# G2DLP Algorithm - Analysis, Implementation and Empirical Results

Davide Galassi

September 24, 2015

# Contents

**1 Introduction** **3**

**2 Prerequisites and Definitions** **4**

**3 Support Algorithms** **6**
3.1 Lexicographic permutation generation . . . . . 6
3.1.1 Algorithm . . . . . 7
3.1.2 Analysis . . . . . 8
3.1.3 Implementation . . . . . 8
3.1.4 Empirical results . . . . . 9
3.2 Generating all partitions of an integer . . . . . 11
3.2.1 Algorithm . . . . . 11
3.3 Generating all partitions of an integer with a fixed number of parts 12
3.3.1 Algorithm . . . . . 12
3.3.2 Analysis . . . . . 13
3.3.3 Implementation . . . . . 13
3.3.4 Empirical results . . . . . 16
3.4 Alternate Merge . . . . . 17
3.4.1 Algorithm . . . . . 17

**4 Lattice paths generation with a given number of turns** **17**
4.0.2 Definitions and theorems . . . . . 17
4.1 Trivial approach . . . . . 20
4.1.1 Algorithm . . . . . 21
4.1.2 Analysis . . . . . 21
4.1.3 Implementation . . . . . 22
4.1.4 Empirical results . . . . . 22
4.2 G2DLP . . . . . 25
4.2.1 Algorithm . . . . . 26
4.2.2 Analysis . . . . . 27
4.2.3 Implementation . . . . . 31
4.2.4 Empirical results . . . . . 32
4.3 Applications . . . . . 38
4.3.1 Cryptography . . . . . 38
4.3.2 Probability and game theory . . . . . 40

**References** **42**

**Abstract.** We discuss an algorithm to generate efficiently all the paths of a two-dimensional lattice with a given number of turns. The algorithm was first proposed by Kuo [4] and in this paper we will go further with an in depth complexity analysis, implementation and experimental results. The algorithm performances are then compared against a naive generation approach. Finally two potential applications of such kind of counting techniques are presented. The discussed algorithms are all accompained by a modern and reusable C++ implementation.

***Keywords*:** multiset, multiset permutation, lexicographic order, colexicographic order, integer partition, lattice, lattice path, turns.

# 1 Introduction

Given a $2D$ point lattice, i.e. a grid of points, in this work we will study an efficient way to generate all the paths between two arbitrary points with a given number of turns, that is the paths where the direction of the path changes a given number of times. In litterature, there are plenty of algorithms and theoretical results about lattice paths generation, but not too many that restrict the enumeration to the paths with a given number of turns.

The motivations to be interested in such counting problems touch a very different set of disciplines such as probability, statistics, cryptography, scheduling, commutative algebra, etc.

The problem of turn enumeration of lattice paths was attacked in many different ways. However there is a uniform approach which is able to handle all these problems, which is by encoding paths in terms of two-rowed arrays. Actually this is the way in which Narayana, who probably was the first to count paths with respect to their turns, used to see turn enumeration problems.

This study goes from a fairly rigorous complexity analysis to experimentation.

### Experimental notes

All the source code used for the empirical results has been implemented in C++ and all the tests were run on an Intel Core i7 CPU Q 720 @ 1.6 GHz PC.

The sources were compiled using the GNU g++ compiler without any optimization. This choice is due to the fact that compiler optimizations can heavily alter the program flow and logic from the original implementation, for example with loops unrolling, special machine instructions, branch prediction, etc. We've preferred extract our conclusions from the plain algorithms implementation that are more faithful to the theoretic descriptions. With this approach, the experimental results are also more easily comparable with other implementations that use a different programming language or for tests over different machine architectures.

The experimental results were post-processed through the *Wolfram Mathematica* ® software to produce graphs, tables and linear functions interpolations.

Some linear functions resulting from the experimental data interpolation have an intercept that is less than zero where the interpolated data is always greather than, or equal, to zero. This is mainly caused by the smaller results values disturbing the interpolation algorithm. In general, algorithm benchmarking is not very accurate for short time intervals and small quantities and this is especially true when the performance tests are executed under a multithreaded preemptive enviroment with caches, *mmu* and virtual memory (e.g. home PC). Under these conditions a bit of randomnes is tipically introduced and an algorithm execution tend to give more stable results for bigger data values and time intervals. Indeed, for *big values*, the given equations are fairly accurate.

# 2 Prerequisites and Definitions

**Multiset.** A *multiset* is the generalization of the concept of a set that, unlike a set, allows multiple instances of the elements. The *multiplicity* of an element is the number of instances of that element in a specific multiset.

For example, an infinite number of multisets exists which contain elements $a$ and $b$, varying only by multiplicity.

$\{a,b\}, \{a,a,b\} = \left\{a^2,b\right\}, \{a,a,a,b,b,b,b\} = \left\{a^3,b^4\right\}, \ldots, \{a^{m_1}b^{m_2}\}$

**Multiset permutation.** If $M$ is a finite multiset, then a multiset permutation is an ordered arrangement of elements of $M$ in which each element appears exactly as often as its multiplicity in $M$.

An anagram of a word having some repeated letters is an example of a multiset permutation.

If the multiplicities of $M$ are $m_1, \ldots, m_k$ and their sum is $n$ then the number of the multiset permutations is given by $\binom{n}{m_1 \ldots m_k}$ that is the *multinomial coefficient*.

**Lexicographic order.** Given two partially ordered sets $A$ and $B$, the lexicographical order on the cartesian product $A \times B$ is defined as $(a,b) \leq (a'b')$ if and only if $\left(a < a^{'}\right) \vee \left(a = a^{'} \wedge b \leq b^{'}\right)$. The result is a partial order.

The definition can be extended to cartesian products of arbitrary length by recursively apply the definition, i.e., by observing that $A \times B \times C = A \times (B \times C)$ .

When applied to digit permutations, lexicographic order is an increasing numerical order.

For example, the permutations of $\{1,2,3\}$ in lexicographic order are
$123, 132, 213, 231, 312, 321$.

**Colexicographic order.** Given two partially ordered sets $A$ and $B$, the colexicographical order on the cartesian product $A \times B$ is defined as $(a,b) \leq (a'b')$ if and only if $\left(b < b^{'}\right) \vee \left(b = b^{'} \wedge a \leq a^{'}\right)$. The result is a partial order.

As with colexicontraphic order, the definition can be extended to cartesian products of arbitraty length.

For example, the permutations of $\{1, 2, 3\}$ in colexicographic order are $321, 231, 312, 132, 213, 123$.

**Integer partition.** Given a non a non negative integer $n$, a *partition* can be defined as a sequence of non negative integers $a_1 \geq a_2 \geq \ldots$ such that $n = a_1 + a_2 + \ldots$.

For example, one partition of the integer 7 is $a_1 = 3, a_2 = 3, a_3 = 1, a_4 = a_5 = \ldots = 0$.

The non zero terms are called *parts* and the zero terms are usually suppressed.

Two sums that differ only for the order of their summands are considered the same partition, so the order does not matter.

