for\ j = 1, a_1 = 0, we\ get\ p_1p_2 = 12$
$for\ j = 2, a_2 = 0, we\ get\ p_1p_2p_3 = 123$
$for\ j = 3, a_3 = 3, we\ get\ p_1p_2p_3p_4 = 4123$

$Finding\ the\ 9th\ permutation\ for\ n = 4$

$9 - 1 = 0 \cdot 1! + 1 \cdot 2! + 1 \cdot 3!$
$for\ j = 1, a_1 = 0, we\ get\ p_1p_2 = 12$
$for\ j = 2, a_2 = 1, we\ get\ p_1p_2p_3 = 231$
$for\ j = 3, a_3 = 1, we\ get\ p_1p_2p_3p_4 = 2413$

$Finding\ the\ 15th\ permutation\ for\ n = 4$

$15 - 1 = 1 \cdot 1! + 2 \cdot 2! + 3 \cdot 3!$
$for\ j = 1, a_1 = 0, we\ get\ p_1p_2 = 12$
$for\ j = 2, a_2 = 1, we\ get\ p_1p_2p_3 = 231$
$for\ j = 3, a_3 = 2, we\ get\ p_1p_2p_3p_4 = 3124$

$Finding\ the\ 8th\ permutation\ for\ n = 4$

$8 - 1 = 1 \cdot 1! + 0 \cdot 2! + 1 \cdot 3!$
$for\ j = 1, a_1 = 1, we\ get\ p_1p_2 = 21$
$for\ j = 2, a_2 = 0, we\ get\ p_1p_2p_3 = 132$
$for\ j = 3, a_3 = 1, we\ get\ p_1p_2p_3p_4 = 2314$

this operation takes O(n^2).

<u>More efficient way to generate s_n</u>

This algorithm generates the next element of s_n in $O(1)$ time and $O(n)$ space.

X *is the current value of s_n*

Follow[i] *is the number that will be generated in the next position which is multiple of i!, initialized to i+1, $2 \leq i \leq n$.*

Count[i] *is the number of times i has appeared since the last time i+1 appeared,*

$3 \leq i \leq n-1$

Count[n] *is the number of times n has been generated and* ***Count[n+1]*** *is a sentinel. (count[i], $3 \leq i \leq n+1$, is initially 0).*

NEXT algorithm

```
1 . if x ≠ 2 then x: = 2
else begin
2.     x:= follow[2];
3.    follow[2] :=3;
4.    Count[x] = count[x]+1;
5.    If count[x] = x-1 then
6.   begin count [x] :=0;
7.             Follow [x-1]:=follow[x];
8.            Follow[x]:= x+1
      end
end;
```

The program

```
X:=2;
While x≤n do begin output(x); NEXT end;
```

With initial values as stated above, outputs the sequence s_n.

Iteration 1	***Iteration 2***	***Iteration 3***	***Iteration 4***	***Iterarion 5***
Output = 2 *x=3* *follow[2]=3* *count[3]=1*	***output=3*** *x=2*	***output=2*** *x=3* *follow[2]=3* *count[3]=2* *count[3]=0* *follow[2]=4* *follow[3]=4*	***output=3*** *x=2*	***output =2*** *x=4* *follow[2]=3*

Notice there are no loops in the NEXT algorithm, hence the generation of each element takes O(1) . for every n there are n! -1 elements in s_n , hence the overall time complexity of generating s_n is O(n!)

:

Adjacent transpositions algorithm

The material that formed the basis of our findings for this algorithm is taken from the article of Shimon Even[3]. The way one permutation is changed to the next is by an adjacent transposition. A transposition consists of two items changing places. A transposition is called adjacent if the two items are adjacent. We denote by an arrow above each integer its direction, namely, the direction in which it tends to go. At the beginning all the directions pointing from right to left. For example, if n = 3, the first permutation in the list is: $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$

An integer k is mobile if there exists an integer smaller then k adjacent to k on the side where the direction of k points to. For example, in $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$ *, 3 and 2 are mobile.*

<u>The algorithm to produce the permutations:</u>

1. if there are no mobile integers, stop.

2. call the largest mobile integer m

3. let m switch places with the adjacent integer to which mth direction points to.

4. switch the direction of all integers k for which k>m. return to step (1)

There are n! Iterations and in each we will have to find the active element. in the worst case the complexity of the algorithm is O(n!n)

<u>Demonstrating of the algorithm</u>

n=2:

$$\overleftarrow{1}\ \overleftarrow{2}$$

$$\overleftarrow{2}\ \overleftarrow{1}$$

n=3 :

$$\begin{array}{ccc}
\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3} & \qquad & \overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1} \\
\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2} & & \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1} \\
\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2} & & \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}
\end{array}$$

n=4 :

$$\begin{array}{ccc}
\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{4} & \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4} & \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{4} \\
\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}\ \overleftarrow{3} & \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2} & \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{4}\ \overleftarrow{1} \\
\overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{3} & \overleftarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2} & \overleftarrow{2}\ \overleftarrow{4}\ \overrightarrow{3}\ \overleftarrow{1} \\
\overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3} & \overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2} & \overleftarrow{4}\ \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1} \\
\overrightarrow{4}\ \overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2} & \overrightarrow{4}\ \overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1} & \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3} \\
\overleftarrow{1}\ \overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2} & \overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1} & \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}\ \overrightarrow{3} \\
\overleftarrow{1}\ \overleftarrow{3}\ \overrightarrow{4}\ \overleftarrow{2} & \overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1} & \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}\ \overrightarrow{3} \\
\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}\ \overrightarrow{4} & \overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4} & \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}\ \overrightarrow{4}
\end{array}$$

Proof of validity

We will use induction for the proof of validity of the algorithm. In case n=2 we already saw that after one step starting from $\overleftarrow{1}\ \overleftarrow{2}$ we get $\overleftarrow{2}\ \overleftarrow{1}$ .since there are no mobile integers, the algorithm stops. Assume now that the algorithm works for n, and we want to show that this implies its validity for n+1. Notice that the direction of n+1 is to the left, and it will move to left until we get $\overleftarrow{n+1}\ \overleftarrow{1}\ \overleftarrow{2}\ \dots\ \overleftarrow{n}$. Until this point 1,2,…,n kept their relative positions and their original directions and all the permutations where n+1 is interwoven into this order have been generated, and n+1 is not mobile. Therefor, m is the same number that would be determined if the algorithm were applied to 1, 2,…,n only. Another thing to notice here is that the two integers which switch places are the same as if n+1 were not there. The next section that is going to happen now is n+1 moves all the way to the right. After that, a change which would have taken place among 1, 2,…,n if n+1 were missing takes place, and n+1 moves all the way to the other side. We then can conclude that for each of the permutations if 1, 2,…,n, the integer n+1 is places in all the possible intervals.in the last permutation of 1,2,..,n , after n+1 moves all the way to the other end, there are no mobile integers and thus the algorithm stops.

Generating the permutation from the index

for n=4 we need to calculate $a_4a_3a_2$.the meaning of every a_i is how many blank spaces does the number i has to its right in the array. For example, if $a_4 = 1$ then we know where to put the number 4 – in the second place in our empty array, starting to count from the right, because it's the only place which 4 would have 1 blank spot to its right.

b_n is the number of the current permutation (starting the enumeration from 0)

N makes n-1 moves, rests one move and then makes n-1 moves in the reverse direction. Thus, we can say that n has made q_n complete sweeps, each consist of n permutations, and has made r_n additional moves.

Rules we are going to use:

1. $b_n = nq_n + r_n\ , 0 \leq r_n < n$
2. $\mathrm{b}_{n-1} = q_n$

3. $a_i = \begin{cases} r_i & if\ q_i\ is\ even \\ i-1-r_i & if\ q_i\ is\ odd \end{cases}$

Examples:

Finding permutation number 15 (the 16th) in the generation for n=4:

$15 = 4 \cdot 3 + 3 \ \rightarrow a_4 = 4 - (3+1) = 0$

$b_3 = q_4$

$3 = 3 \cdot 1 + 0 \ \rightarrow a_3 = 3 - (0+1) = 2$

$b_2 = q_3$

$1 = 2 \cdot 0 + 1 \ \rightarrow a_2 = 1$

			4

3			4

3	2		4

3	2	1	4

Finding permutation number 5 (the 6th) in the generation for n=4

$5 = 4 \cdot 1 + 1 \ \rightarrow a_4 = 4 - (1+1) = 2$

$b_3 = q_4$

$1 = 3 \cdot 0 + 1 \ \rightarrow a_3 = 1$

$b_2 = q_3$

$0 = 2 \cdot 0 + 0 \ \rightarrow a_2 = 0$

	4		

	4	3	

	4	3	2

1	4	3	2

Finding permutation number 11 (the 12th) in the generation for n=4:

$$\begin{tikzcd}
11 = 4 \cdot 2 + 3 \; \rightarrow a_4 = 3 \arrow[d, "b_3 = q_4"] \\
2 = 3 \cdot 0 + 2 \; \rightarrow a_3 = 2 \arrow[d, "b_2 = q_3"] \\
0 = 2 \cdot 0 + 0 \; \rightarrow a_2 = 0
\end{tikzcd}$$

4			

4	3		

4	3		2

4	3	1	2

Modulo/remainder is a O (1) operation (it's essentially just a variation on division, which takes constant time on fixed-sized numbers), which is done n-1 times, therefor the time complexity here and in the following phase of finding the index from the permutation is O(n).