For example, the integer $n = 4$ can be partitioned in five distinct ways $4, 3 + 1, 2 + 2, 2 + 1 + 1, 1 + 1 + 1 + 1$.

**Lattice.** A *point lattice* is a regularly spaced array of points. Unless otherwise specified, point lattices may be taken to refer to points in a square array, i.e. points with coordinates $(n_1, n_2, \ldots)$, where $n_1, n_2, \ldots$ are integers. Such an array is often called a grid or a mesh and is indicated as $L(n_1, n_2, \ldots)$.

Point lattices are frequently simply called lattices, which unfortunately conflicts with the same term applied to ordered sets treated in lattice theory. In this study we will use the word *lattice* to refer to a *point lattice*.

For our scopes we will consider just $2D$ lattices, $L(n_1, n_2)$, which denotes an integer rectangular lattice that has an horizontal $x$-axis and a vertical $y$-axis.

An in-depth description about lattice theory can be found in Stanley [6].

**Lattice path.** A *path* is composed of connected horizontal and vertical line segments, each passing between adjacent lattice points. A lattice path is therefore a sequence of points $P_0 \ldots P_n$ with $n \geq 0$ such that $P_i$ is a lattice point and $P_{i+1}$ is obtained by offsetting one unit East (or West) or one unit North (or South).

When the path steps are restricted to just East ($E$-steps) and North ($N$-steps) then the path length from the origin $P_0 = (0, 0)$ to a point $P_f = (a, b)$ is $a + b$. The set of all these paths is then given by all the permutations of the multiset $\left\{N^a, E^b\right\}$, that is all the combinations of a sequence of $a$ $N$-steps and $b$ $E$-steps, and there are $\binom{a+b}{a}$ different paths.

**Turning points.** A point in a path $P$ which is the end point of a North step and at the same time the starting point of an East step is called a *North-East turn* ($NE$-turn) of the path. Similarly, a point in the path $P$ which is the end point of a East step and at the same time the starting point of a North step is called an *East-North turn* ($EN$-turn for short) of the path. In this paper

the steps set is restricted to $E$-steps and $N$-steps so, for brevity, we will call $EN$-turns just $N$-turns and $NE$-turns just $E$-turns.

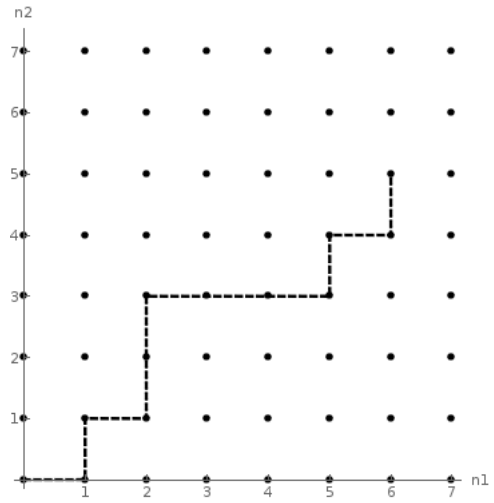


Figure 1: Turning Points

Figure 1 illustrates a possible path from the point $P_0 = (0, 0)$ to the point $P_f = (6, 5)$. The $E$-turns of that path are at $(1, 1)$,$(2, 3)$,$(5, 4)$ and the $N$-turns are at $(1, 0), (2, 1)$,$(5, 3)$,$(6, 4)$.

The algorithm subject to study has been developed with the intent to minimize the enumeration time of all the paths from the starting point $P_0$ to a final point $P_f$ with a given number of turns by omitting the generation of unwanted paths.

From a multiset perspective, a *turn* refers to two consecutive elements of a permutation that belong to two different items (in this context $N$ and $E$). In fact, if we call a *block* a sequence of consecutive elements belonging to the same multiset item, then the number of turns is equal to the number of blocks minus one.

Referring again to the Figure 1, that sample path multiset permutation is generated from the multiset $\{N^5, E^6\}$and is $ENENNEEENEN$. Here the number of turns is 7 and the number of blocks is clearly 8.

# 3 Support Algorithms

## 3.1 Lexicographic permutation generation

The algorithm goes back to Narayana Pandita in $14th$ century India, and has been frequently rediscovered ever since, indeed an in deepth description has been given by Knuth [2] as *Algorithm L*.

The idea follows a simple principle, given a sequence of $n$ elements $a_1 a_2 \dots a_n$ initially sorted so that $a_1 \leq a_2 \leq \dots \leq a_n$ this algorithm produces all permutations visiting them in lexicographic order.

Example. The permutations of $\{1, 2, 2, 3\}$, ordered lexicographically, are
$1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221$

In general, the lexicographic successor of any combinatorial pattern $a_1 \dots a_n$ is obtained with a three-step procedure:

1. Find the largest $i$ such that $a_i$ can be increased.
2. Increase $a_i$ by the smallest feasible amount.
3. Find the lexicographically least way to extend the new $a_1 ... a_i$ to a complete pattern.

### 3.1.1 Algorithm

This particular algorithm has some characteristics that make it attractive for our scopes.

It is *fast*: its time complexity to generate the next permutation is linear in the length of the array $O(N)$.

It is *compact*: it modifies the input array in-place.

It is *stateless*: no memory overhead is needed to keep track of how far along it is in enumerating the permutations. The input array itself is enough to determine the next permutation.

Its output is *sorted*: it enumerates all permutations in lexicographic order.

Its output has *no repetitions*: when the input array has duplicate elements, it correctly outputs only distinct permutations.

**Steps.**

1. [Visit] Visit the permutation $a_1 \dots a_n$.
2. [Find $i$] Set $i \leftarrow n - 1$. While $a_i \geq a_{i+1}$ decrease $i$ by 1 repeatedly. Terminate if $i = 0$. (At this point $i$ is the smallest subscript such that we have already visited all permutations beginning with $a_1 \dots a_i$. Therefore the lexicographically next permutation will increase the value of $a_i$)
3. [Increase $a_i$] Set $j \leftarrow n$. While $a_i \geq a_j$ decrease $j$ by 1 repeatedly. Then interchange $a_i \leftrightarrow a_j$. (Since $a_{i+1} \geq \dots \geq a_n$, the element $a_j$ is the smallest element greather than $a_i$ that can legitimately follow $a_1, \dots, a_{i-1}$ in a permutation. Before the interchange we had $a_{i+1} \geq \dots \geq a_{j-1} \geq a_j > a_i \geq a_{j+1} \geq \dots \geq a_n$; after the interchange we have $a_{i+1} \geq \dots \geq a_{j-1} \geq a_i > a_j \geq a_{j+1} \geq \dots \geq a_n$)
4. [Reverse $a_{i+1} \dots a_n$] Set $k \leftarrow i + 1$ and $j \leftarrow n$. Then, if $k < j$, interchange $a_k \leftrightarrow a_j$, set $k \leftarrow k + 1$, set $j \leftarrow j - 1$ and repeat until $k \geq j$. Return to step 1.

### 3.1.2 Analysis

The input string can be viewed as a multiset $M$ of cardinality $n$ and multiplicities $m_1, \ldots, m_k$.