Finding the index from the permutation

Rules we are going to use:

First, $b_2 = r_2 = a_2$. From there on we find

$$r_i = \begin{cases} a_i & if\ b_{i-1}\ is\ even \\ i - 1 - a_i & if\ b_{i-1}\ is\ odd \end{cases}$$

$$and\ \ b_i = i \cdot b_{i-1} + r_i \qquad for\ i = 3,4, \dots , n$$

Examples:

3214 -> $a_4 = 0 , a_3 = 2 , a_2 = 1$
$b_2 = r_2 = a_2 = 1$

$r_3 = 3 - 1 - 2 = 0$
$b_3 = 3 \cdot 1 + 0 = 3$

$r_4 = 4 - 1 - 0 = 3$
$b_4 = 4 \cdot 3 + 3 = \boxed{15}$

1432 -> $a_4 = 2 , a_3 = 1 , a_2 = 0$
$b_2 = r_2 = a_2 = 0$

$r_3 = a_3 = 1$
$b_3 = 3 \cdot 0 + 1 = 1$

$r_4 = 4 - 1 - 2 = 1$
$b_4 = 4 \cdot 1 + 1 = \boxed{5}$

4312 -> $a_4 = 3 , a_3 = 2 , a_2 = 0$
$b_2 = r_2 = a_2 = 0$

$r_3 = a_3 = 2$
$b_3 = 3 \cdot 0 + 2 = 2$

$r_4 = a_4 = 3$
$b_4 = 4 \cdot 2 + 3 = \boxed{11}$

A simplified loop -free algorithm for generating permutations

The material that formed the basis of our findings for this algorithm is taken from the article of Nachum Dershowitz [4]. The algorithm presented here is a simplification of Ehrlich's loop- free version of Johnson and Trotter's algorithm. All the algorithms generate the same sequence of permutations, and they differ in the manner in which they determine which is the pair of items to be interchanged next. Note that both phases of generating the permutation from the index and finding the index from a given permutation are done exactly like we explained in the previous article - "Generation of all permutations by adjacent transpositions". First, we will present the algorithm for producing the permutations from the previous article, but we will phrase it in a different way. This formulation will help us to introduce the new algorithm and explain why its runtime complexity is better.

The Johnson and Trotter Algorithm

The following conventions and definitions will be of use.

1. The current "direction"-left or right-of an integer is symbolized by an appropriate arrow above the integer.

2. An integer is said to "face" the integers in the direction of its arrow;an integer's "neighbor" is the adjacent integer it is facing. (If it faces none, it has no neighbor.)

3. An integer can be in either an "active" or an "inactive" state; when active, it is underlined.

4. The largest active integer has a double underline and is referred to as the "cursor".

5. The cursor is considered "stuck" if it has no neighbor or if its neighbor is greater than itself.

JT-algorithm

1. initially the *n* integers are in their natural sequence, all facing left,and all but the smallest are active.

2. output the current permutation.

3. if no integer is active, terminate.

4. find the cursor i.

5. transpose the cursor with its neighbor.

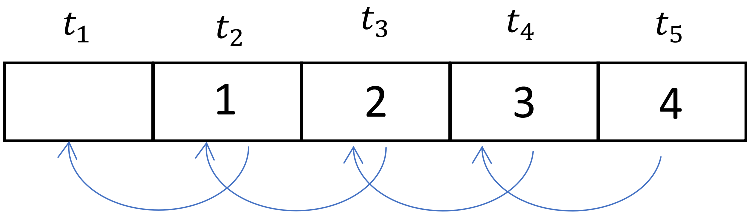
6. reactivate all integers greater than i.

7. if i is stuck, reverse its direction, and deactivate it.

8. continue with Step 2.

Improved algorithm

Note that in the previous algorithm we mark numbers as active to find the cursor. We are now interested in finding a way to find out who the cursor is without having to scan the array each time for that purpose. Note that the cursor follows a definite pattern. For n=4 the cursor takes on the values: 443443444244434443444. 4 repeats thrice until getting stuck, while 3 repeats twice; once 3 is stuck, 2 will follow 4 the next time 4 gets stuck. In general, if *i* is stuck, the next largest active integer *j, j* < *i,* will replace *i* as the cursor when again all *k, k* > *i,* are stuck. After *j* is moved, the pattern of cursors greater than *j* repeats itself. This provides the motivation for designing a data structure which will keep track of the cursor (by keeping track of all active integers), thereby eliminating the need to search for it.At the same time, the reactivation loop of Step 6 will be removed. We pay with the updating of the structure. We will use an array that represents a linked list.The T-algorithm below employs a vector T=$(t_2, t_3, \dots, t_{n+1})$within which the linked list $t_{n+1}, t_{t_{n+1}}, \dots$ is embedded.
We shall prove in the next section that this a list of the active integers in descending order, and it follow that t_{n+1} is the cursor. In all other respects the algorithm parallels the JT-algorithm. At the beginning, for n=4, our data structure looks like this:



The meaning of each value is to where we are going to point at this stage. for example, the value of t_5 is 4, therefor t_5 points to t_4.

Beside this vector, we also maintaining an array from which we are going to output the permutation in each stage. when $t_{n+1} = 1$ the algorithm terminates.

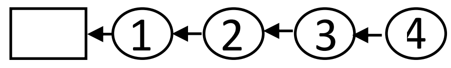
Note that for each rule in the algorithm below, if there is a reference in a stage to any t_i, then the reference is to an action needs to be made in the vector that simulates a linked list. If there is no reference to any t_i at all, it refers to an action in the array from which we print the relevant permutation each time. the initial stage of this array is

1	2	3	4

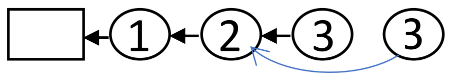
T-algorithm

```
1. initially the $n$ integers are in their natural sequence, facing left,
   and for $j = 2,3,\ldots,n+1$: $t_j = j - 1$
2. output the current permutation.
3. if $t_{n+1} < 2$, terminate.
4. Set the cursor: $i \leftarrow t_{n+1}$
5. transpose the cursor with its neighbor.
6. set: $t_{n+1} \leftarrow n$
7. if i is stuck,
   reverse its direction and set:
       a) $t_{i+1} \leftarrow t_i$
       b) $t_i \leftarrow i - 1$.
8. continue with Step 2.
```

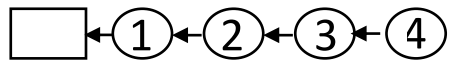
As noted earlier, the order in which the permutations are created is the same as the order in which they are formed in the adjacent transposition article. Also, the values in our auxiliary data structure do not change at each stage. To create most of the permutations out of the 24 permutations (for n = 4) it stays the same. for the sake of illustration, we will now present the steps in the algorithm in which changes are made in this auxiliary data structure.



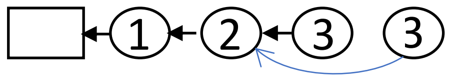
For permutation number 0 and the cursor is 4



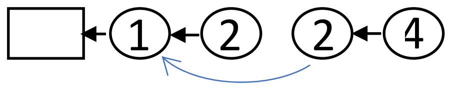
For permutation number 3 and the cursor is 3



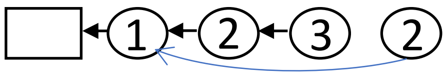
For permutation number 4 and the cursor is 4



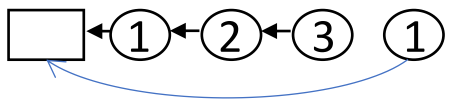
For permutation number 7 and the cursor is 3



For permutation number 8 and the cursor is 4



For permutation number 11 and the cursor is 2



For permutation number 23

The runtime complexity of the improved algorithm is n !, since we perform operations in O(1), n! times.

Validity

JT-algorithm validity has been shown in our previous article review-"generation of all permutations by adjacent transpositions". Are goal now is to explain why T-algorithm is equivalent to the JT-algorithm, whose validity has already been shown. Before showing the equivalence of the T and JT algorithms, we prove two lemmas by induction on the execution of the T-algorithm.
Lemma 1.$t_j < j \; for \; j = 2,3,\ldots,n+1$.

Lemma 2.for k=2,3,…,n+1, if j satisfies $t_k < j < k, then \; t_j = j-1$.