To find all the permutations the algorithm outer loop is executed $\binom{n}{m_1 \ldots m_k} - 1$ times (the first visit comes for free).

For each outer loop iteration the steps 1 to 4 are executed, whose complexities are, in general, $O(n)$.

The worst case, that is the greater number of permutations, is given when all the elements of the input sequence are distinct. In this case, ignoring the inner steps complexities, the cost of the algorithm to visit all the permutations is clearly $O(n!)$.

### 3.1.3 Implementation

The implementation of the algorithm consists of a generic template function that, given an arbitrary sequence, generates the next permutation by following the lexicographic order [1].

If the function can determine the next higher permutation, it rearranges the elements as such and returns true. If that was not possible, because it is already at the largest possible permutation, it rearranges the elements according to the first permutation (sorted in ascending order) and returns false.

```
//! Rearranges the elements in the range [first, last) into
//! the next lexicographically greater permutation.
//!
//! @param first
//!    sequence first element iterator
//! @param last
//!    sequence last element iterator
//! @return
//!    True if the function can rearrange the object as a
//!    lexicographically greater permutation. Otherwise, the
//!    function returns false to indicate that the arrangement
//!    is not greater than the previous, but the lowest
//!    possible (sorted in ascending order).
template <typename Iterator>
bool next_permute(Iterator first, Iterator last)
{
    Iterator &a = first;
    int n = last-first;

    // Step 2 : Find i
    int i = n-2;
    while (0 <= i && a[i] >= a[i+1])
        i--;

    if (i < 0)  // Termination condition
    {
```

[1] An equivalent implementation of this algorithm is used by the ISO C++ Standard Library function *next_permutation*

```
        // rearrange to the smaller permutation
        for (int k = 0, j = n-1; k < j; k++, j--)
            std::swap(a[k], a[j]);
        return false;
    }

    // Step 3: increase a[i]
    int j = n-1;
    while (a[i] >= a[j])
        j--;
    std::swap(a[i], a[j]);

    // Step 4: reverse a[i+1] ... a[n-1]
    for (int k = i+1, j = n-1; k < j; k++, j--)
        std::swap(a[k], a[j]);

    return true;
}
```

Along with the main function, an additional function is proposed to visit each generated permutation.

```
//! Visit function pointer type alias.
template <typename Iterator>
using visit_func = void (*)(Iterator first, Iterator last);

//! Generates and visits all the sequence permutations.
//!
//! @param first
//!    Sequence first element iterator.
//! @param last
//!    Sequence last element iterator.
//! @param visit
//!    Visit function pointer.
template <typename Iterator>
void visit_permutations(Iterator first, Iterator last,
                        visit_func<Iterator> visit)
{
    do {
        // Step 1 : Visit
        visit(first, last);
    } while (next_permute(first, last));
}
```

### 3.1.4 Empirical results

The algorithm has been applied to lists with distinct elements to examine the timing of the worst case.

#### Variables

- $n$ : sequence length;
- *perms* : number of permutations;

- *time* : computation time in milliseconds.

| n | perms | time |
|---|---|---|
| 2 | 2 | 0.000738 |
| 3 | 6 | 0.001853 |
| 4 | 24 | 0.005877 |
| 5 | 120 | 0.027832 |
| 6 | 720 | 0.166443 |
| 7 | 5040 | 1.2794 |
| 8 | 40 320 | 10.2236 |
| 9 | 362 880 | 31.5605 |
| 10 | 3 628 800 | 284.302 |
| 11 | 39 916 800 | 3138.69 |
| 12 | 479 001 600 | 38072.1 |
| 13 | 6 227 020 800 | 493848. |

Table 1: Permutations results

The permutations numbers grows very fast, to allow a better view of the graph points the $x$ and $y$ axis values were logarithmically scaled.

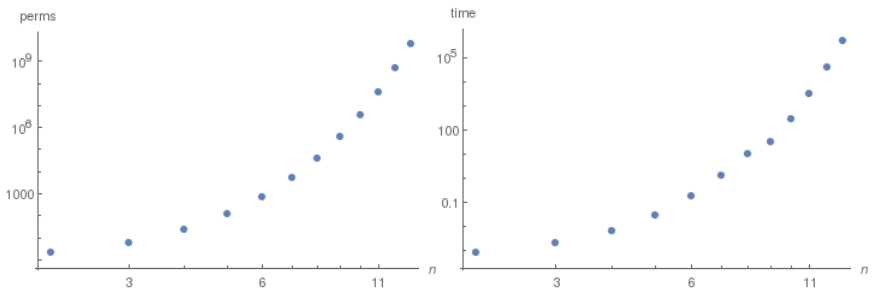


Figure 2: $perms = f(n)$, $time = g(n)$

As expected the empirical results are coherent with the theoretical ones. When all the elements are different the complexity is $O(n!)$. We can also note that the time fits well as a linear function of the number of permutations; that means that the generation complexity of each permutation can be assumed to be linear.

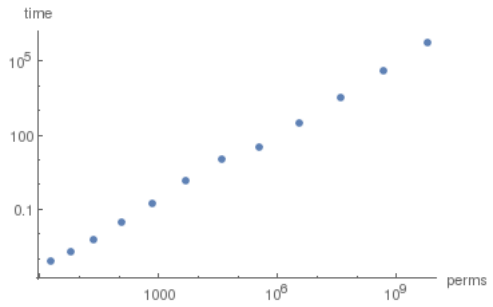


Figure 3: $time = h(perms)$

The data interpolation, when all the sequence elements are distinct, gives the following linear relation between paths and time:

$$time = 0.0000793073 \cdot perms + 5.29699$$

## 3.2 Generating all partitions of an integer

Even if this algorithm is not directly used by the *G2DLP* algorithm, it is conceptually linked with the other partitioning algorithm presented that is instead used directly, therefore, for completeness, it has been examined and implemented as well[2].

The simplest and one of the fastest way to generate all partitions is to visit them in reverse lexicographic order starting with $n$ and ending with $1 \ldots 1$ .

Example. The partitions of 8 are

$8, 71, 62, 611, 53, 521, 5111, 44, 431, 422, 4211, 41111, 332, 3311,$

$3221, 32111, 311111, 2222, 22211, 221111, 2111111, 11111111$

The algorithm was described by Knuth [1] as *Algorithm P*.

If a partition isn't all 1s, it ends with $(x+1)$ followed by zero or more 1s, for some $x > 1$; therefore the next smallest partition in lexicographic order after $[\ldots](x+1)[1 \ldots 1]$ is obtained by replacing the suffix $(x+1)[1 \ldots 1]$ by $x[\ldots x]r$ for some appropriate remainder $r \leq x$.

That is, given the partition $p_i = [\ldots](x+1)[1 \ldots 1]$, the next partition in lexicographic order will be $p_{i+1} = [\ldots]x[\ldots x]r$. If we set $k$ equal to the number of 1s in the suffix of $p_i$ plus one, then the number of $x$s of $p_{i+1}$ is given by one plus the integer quotient of $k/x$ and $r$ is given by its remainder .

Example. $p_i = 9741111111$ then $x = 3$, $k = 8$ and $p_{i+1} = 973332$.