They are shown to be true upon initialization and that once true, they remain true after executing steps 6 and 7, the only steps that alter T. For convenience of explanation, step 6 becomes $t'_{n+1} \leftarrow n$ and step 7 is (a) $t'_{i+1} \leftarrow t_i \quad (b) t'_i \leftarrow i-1$ .Proof for the first lemma- Initilally $t_j = j-1 < j$. Hypothesizing$t_j < j$, the execution of steps 6 and 7 results in: step 6 : $t'_{n+1} = n < n+1$ step 7: $(a)\; t'_{i+1} = t_i < i < i+1$.
$(b)\; t'_i = i-1 < i$. For the proof of the second Lemma : PROOF. Initially all $t_k = k-1$ and the lemma is trivially true. Assuming that $t_k \; < j \; < \; k$ impliest $t_j = j-1$, executing Steps 6 and 7 yields: Step (6) $t'_{n+1} = n$ and the lemma holds. Step (7) (a) for j such that $t'_{i+1} = t_i < j < i$, we have, by assumption, $t'_j = t_j = j-1$, and (b) for $j = i, t'_j = j-1$. Together, (a) and (b) give: for $j, t'_{i+1} < j < i+1, t_j = j-1$. Since there is no k such that $t_k < i+1 < k$, setting t'_{i+1} has no other effect. THEOREM. The ordered set S defined by the T-algorithm, where S = $\{t_{n+1}, t_{t_{n+1}}, \ldots, t_l\}, t_{n+1} > t_{t_{n+1}} > \cdots > t_l$, and l is such that $t_l \geq 2$ but $t_{t_l} < 2$, is equal to the set A of active integers, as defined by the JP-algorithm. t_{n+1} is therefore the cursor. PROOF. Initially S= $\{t_{n+1} = n, t_n = n-1, \ldots, t_3 = 2\} = A$. Assume that at some point in the execution S = A $\neq \emptyset$ and the cursor $i = t_{n+1}$. We show that Step 6 implements the activation of all integers greater

than i. Let A'=$A \cup \{i+1, i+2, \ldots, n\}$. The preceding lemma ensures that $t_j = j-1$ for $i = t_{n+1} < j < n+1$. Setting $t'_{n+1} = n$ results in S' = $S' = \{t'_{n+1} = n, t_{t'_{(n+1)}} = t_n = n-1, t_{n-1} = n-2, \ldots, t_{i+2} = i+1, t_{i+1} = i = t_{n+1}, t_i, \ldots, t_l)$ For all j $(i < j \leq n), j \in S'$, and S'=A'. Step 7 removes i from S'. By Lemma 1, $j > t_j$ and $t_{n+1} > t_{t_{n+1}} > \cdots > t_l$. Setting $t'_{i+1} = t_i < i$ gives, in strictly descending order:

S'' = $\{t'_{n+1} = n, t_n = n-1, \ldots, t_{i+2} = i+1, t'_{i+1} = t_i, \ldots, t_l\}$= A' -{i} =A''. since i∉S'', step (7) (b) $t'_i \leftarrow i-1$ leaves S'' unaltered and the proof is now complete.

User Documentation

Implementation of all four algorithms

General Description

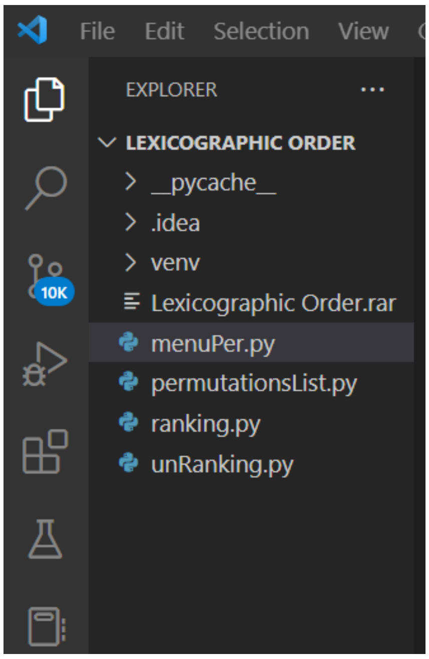
The purpose of this part is the actual implementation of all four algorithms that we explained in detail in the research part. For all four algorithms there is a user menu where the user chooses which of the three operations (list all n! permutations, Ranking, Unranking) he wants to perform. After he selects the action he wants and follows the instructions, the output will be shown to him on the screen. In addition, in the analyzing file, the user can compare the running times of all four algorithms, for each of the three operations. When he runs the operation he wants to perform, he will be presented with an output of the running time it took for the program to be for each algorithm separately.The result displayed in microseconds. The executed users here are expected to be researchers and people from the academia in the fields of computer science, mathematics, statistics and more.

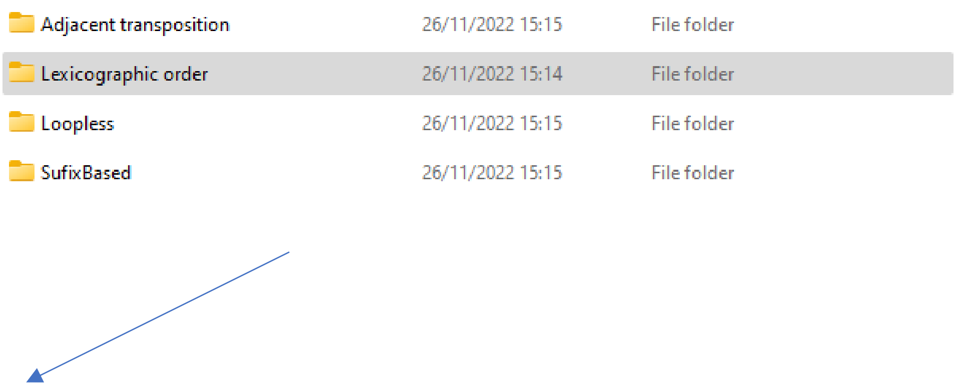
Operating instruction

All algorithms have their own folder. Choose the algorithm you want to run, open the folder in a working environment that is convenient for you, such as visual studio code and run the program. Follow the instructions shown to you in the terminal, the output will be shown to you in the terminal itself.

For example:

Step 1: choose the algorithm and load the folder to your working environment :

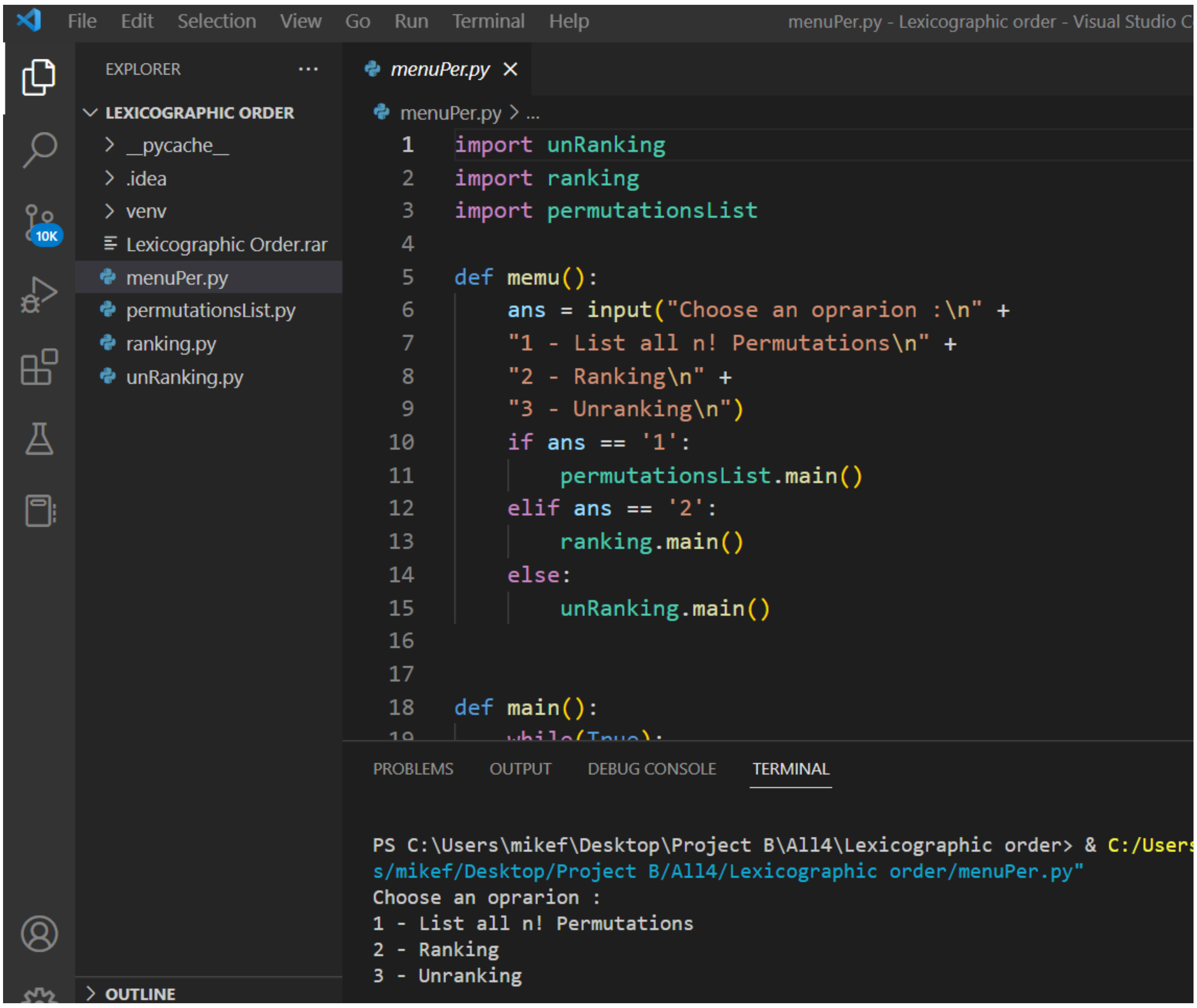




Adjacent transposition	26/11/2022 15:15	File folder
Lexicographic order	26/11/2022 15:14	File folder
Loopless	26/11/2022 15:15	File folder
SufixBased	26/11/2022 15:15	File folder

Step 2

Run the program from the menuPer.py file

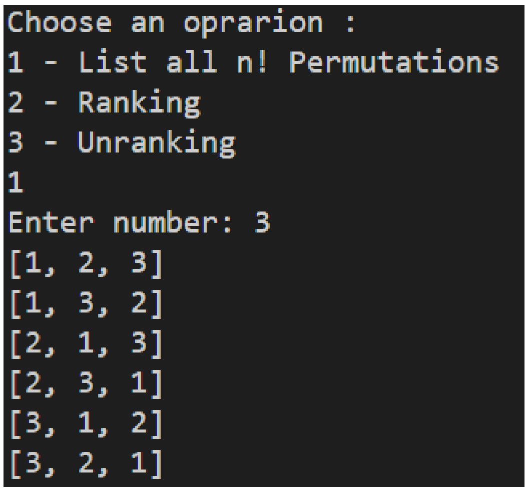


```python
import unRanking
import ranking
import permutationsList

def memu():
    ans = input("Choose an oprarion :\n" +
    "1 - List all n! Permutations\n" +
    "2 - Ranking\n" +
    "3 - Unranking\n")
    if ans == '1':
        permutationsList.main()
    elif ans == '2':
        ranking.main()
    else:
        unRanking.main()


def main():
```

```
PS C:\Users\mikef\Desktop\Project B\All4\Lexicographic order> & C:/Users
s/mikef/Desktop/Project B/All4/Lexicographic order/menuPer.py"
Choose an oprarion :
1 - List all n! Permutations
2 - Ranking
3 - Unranking
```

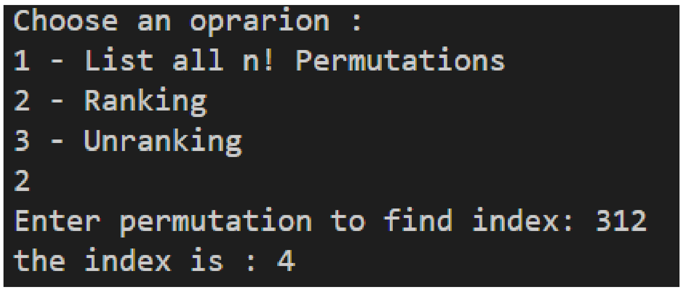
Step 3:

Choose the option you want from the menu and follow the instructions. Example for the first option of “List all n! Permutations “ :



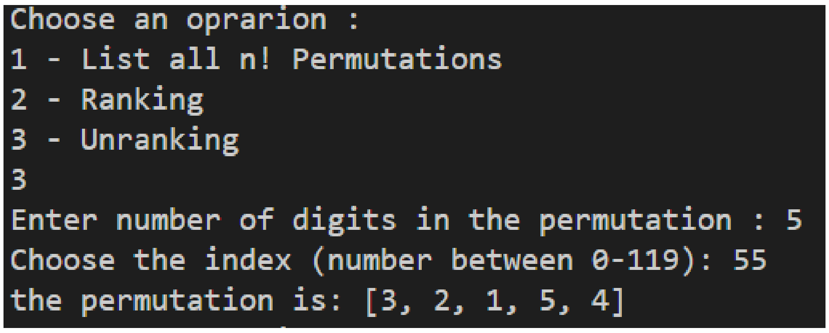
```
Choose an oprarion :
1 - List all n! Permutations
2 - Ranking
3 - Unranking
1
Enter number: 3
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Example for the second option of “Ranking “ :



```
Choose an oprarion :
1 - List all n! Permutations
2 - Ranking
3 - Unranking
2
Enter permutation to find index: 312
the index is : 4
```

Example for the third option of “Unranking “ :