### 3.2.1 Algorithm

This algorithm generates all partitions $a_1 \ldots a_m$ for the positive integer $n \geq 1$ with $a_1 \geq a_2 \geq \ldots \geq a_m \geq 1$ , $a_1 + a_2 + \ldots + a_m = n$, and $1 \leq m \leq n$ .

[2] An implementation is provided with the companion sources

**Steps.**

1. [Initialize] Set $a_0 \leftarrow 0$ and $m \leftarrow 1$.

2. [Store the final part] Set $a_m \leftarrow n$ and if $n = 1$ then $q \leftarrow m-1$ else $q \leftarrow m$.

3. [Visit] Visit the partition $a_1 a_2 \ldots a_m$. Then if $a_q \neq 2$ go to Step 5 .

4. [Change 2 to 1 + 1] Set $a_q \leftarrow 1$, $q \leftarrow q-1$, $m \leftarrow m+1$, $a_m \leftarrow 1$, and return to step 3.

5. [Decrease $a_q$] Terminate the algorithm if $q = 0$. Otherwise set $x \leftarrow a_q - 1$, $a_q \leftarrow x$, $n \leftarrow m - q + 1$,and $m \leftarrow q + 1$.

6. [Copy $x$ if necessary] If $n \leq x$, return to step 2. Otherwise set $a_m \leftarrow x$, $m \leftarrow m+1$, $n \leftarrow n - x$, and repeat this step.

## 3.3 Generating all partitions of an integer with a fixed number of parts

The following algorithm generates all partitions of a positive integer $n$ into a fixed number of parts $m$ in colexicographic order.

For example, when $n = 11$ and $m = 4$ the successive partitions visited are:
$8111, 7211, 6311, 5411, 6221, 5321, 4421, 4331, 5222, 4322, 3332$

The method was featured in Hindenburg's 18-th century dissertation (*Infinitinomii Dignitatum Exponentis Indeterminati*) and was deeply described by Knuth [1] as *Algorithm H.*

The basic idea is that colexicographic order goes from one partition $a_1 \ldots a_m$ to the next by finding the smallest $j$ such that $a_j$ can be increased without changing $a_{j+1} \ldots a_m$.

The new partition $a_1^{'} \ldots a_m^{'}$ will have $a_1^{'} \geq \ldots \geq a_j^{'} = a_j + 1$ and $a_1^{'} + \ldots + a_j^{'} = a_1 + \ldots + a_j$, and these conditions are achievable if and only if $a_j < a_1 - 1$. Furthermore, the smallest such partition $a_1^{'} \ldots a_m^{'}$ in colexicographic order has $a_2^{'} = \ldots = a_j^{'} = a_j + 1$.

### 3.3.1 Algorithm

This algorithm generates all integer $m$-tuples $a_1 \ldots a_m$ such that $a_1 \geq \ldots \geq a_m \geq 1$ and $a_1 + \ldots + a_m = n$, assuming that $n \geq m \geq 2$.

**Steps.**

1. [Initialize] Set $a_1 \leftarrow n - m + 1$ and $a_j \leftarrow 1$ for $1 < j \leq m$. Also set $a_{m+1} \leftarrow -1$.

2. [Visit] Visit the partition $a_1 \ldots a_m$. Then go to step 4 if $a_2 \geq a_1 - 1$.

3. [Tweak $a_1$ and $a_2$] Set $a_1 \leftarrow a_1 - 1$, $a_2 \leftarrow a_2 + 1$, and return to step 2.

4. [Find $j$] Set $j \leftarrow 3$ and $s \leftarrow a_1 + a_2 - 1$. Then, if $a_j \geq a_1 - 1$, set $s \leftarrow s + a_j$, $j \leftarrow j + 1$, and repeat until $a_j < a_1 - 1$. (Now $s = a_1 + \ldots + a_{j-1} - 1$)

5. [Increase $a_j$] Terminate if $j > m$. Otherwise set $x \leftarrow a_j + 1$, $a_j \leftarrow x$, $j \leftarrow j - 1$.

6. [Tweak $a_1 \ldots a_j$] While $j > 1$, set $a_j \leftarrow x$, $s \leftarrow s - x$, $j \leftarrow j - 1$. Finally set $a_1 \leftarrow s$ and return to step 2.

### 3.3.2 Analysis

Given two non negative integers $n$ and $m$, with $m \leq n$, the generation time complexity of the set of all the partitions of $n$ with a fixed number of parts $m$ is strictly bound to the cardinality of the resulting set. Such a number is given by the recursive function

$$p(n,m) = \begin{cases} 1 & m = 1 \\ p(n-1, m-1) + p(n-m, m) & m \leq n \\ 0 & default \end{cases}$$

A plot for this function for a fixed value of $n$ and $0 < m \leq n$ is given in Figure 6. The graph of the function allows us to speculate that the maximum number of partitions is given for $m \approx 0.2\,n$. Such a value can be taken as the algorithm worst case in terms of running time with a given value $n$.

Furthermore, analyzing the function graph for a fixed value of $m$ we can deduce that the grow is clearly exponential in function of $n$.

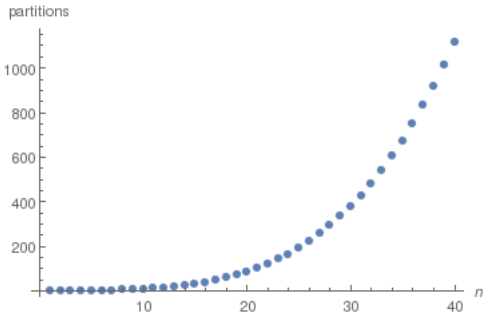


Figure 4: Partition results: $partitions = f(n, m = 5)$

An in-depth study about this topic is given by Wilf [8].

### 3.3.3 Implementation

The algorithm has been implemented as a function template that generates the next partition following the current in lexicographic order. If the function can determine the next partition, it rearranges the elements as such and returns

true. If that was not possible it rearranges the elements according to the first partition (in lexicographic order) and returns false.

Note, this implementation differs from the Kuo's choice to use directly the original implementation provided by Knuth, that is the one where the results are given by following a colexicographic order. Our motivations for the lexicographic ordering choice are clear if we consider that (later) the output of the algorithm *Partition* will be used as the input of the algorithm *Permute*, and this last one needs an input sequence that is already lexicographically ordered. Although this modification omits only a linear time operation (the inversion of each sequence), and as such does not alter the overall complexity, the solution is clearly more elegant.

```
//! Uses the input sequence [first,last) elements to generate the
//! next greater partition in lexicographic order.
//!
//! @param first
//!    Sequence first element iterator.
//! @param last
//!    Sequence last element iterator.
//! @return
//!    True if the function could generate a lexicographically
//!    greater partition. Otherwise, the function returns false
//!    to indicate that the arrangement is not greater than the
//!    previous, but the lowest possible (in ascending order).
template <typename Iterator>
bool next_partition(Iterator first, Iterator last)
{
    Iterator &a = first;
    int m = last-first;
    if (m < 2)
        return false;
    if (a[m-2] < a[m-1]-1)
    {
        // Step 3 : tweak a[0] and a[1]
        a[m-2]++;
        a[m-1]--;
    }
    else
    {
        // Step 4 : find j
        int j = m-3;
        int s = a[m-2] + a[m-1] - 1;
        while (j != -1 && a[j] >= a[m-1]-1)
        {
            s += a[j];
            j--;
        }

        if (j == -1)    // Termination condition
        {
            // Reset to the smallest partition
            first_partition(s+1, first, last);
            return false;
        }
```

```
        // Step 5 : increase a[j]
        int x = a[j] + 1;
        a[j] = x;
        j++;

        // Step 6 : tweak a[j]...a[m-1]
        while (j < m-1)
        {
            a[j] = x;
            s -= x;
            j++;
        }
        a[j] = s;
    }

    return true;
}
```