```
Choose an oprarion :
1 - List all n! Permutations
2 - Ranking
3 - Unranking
3
Enter number of digits in the permutation : 5
Choose the index (number between 0-119): 55
the permutation is: [3, 2, 1, 5, 4]
```

GUI App

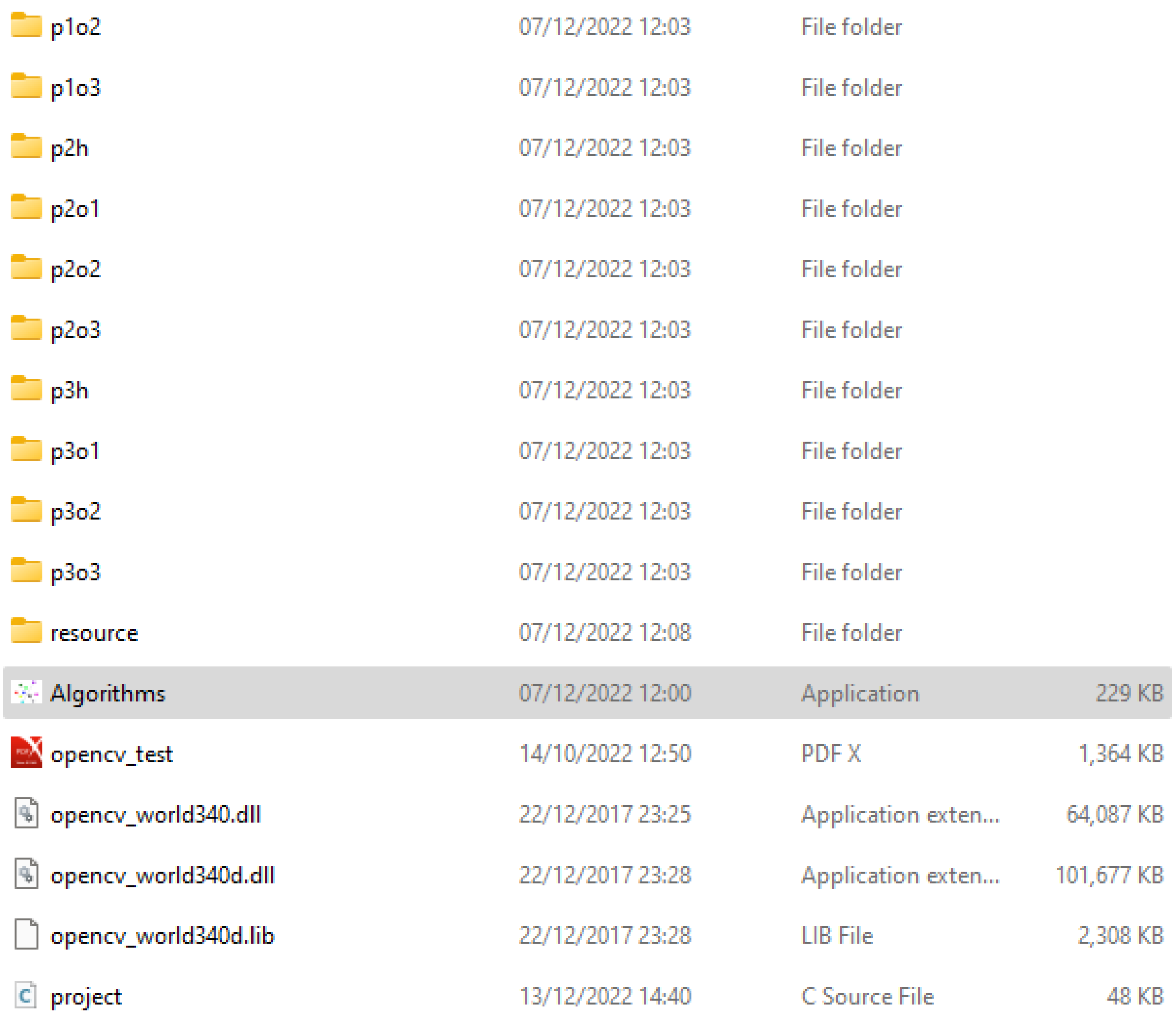
General Description

The goal of the GUI program is to allow the user to understand how all the algorithms perform their actions step by step. Each step in the algorithms is accompanied by explanations and virtualization, in a friendly way as possible. This software was built with the aim of being adapted to any user, no prior knowledge of computer science or mathematics is necessary. The user can choose which algorithm they want to start learning, and then through the ability to move step by step at their own pace, they can conveniently learn how and why each action is being conducted.

Operating instruction

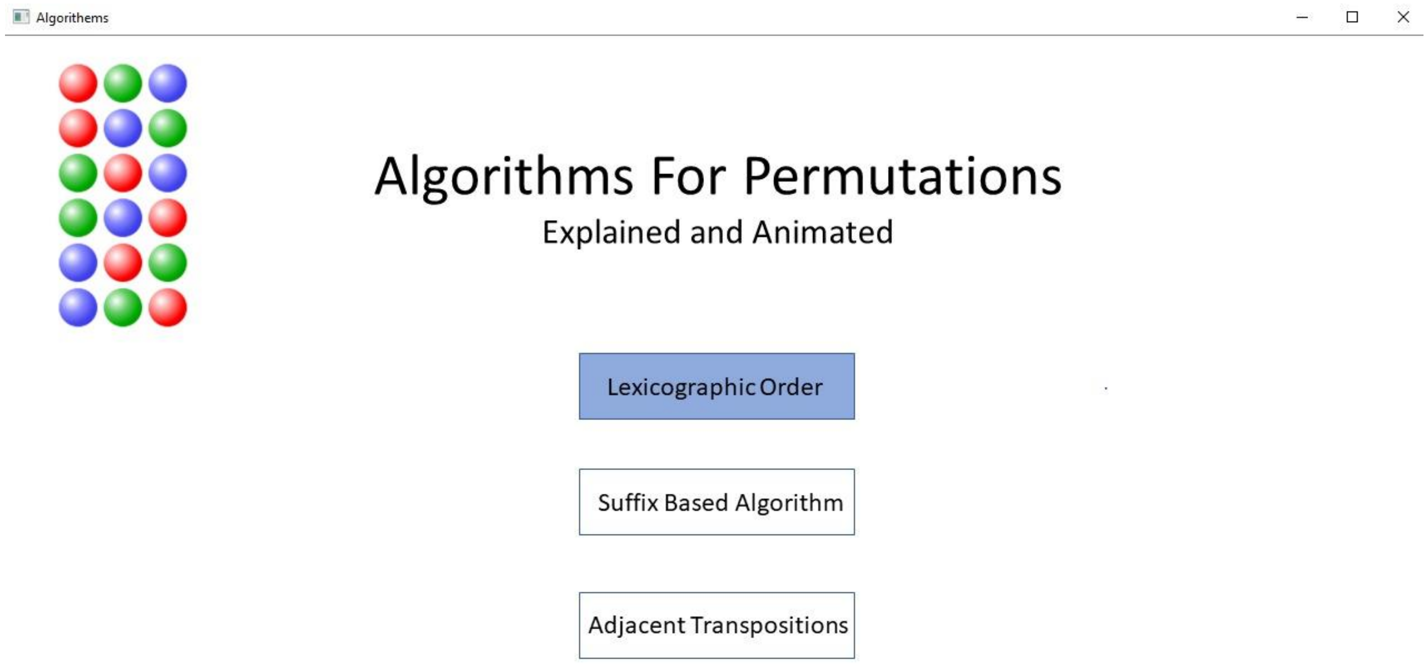
- Open the folder of the GUI, to run the program double click the exe file called "Algorithms":

Note: make sure you have installed OpenCV before trying to run the program.



p1o2	07/12/2022 12:03	File folder	
p1o3	07/12/2022 12:03	File folder	
p2h	07/12/2022 12:03	File folder	
p2o1	07/12/2022 12:03	File folder	
p2o2	07/12/2022 12:03	File folder	
p2o3	07/12/2022 12:03	File folder	
p3h	07/12/2022 12:03	File folder	
p3o1	07/12/2022 12:03	File folder	
p3o2	07/12/2022 12:03	File folder	
p3o3	07/12/2022 12:03	File folder	
resource	07/12/2022 12:08	File folder	
Algorithms	07/12/2022 12:00	Application	229 KB
opencv_test	14/10/2022 12:50	PDF X	1,364 KB