Along with the main function, two additional functions are proposed. One to generate the first integer partition (in lexicographical order) and one to sequentially visit each generated partition.

```
//! Generates the lexicographically smaller partition that can
//! be obtained from the integer n filling the sequence elements.
//! All but the last elements of the sequence are set to 1.
//! Precondition: (last-first) <= n
//!
//! @param n
//!    Integer to partition.
//! @param first
//!    Sequence first element iterator.
//! @param last
//!    Sequence last element iterator.
template <typename Integer, typename Iterator>
void first_partition(Integer n, Iterator first, Iterator last)
{
    Iterator& a = first;
    Integer m = last-first;
    // Step 1 : initialize
    while (a != last-1)
        *a++ = 1;
    *a = n - m  + 1;
}

//! Visit function pointer type alias.
template <typename Iterator>
using visit_func = void (*)(Iterator first, Iterator last);

//! Generates and visits all the sequence partitions.
//!
//! @param first
//!    Sequence first element iterator.
//! @param last
//!    Sequence last element iterator.
//! @param visit
//!    Visit function pointer.
```

```
template <typename Iterator>
void visit_partitions(Iterator first, Iterator last,
                      visit_func<Iterator> visit)
{
    do {
        // Step 2 : Visit
        visit(first, last);
    } while (next_partition(first, last));
}
```

### 3.3.4 Empirical results

In the experiment we've analyzed the partitioning time of an integer $n$ in $m$ parts, with $m \leq n$ and $n \leq 100$, using the provided implementation.

#### Variables

- $n$ : positive integer to partition;
- $m$ : number of parts;
- *partitions* : number of partitions;
- *time* : computation time in milliseconds.

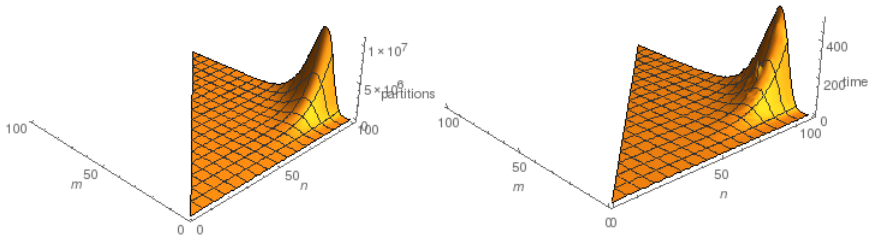


Figure 5: $partitions = f(n, m)$, $time = g(n, m)$

The generation time is clearly linearly related to the number of partitions. This means that, once again, we are able to find a formula that allows us to estimate the generation time given the number of partitions.

$$time \approx 0.0000476257 \cdot partitions + 0.439574$$

Looking at the graph we can deduce that the behaviour of both functions is very regular with respect to the number $n$. Follows the detail of how the partitions number changes for a fixed value $n$ and in function to the number of parts $m$.

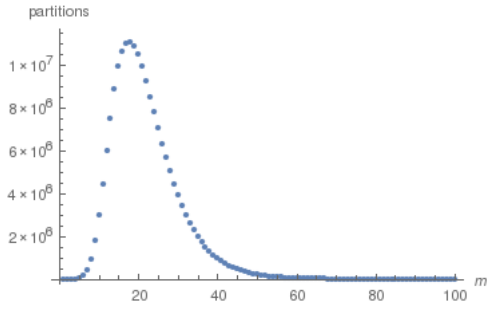


Figure 6: Partition results: $partitions = f(n = 100, m)$

From the experimental results we can deduce that, for a fixed non negative integer $n$, the maximum number of partitions is given when the number of parts is approximately $m = 0.2 \cdot n$.

## 3.4 Alternate Merge

This is a trivial algorithm used to merge two sequencies by alternating their elements. The only precondition of the algorithm is that the length of the second sequence is equal or one element less than the length of the former.

### 3.4.1 Algorithm

Given two sequencies $A \leftarrow \{a_1, a_2, \ldots, a_n\}$ and $B \leftarrow \{b_1, b_2, \ldots, b_m\}$, with $0 \leq n - m \leq 1$,

if $n = m$ then $AltMerge(A, B) = \{a_1, b_1, a_2, b_2, \ldots, a_n, b_m\}$

else if $n = m + 1$ then $AltMerge(A, B) = \{a_1, b_1, a_2, b_2, \ldots, a_{n-1}, b_m, a_n\}$

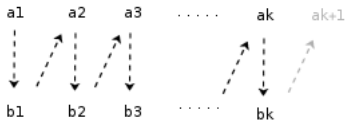


Figure 7: Alternate Merge

# 4 Lattice paths generation with a given number of turns

### 4.0.2 Definitions and theorems

In the Kuo [4] paper all the theorems are given "*as-is*" without any proof. Perhaps the choice is due to the fact that most of the proofs looks trivial,

however, we prefer to fill the gap and add an argument for each assertion.

**Definition 1.** Let $t$ denote the number of turns of a path that has $t_E$ $E$-turns and $t_N$ $N$-turns, that is, $t = t_E + t_N$.

**Lemma 1.** $|t_E - t_N| = 0$ or $|t_E - t_N| = 1$

*Proof.*

All the possible paths from one point to the other are composed by sequences of $N$-steps and $E$-steps.

If a path starts with an $E$-step then it can terminate with an $E$-step or with a $N$-step. If it terminates with an $E$-step then, inside the path, for each $N$-turn there is exactly one $E$-turn to re-establish the $E$-direction, that is $|t_E - t_N| = 0$. If it terminates with a $N$-step then, inside the path there must be an $N$-turn more, so we have $|t_E - t_N| = 1$.

If a path starts with a $N$-step the argument is similar.

□

**Lemma 2.** If $t$ is even $(2k)$, then $t_E = t_N = k$. If $t$ is odd $(2k-1)$, then $t_N = k$ and $t_E = k-1$ when the first step is eastward, or $t_E = k$ and $t_N = k-1$ when the first step is northward.

*Proof.*

Using *Lemma 1.*

If $t$ is even $(2k)$, then it should be $|t_E - t_N| = 0$, that imply $t_E = t_N = k$. This is the case that happens when the path ends with the same step as the first one.

If $t$ is odd $(2k-1)$, then, it should be $|t_E - t_N| = 1$. If the first step is eastward the last step must be northward and there is an $N$-turn more, that is $t_N = k$ and $t_E = k-1$ $(t_N - t_E = 1)$.

If the first step is northward the argument is similar and $t_E = k$ and $t_N = k-1$ $(t_E - t_N = 1)$.

□

**Lemma 3.** Given a path from $P_0 = (0,0)$ to an arbitrary end point $P_f = (n_1, n_2)$ then the path to reach $P_f^{'} = (n_1^{'}, n_2^{'}) = (n_1 + 1, n_2 + 1)$ adds at least one turn and a maximum of two turns to the previous path.

*Proof.*

The proof is given graphically.

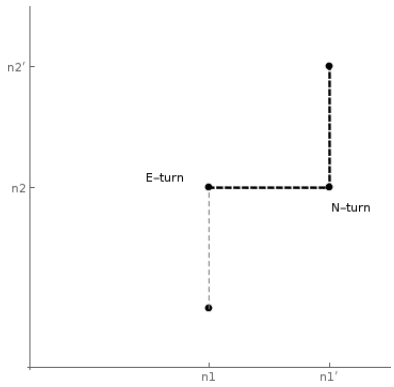


Figure 8:

In Figure 8 is shown how a maximum of two turns can be added to move from $P_f = (n_1, n_2)$ to $P_f^{'} = (n_1^{'}, n_2^{'})$. If the last step to arrive to $P$ was a northward step then we can turn one time East and one time North. If the last step to arrive to $P$ was an eastward step we follow a similar argument.

□

**Lemma 4.** For a $2D$ rectangular lattice $L(n_1, n_2)$, $n_1 > 0$ and $n_2 > 0$, assume that $n_1 \leq n_2$, the minimum number of turns of a path is 1 and the maximum number of turns of a path is $2n_1 - 1$ if $n_1 = n_2$ or $2n_1$ if $n_1 < n_2$.

*Proof.*

The minimum number of turns $t$ is trivially 1 so we concentrate with the maximum number of turns. The proof is by induction on the lattice size.

When $n_1 = n_2$.

Base case. $n_1 = n_2 = 1$ and $t = 2n_1 - 1 = 1$.

Inductive hypothesis: $n_1 = n_2$ and $t_{max} = 2n_1 - 1$.

Inductive step: $n_1^{'} = n_2^{'} = n_1 + 1$ and, for *Lemma 3*, the maximum number of turns $t^{'} = t + 2$. That is $t^{'} = t + 2 = (2n_1 - 1) + 2 = 2n_1 + 1 = 2(n_1 + 1) - 1 = 2n_1^{'} - 1$.

When $n_1 < n_2$.

Base case: $n_1 = 1$, $n_2 = k > 1$, and $t = 2n_1 = 2$.

If the path starts in the eastward direction and has an $N$-turn before $k$ then we have a maximum of two turns because an $E$-turn is required to reach the final position.

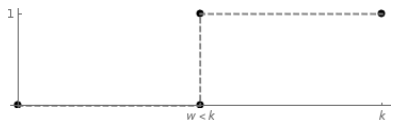


Figure 9:

Inductive hyphothesis: $n_1 < n_2$ and $t = 2n_1$.

Inductive step: $n_1^{'} = n_1 + 1$, $n_2^{'} = n_2 + 1$ and, for *Lemma 3*, the maximum number of turns $t^{'} = t + 2$. That is $t^{'} = t + 2 = 2n_1 + 2 = 2(n_1 + 1) = 2n_1^{'}$.

□

**Theorem 1.** Let $P(n_1, n_2, t)$ denote the number of paths with a given number of turns $t$, then

$$P(n_1, n_2, t) = \begin{cases} \binom{n_1-1}{k} \cdot \binom{n_2-1}{k-1} + \binom{n_2-1}{k} \cdot \binom{n_1-1}{k-1} & t = 2k \\ 2 \cdot \binom{n_1-1}{k-1} \cdot \binom{n_2-1}{k-1} & t = 2k-1 \end{cases}$$

*Proof.*

$(t = 2k)$

In the identity (1), the first term of the sum stands for the number of paths starting with an east step, and the second term stands for the number of paths starting with a north step.

For a path with even turns, by *Definition 1* and *Lemma 2*, we know that it must be composed of $k$ $E$-turns and $k$ $N$-turns. Thus, when the first step is eastward, what we need to do in the whole route is choose $k$ points from $n_1 - 1$ points and choose $k - 1$ points from $n_2 - 1$ points. When the first step is northward, what we need to do in the whole route is choose $k$ points from $n_2 - 1$ points and choose $k - 1$ points from $n_1 - 1$ points.

Since the first step is either eastward or northward, we have the first identity.

$(t = 2k - 1)$

Similary, for a path with odd turns, by *Definition 1* and *Lemma 2*, we know that it must be composed of $k$ $E$-turns and $k - 1$ $N$-turns when first step is northward, or that it must be composed of $k$ $N$-turns and $k - 1$ $E$-turns when first step is eastward, what we need to do in the whole path are choose $k - 1$ points from $n_1 - 1$ points and choose $k - 1$ points from $\mathrm{n}_2 - 1$ points. Since the first step is either northward or eastward, we have the second identity.

□

## 4.1 Trivial approach

The straightforward way to generate all the paths from a point to another with a given number of turns is by using brute force.

### 4.1.1 Algorithm

All the paths from a point $P_0 = (0, 0)$ to a point $P_f = (n_1, n_2)$ are composed of $n_1$ $E$-steps and $n_2$ $N$-steps. The algorithm strategy is reduced to:

1. Generate all the possible paths, by generating all the permutations of the multiset $\{E^{n_1}, N^{n_2}\}$.

2. Select the paths with the given number of turns.

For example. If $P_f = (3, 2)$, the paths are generated by considering all the permutations of the multiset $\{E^3, N^2\}$:
$EEENN, EENEN, EENNE, ENEEN, ENENE,$
$ENNEE, NEEEN, NEENE, NENEE, NNEEE$
And then selecting the paths with the given number of turns, e.g. 2:
$EENNE, ENNEE, NEEEN$

### 4.1.2 Analysis

Given the lattice $L(n_1, n_2)$, the algorithm overall complexity is strictly bound to the complexity of the generation of all permutations of the multiset $\{E^{n_1}, N^{n_2}\}$. The number of turns variable is not significative for the analysis, and this is because all the possible permutations are always generated and visited, at least, to perform the $n - 1$ comparisons.

**Theorem.** The time complexity of a simple multiset permutation algorithm for generating two-item $\{E^{n_1}, N^{n_2}\}$ multiset permutations according to a given number of turns is $O\left(\frac{n^{n+1.5}}{n_1^{n_1+0.5} \times n_2^{n_2+0.5}}\right)$, here $n = n_1 + n_2$; and in case $n_1 = n_2$ the time complexity is $O\left(n^{0.5} 2^{n+1}\right)$.

*Proof.*

For a two-item multiset permutation, there are totally $\binom{n_1+n_2}{n_1} = \binom{n}{n_1} = \frac{n!}{n_1! n_2!}$ permutations. Even if we ignore the generation complexity of each permutation, each one of these needs $n - 1$ comparisons to calculate how many turns it has (eg. this can be done in the Visit Step of the Permute Algorithm).