opencv_world340.dll	22/12/2017 23:25	Application exten...	64,087 KB
opencv_world340d.dll	22/12/2017 23:28	Application exten...	101,677 KB
opencv_world340d.lib	22/12/2017 23:28	LIB File	2,308 KB
project	13/12/2022 14:40	C Source File	48 KB

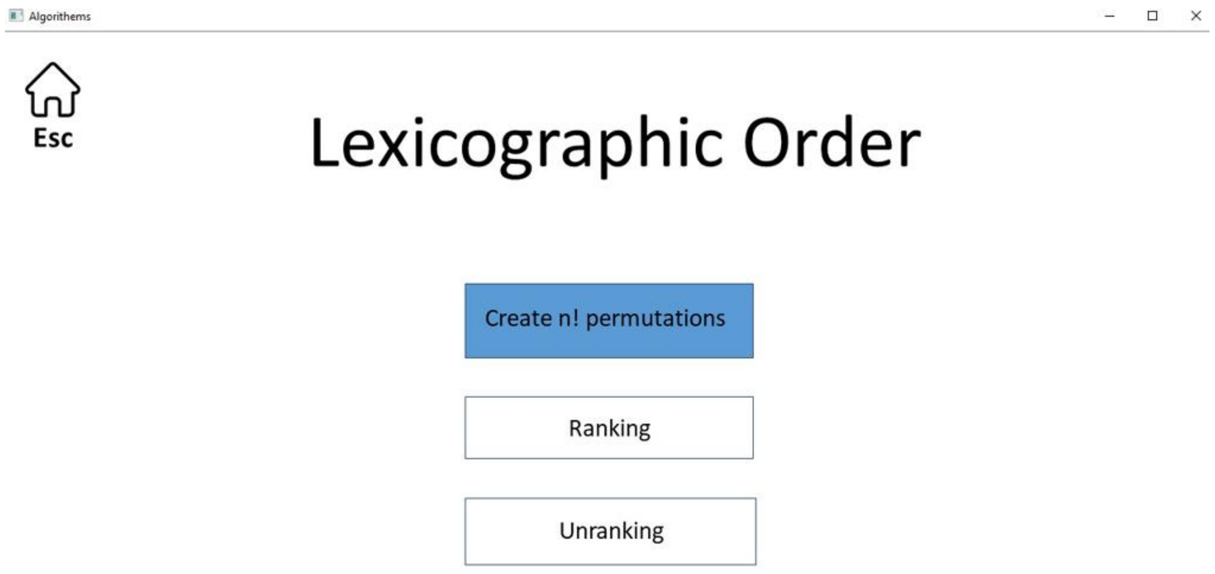
- This is the main menu:



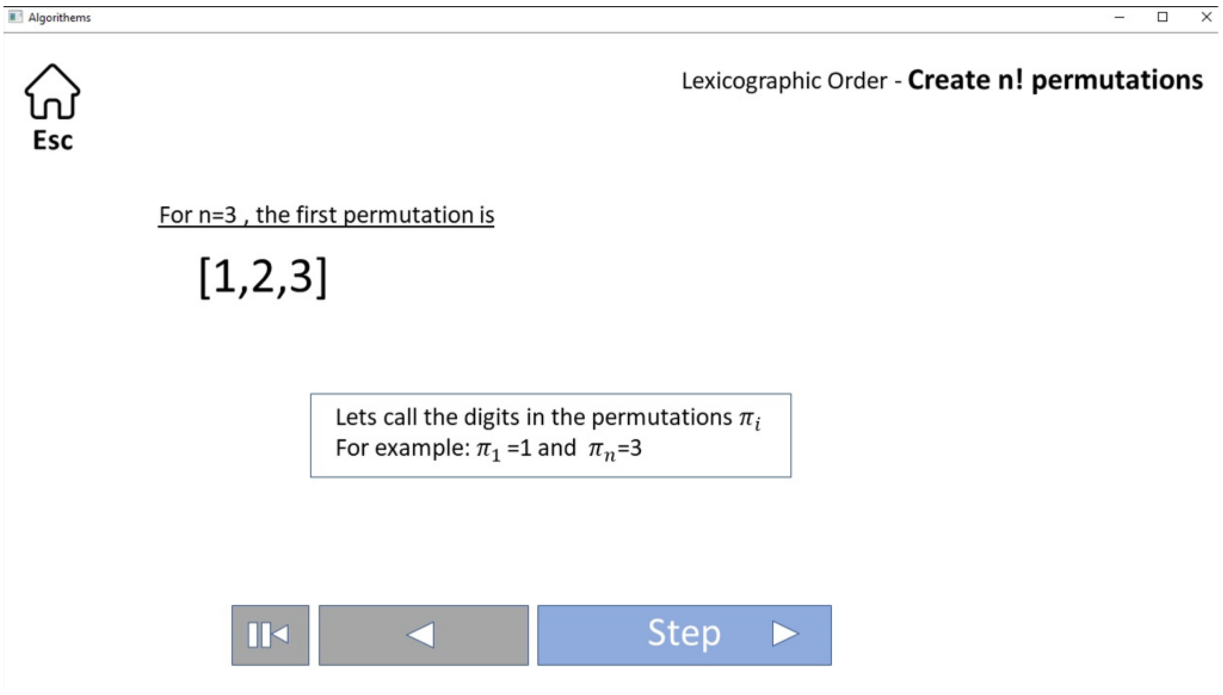
Here the user needs to choose which algorithm we wants to explore. To navigate between them, use the Arrow Keys in your keyboard. The current algorithm you are "standing on" will be emphasize in blue color. To select an algorithm you are currently standing on use the "Enter" key from your keyboard.

- Choose an operation you want to explore :

Note that from now on you can press of "Esc" from your keyboard to return to the main menu.



Example of clicking on the first option of “Create n! permutations”:


Algorithems

Esc

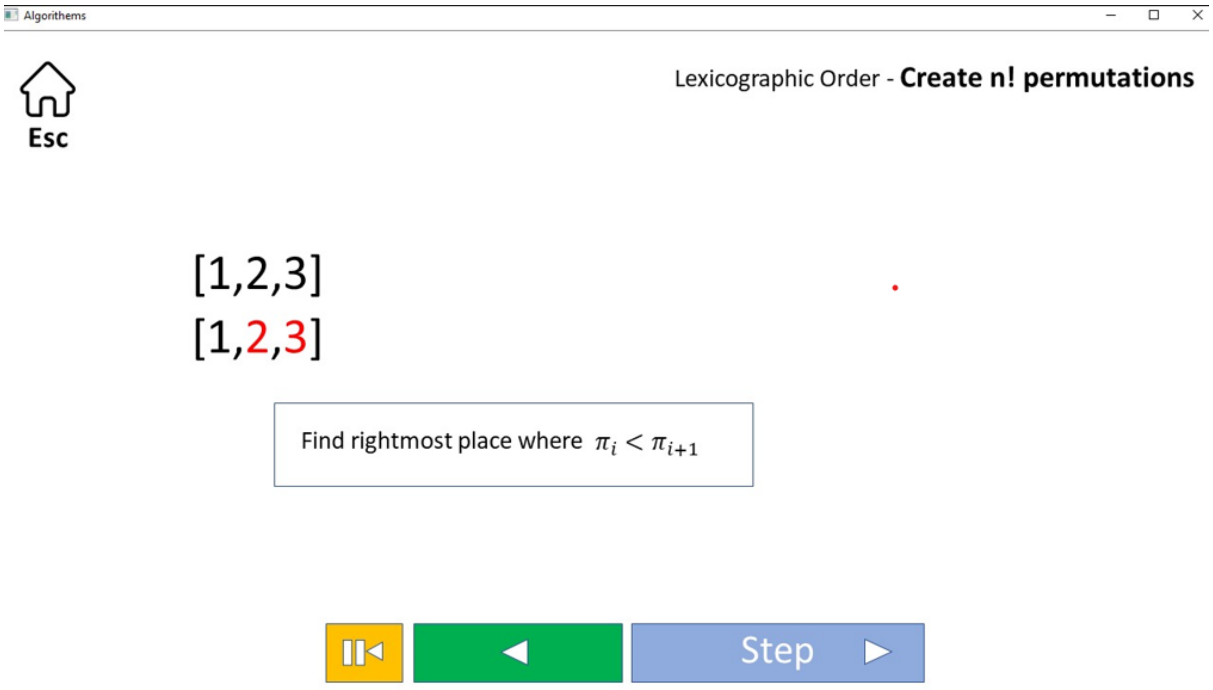
Lexicographic Order - **Create n! permutations**

For n=3 , the first permutation is

[1,2,3]

Lets call the digits in the permutations π_i
For example: π_1 =1 and π_n=3

From here to can choose to move 1 step forward by using the arrow keys and Enter key. Backwards and Reset button are not functional here because this is the first step of the algorithm.


Algorithems

Esc

Lexicographic Order - **Create n! permutations**

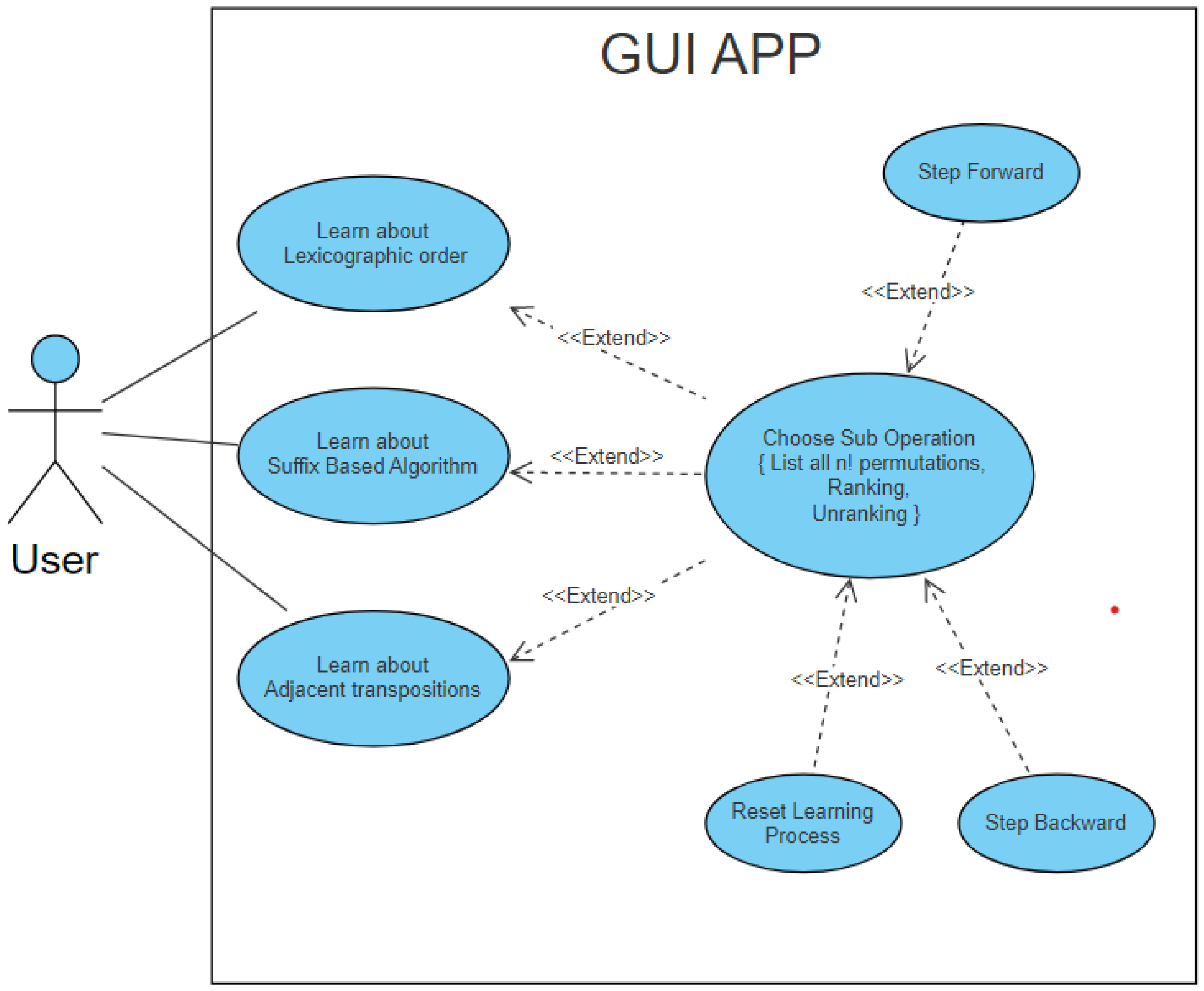
[1,2,3]
[1,2,3]

Find rightmost place where $\pi_i < \pi_{i+1}$

From here you can press “Step” button to move 1 step forward. On the green button to move 1 step backwards, or on the yellow button to Reset the explanations of the current algorithm.