It is well known that the factorial function has exponential growth. By using Stirling approximation, $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, we have:

$$\frac{n!}{n_1! n_2!} = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\sqrt{2\pi n_1}\left(\frac{n_1}{e}\right)^{n_1}\sqrt{2\pi n_2}\left(\frac{n_2}{e}\right)^{n_2}} = \sqrt{\frac{1}{2\pi}} \frac{n^{n+0.5}}{n_1^{n_1+0.5} + n_2^{n_2+0.5}}$$

then after multiplying the term by $n-1$ we have $O\left(\frac{n^{n+1.5}}{n_1^{n_1+0.5} \times n_2^{n_2+0.5}}\right)$. Moreover, in the case $n_1 = n_2$ the time complexity is $O\left(n^{0.5} 2^{n+1}\right)$. In other words, the running time is exponential on the problem size $n$.

□

### 4.1.3 Implementation

The implementation directly uses the *Permute* algorithm with a *Visit* step implemented to perform the $n-1$ comparisons to check if the current permutation has the required number of turns.

```
template <typename Iterator>
void visit(const Iterator first, const Iterator last)
{
    int turns = 0;
    for (Iterator i = first; i != last-1; i++)
    {
        if (*i != *(i+1))
            turns++;
    }
    if (turns == required_turns)
        paths++;
    visits++;
}
```

### 4.1.4 Empirical results

**Variables**

- *n1* : lattice destination point $x$-coordinate;
- *n2* : lattice destination point $y$-coordinate;
- *turns* : number of turns;
- *paths* : number of paths with the given number of turns;
- *visits* : number of generated permutations; this is equal to the number of calls to the *Visit Step*;
- *time* : computation time in milliseconds;

**Experiment 1.** $2 \leq n_1 \leq 15$, $n_2 = n_1$

| n1 | n2 | turns | paths | visits | time |
|---|---|---|---|---|---|
| 2 | 2 | 2 | 2 | 6 | 0.004969 |
| 3 | 3 | 3 | 8 | 20 | 0.021282 |
| 4 | 4 | 4 | 18 | 70 | 0.090375 |
| 5 | 5 | 5 | 72 | 252 | 0.390691 |
| 6 | 6 | 6 | 200 | 924 | 1.71823 |
| 7 | 7 | 7 | 800 | 3432 | 7.17305 |
| 8 | 8 | 8 | 2450 | 12870 | 18.6175 |
| 9 | 9 | 9 | 9800 | 48620 | 43.3554 |
| 10 | 10 | 10 | 31752 | 184756 | 176.945 |
| 11 | 11 | 11 | 127008 | 705432 | 739.125 |
| 12 | 12 | 12 | 426888 | 2704156 | 3036.63 |
| 13 | 13 | 13 | 1707552 | 10400600 | 12550.7 |
| 14 | 14 | 14 | 5889312 | 40116600 | 51858.6 |
| 15 | 15 | 15 | 23557248 | 155117520 | 213612. |

Table 2: Trivial approach results

The permutations numbers grows very fast, to allow a better view of the graph points the $x$ and $y$ axis values were logarithmically scaled.

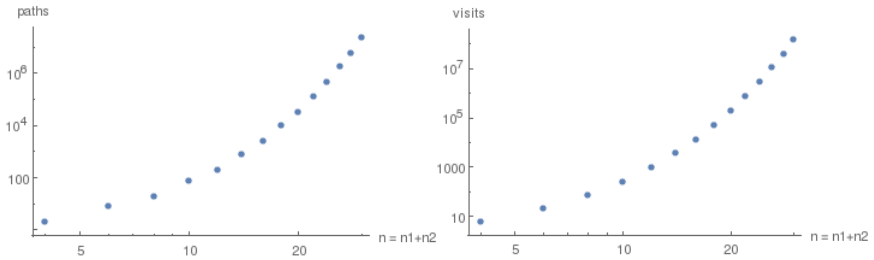


Figure 10: Permute Results: $paths = f(n_1 + n_2)$, $visits = g(n_1 + n_2)$, $n_1 = n_2$

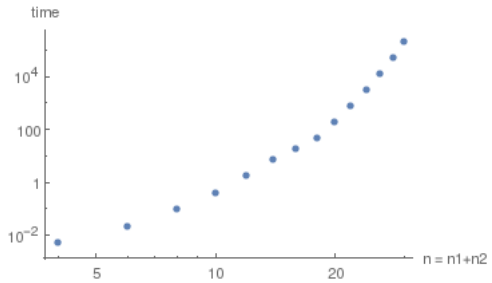


Figure 11: Permute Results: $time = f(n_1 + n_2)$, $n_1 = n_2$

The time is approximately a linear function of the number of visits and paths. That is, given $A$, $B$, $C$, $D$ real numbers

$$time \approx A \cdot paths + B$$
$$time \approx C \cdot visits + D$$

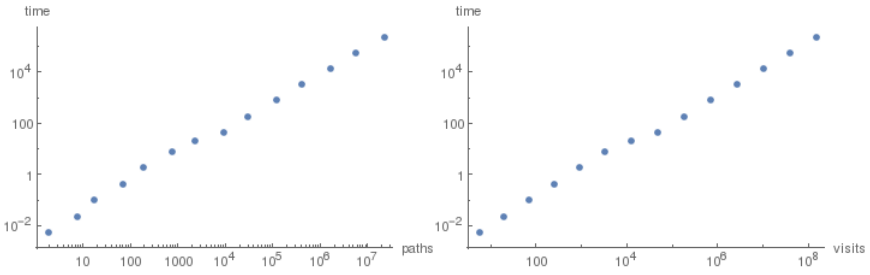


Figure 12: $time = f(paths)$, $time = g(visits)$

With the above experimental results, we've found the following best fitting linear equations:

$$time \approx 0.00906556 \cdot paths - 415.267$$
$$time \approx 0.00137428 \cdot visits - 398.835$$

This also means that the visits and the paths found are linearly related. And in this case the equation is

$$paths = 0.15159 \cdot visits + 1860.29$$

When $n_1 = n_2 = turns$, approximatelly only the 15% of our visits, and consequently of the computation time, is really usefull for the final result.

**Experiment 2.** $2 \leq n_1 \leq 15$, $2 \leq n_2 \leq 15$

The results table is not reported here for space reasons, anyway from the graph plot we can clearly see the exponential growth bound to the value of $n_1$ and $n_2$.