Maintenance Guide

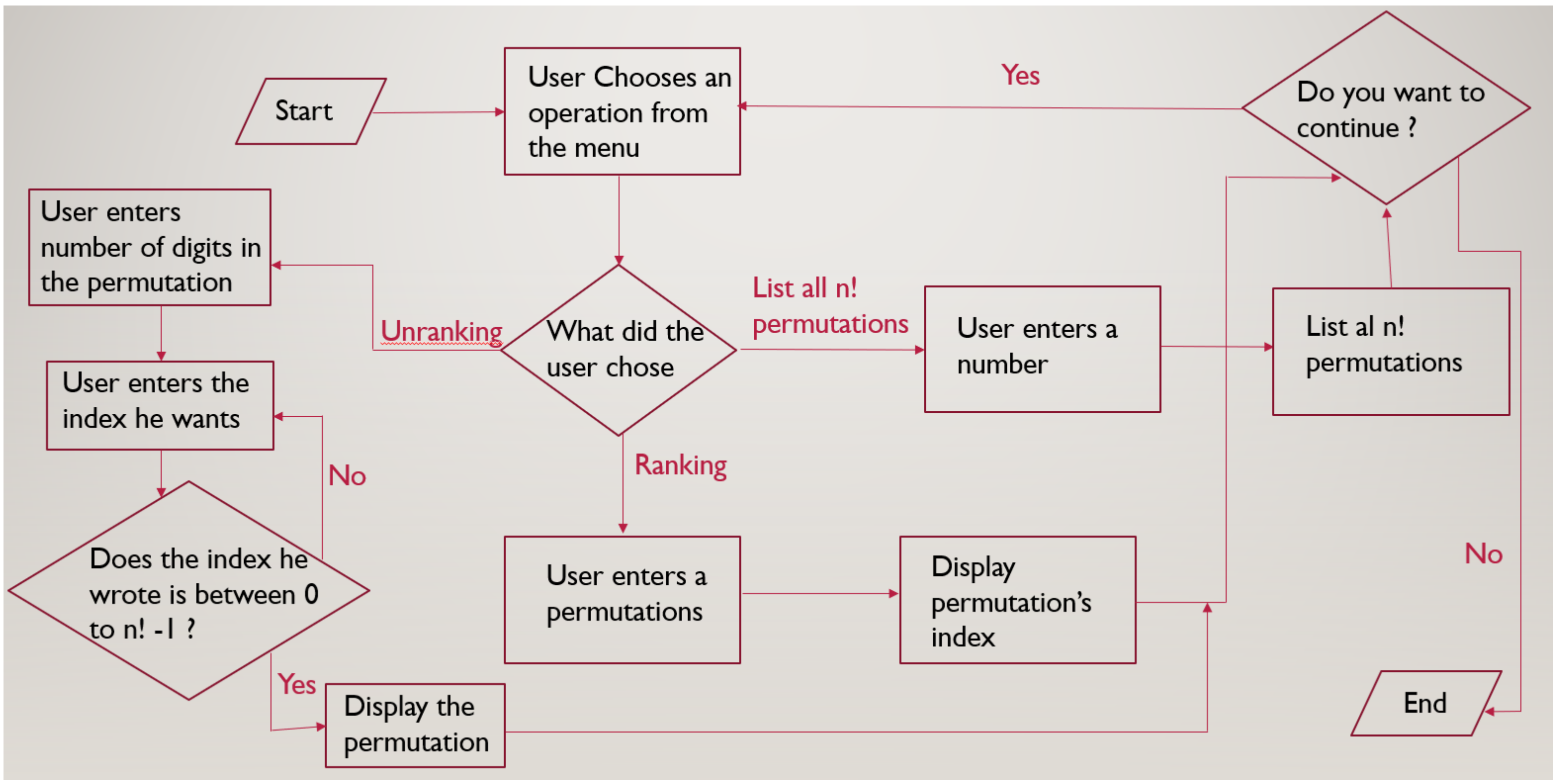
Use Case diagram explaining the GUI APP


GUI APP
User
Learn about Lexicographic order
Learn about Suffix Based Algorithm
Learn about Adjacent transpositions
Step Forward
Choose Sub Operation { List all n! permutations, Ranking, Unranking }
Reset Learning Process
Step Backward
<<Extend>>

Note: In order to run the GUI software, OpenCV must be downloaded and installed. You can download from the following link:

https://opencv.org/releases/

Flow Chart Diagram – for the implementation of all the algorithms

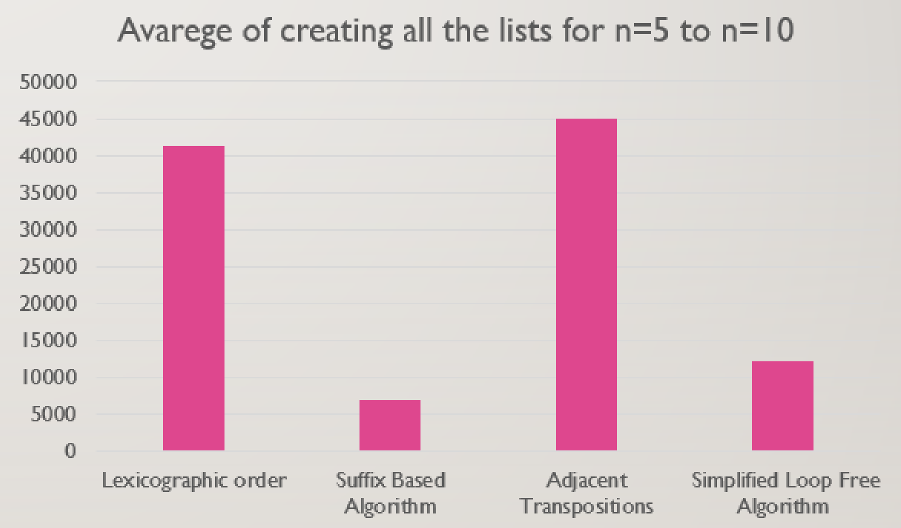

Start
User Chooses an operation from the menu
Yes
Do you want to continue ?
User enters number of digits in the permutation
Unranking
What did the user chose
List all n! permutations
User enters a number
List al n! permutations
User enters the index he wants
No
Ranking
Does the index he wrote is between 0 to n! -1 ?
User enters a permutations
Display permutation's index
No
Yes
Display the permutation
End

Final results and conclusions

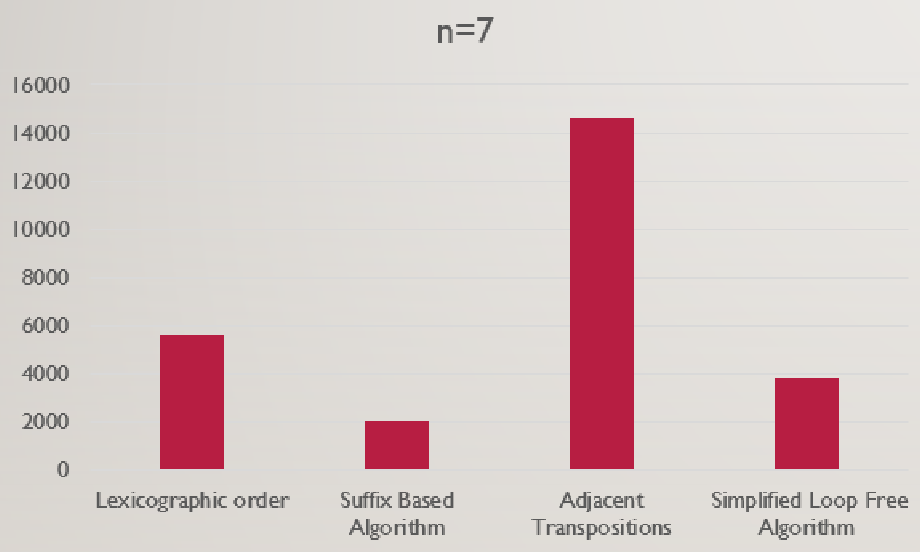
As shown in this book, in the research process we calculated the running times for all four algorithms for each of the three steps (Ranking, Unranking, List all n! permutations). The results described in this part of the project were derived from mathematical calculations of the algorithms themselves. In the programming part of the project, The GUI we created for even users who do not have a background in computer science received great reviews from many users who got to use our product, the reviews we received were that the interface is easy to use and that they were able to understand how all the algorithms work in a fun and convenient way. With the help of the second interface we built - the implementation of all four algorithms,we were able to verify what we wrote in the theoretical part regarding the running times of the algorithms. Below are some of the results:

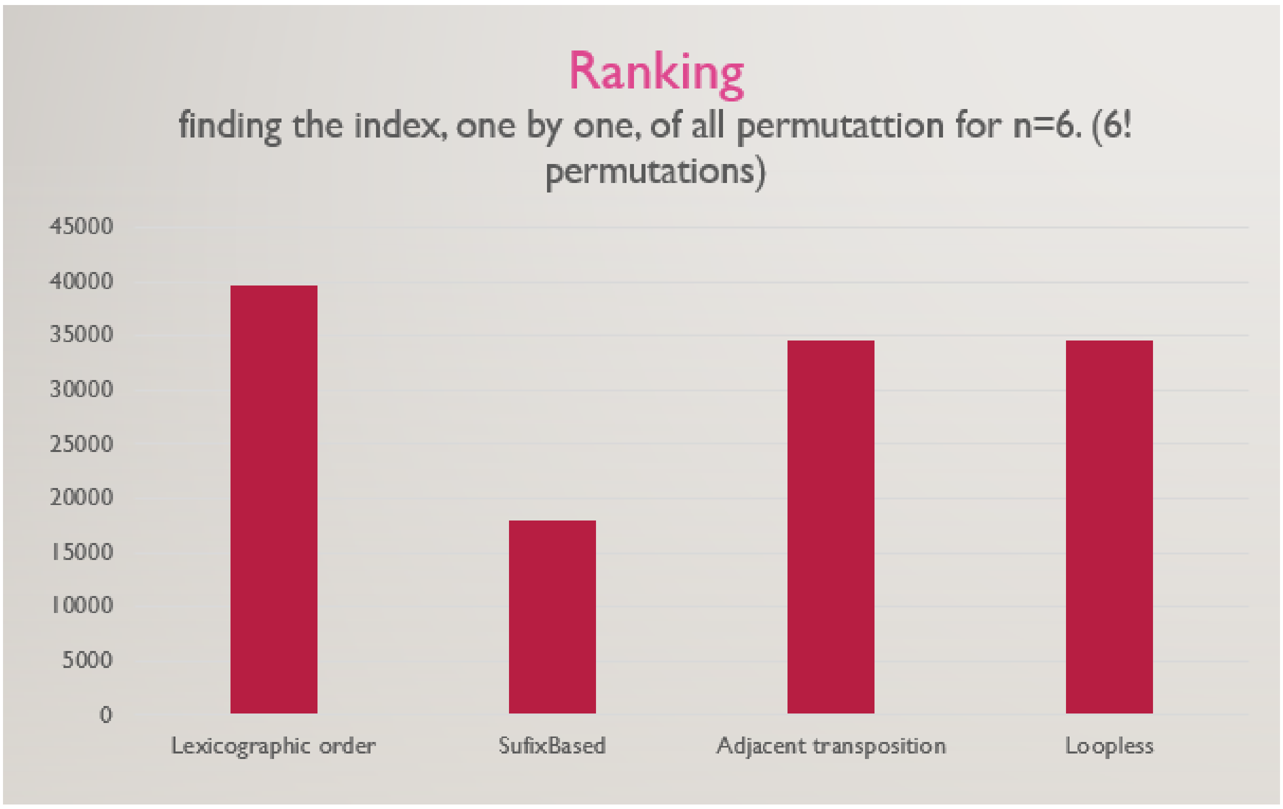
Note that time presented in microseconds.

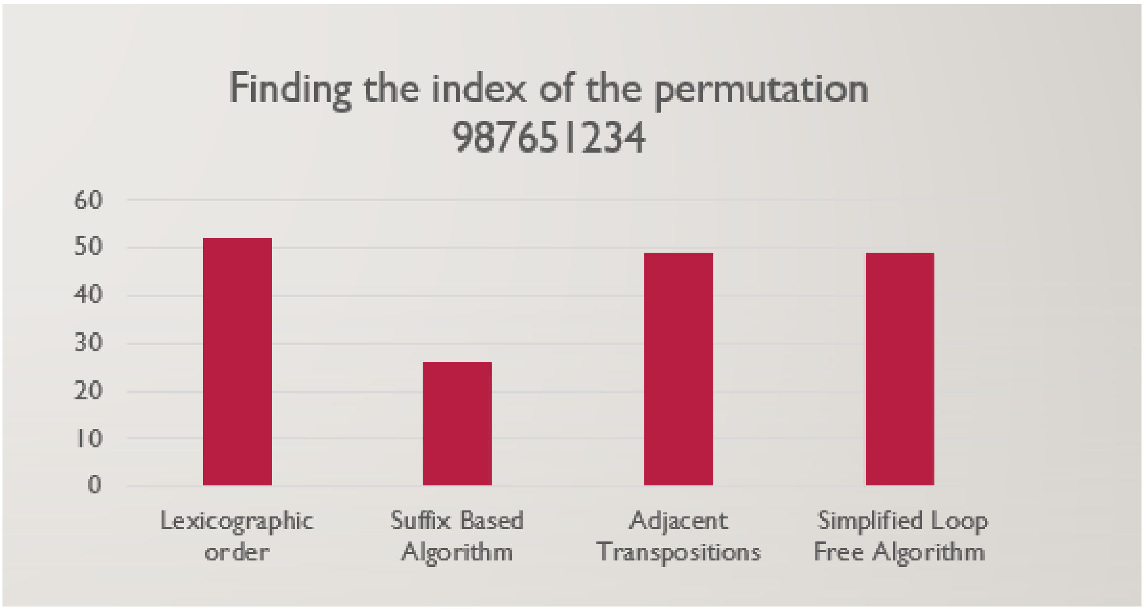
Creating all n! permutations



Creating all n! permutations

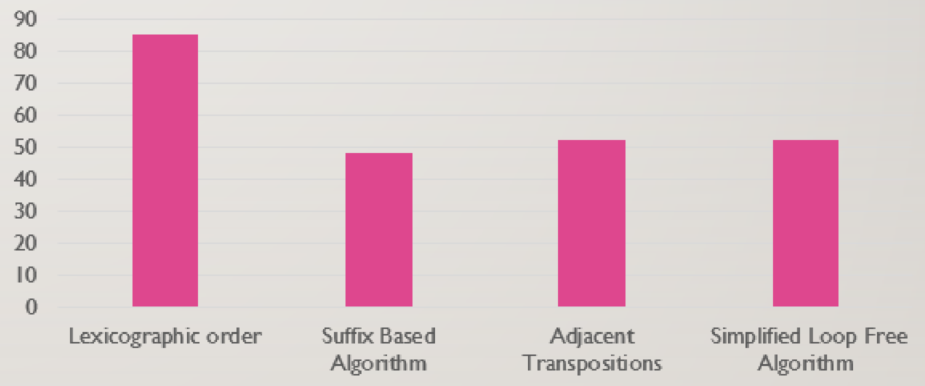



Ranking
finding the index, one by one, of all permutattion for n=6. (6! permutations)
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
Lexicographic order
SufixBased
Adjacent transposition
Loopless

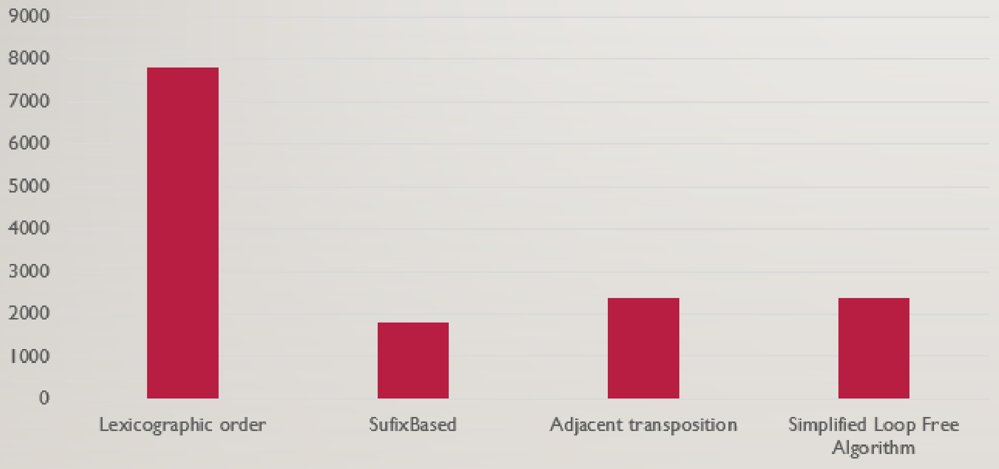

Finding the index of the permutation 987651234
60
50
40
30
20
10
0
Lexicographic order
Suffix Based Algorithm
Adjacent Transpositions
Simplified Loop Free Algorithm

Un-ranking

Finding the permutation at index 99999999999999 for n= 30



Finding all the permutations one by one for all the indexes, for n=5. (120 indexes)



References

[1] Edward M.Reingold, Jurg Nievergelt, Narsingh Deo:

"Combinatorial Algorithms Theory and Practice"

[2] Prof. Shmuel Zaks:

” new algorithm for generation of permutations.”

[3] Johnson and Trotter algorithm:

"Algorithmic combinatorics" (written by Shimon even)

[4] Nachum Dershowitz :

"A simplified loop-free algorithm for generating permutations".