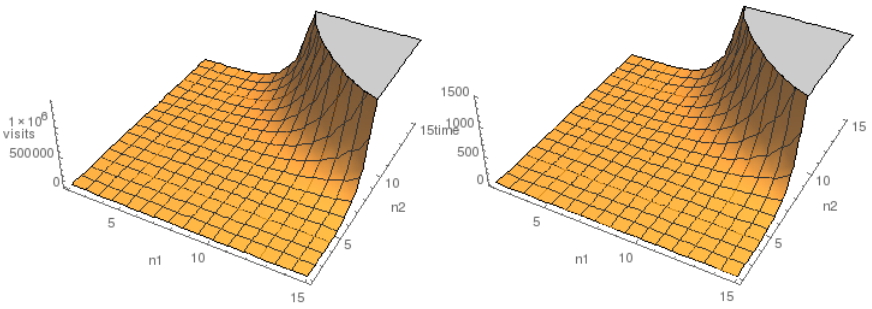


Figure 13: $visits = f(n_1, n_2)$, $time = g(n_1, n_2)$

Again, from the experiments the time is clearly a linear function of the number of visits (and paths).

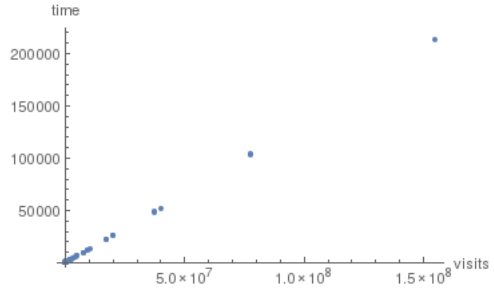


Figure 14: $time = h(visits)$

By using the same best fitting technique used by the $n_1 = n_2$ case, we reach to the following linear equation:

$$time \approx 0.00134622 \cdot visits - 166.451$$

The result can be assumed to be compatible with the previous interpolation.

## 4.2 G2DLP

The *G2DLP* algorithm has been proposed by Kuo [4] and is the argument at the center of this paper. In his article Kuo asserts that the algorithm is linear in time. Unfortunately, it is not very clear on what is linear and in function of what. In this discussion we will show that, if the dependent variable is the algorithm running time, then the only linearly related variable is the number of paths. Moreover, given a *2D* lattice of arbitrary size, we will extrapolate a running time value approximation that gives us an idea of the problem complexity.

The experimental results allows a direct comparison against the trivial approach, and this will clearly shows how the algorithm significantly reduces the

time complexity in the generation of all the paths from a point to another with a given number of turns, even in the worst case.

The algorithm borrows the *Permute*, *Partition* and *AltMerge* algorithms.

### 4.2.1 Algorithm

To construct all the paths from the point $P_0 = (0, 0)$ to the point $P_f = (n_1, n_2)$ with a given number of turns, the strategy is to divide the $x$ and $y$ axis into blocks (or parts) of $E$-steps and $N$-steps using the following rules:

- [*Partition*] Generate every $n_1$ and $n_2$ partition
  - If the number of turns is even ($t = 2k$)
    1. When the first step is eastward there will be $k$ $E$-turns, $k$ $N$-turns, $k + 1$ $E$-blocks and $k$ $N$-blocks. Partition $n_1$ into $k + 1$ parts, $parts_A$, and $n_2$ in $k$ parts, $parts_B$.
    2. When the first step is northward there will be $k$ $E$-turns, $k$ $N$-turns, $k$ $E$-blocks and $k+1$ $N$-blocks. Partition $n_2$ in $k+1$ parts, $parts_C$, and $n_1$ in $k$ parts, $parts_D$.
  - If the number of turns is odd ($t = 2k - 1$)
    1. When the first step is eastward there will be $k-1$ $E$-turns, $k$ $N$-turns, $k$ $E$-blocks and $k$ $N$-blocks. Partition both $n_1$ and $n_2$ in $k$ parts, $parts_A$ and $parts_B$.
    2. When the first step is northward there will be $k$ $E$-turns and $k - 1$ $N$-turns, $k$ $E$-blocks and $k$ $N$-blocks. Partition both $n_1$ and $n_2$ in $k$ parts, $parts_C$ and $parts_D$.
- [*Permute*] For each $X$ in $\{A, B, C, D\}$, generate all the permutations of $parts_X$ and store the results in $perms_X$ list.
- [*Merge*] Alternate merge every element of $perms_A$ with every element of $perms_B$ and every element of $perms_C$ with every element of $perms_D$ to get all the possible paths from $P_0$ to $P_f$ with the given number of turns.

**Example.** Generation of the set of all paths from the point $P_0 = (0, 0)$ to $P_f = (n_1 = 4, n_2 = 3)$ with 4 (even) turns.

Generate all the paths starting with an $E$-step.

$parts_A = Partition(n_1 = 4, k = 3) = \{112\}$

$perms_A = Permute(parts_A) = \{112, 121, 211\}$

$steps_A = ToSteps(perms_A, E) = \{\langle E, E, EE\rangle, \langle E, EE, E\rangle, \langle EE, E, E\rangle\}$

$parts_B = Partition(n_2 = 3, k = 2) = \{12\}$

$perms_B = Permute(parts_B) = \{12, 21\}$

$steps_B = ToSteps(perms_B, N) = \{\langle N, NN\rangle, \langle NN, N\rangle\}$

$paths_E = AltMerge(steps_A, steps_B) =$

$\{\langle ENENNEE\rangle, \langle ENNENEE\rangle, \langle ENEENNE\rangle\}$

$$\{\langle ENNEENE\rangle, \langle EENENNE\rangle, \langle EENNENE\rangle\}$$

Generate all the paths starting with an $N$-step.

$parts_C = Partition(n_2 = 3, k = 3) = \{111\}$

$perms_C = Permute(parts_C) = \{111\}$

$steps_C = ToSteps(perms_C, N) = \{\langle N, N, N\rangle\}$

$parts_D = Partition(n_1 = 4, k = 2) = \{13, 22\}$

$perm_D = Permute(parts_D) = \{13, 31, 22\}$

$steps_D = ToSteps(perm_D, E) = \{\langle E, EEE\rangle, \langle EEE, E\rangle, \langle EE, EE\rangle\}$

$paths_N = AltMerge(steps_C, steps_D) =$

$$\{\langle NENEEE\rangle, \langle NEEENE\rangle, \langle NEENEE\rangle\}$$

Add the paths to obtain all the possible paths with 4 turns from $P_0 = (0, 0)$ to $P_f = (4, 3)$.

$paths = paths_E \cup paths_N$

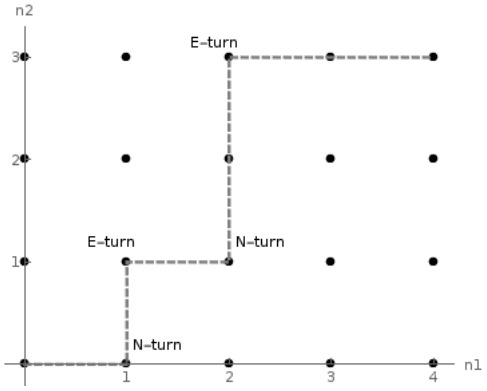


Figure 15:

In Figure 15 one path from the above example, in particular the path$\langle ENENNEE\rangle$.

### 4.2.2 Analysis

Theoretically, once we found the sets $perms_{A,B,C,D}$ the problem can be declared as solved, but in practice we want to finish the paths generation process. This is where the algorithm bottleneck is, we need to compute the cartesian products $perms_A \times perms_B$ and $perms_C \times perms_D$ and this operation complexity is clearly bound to the sets number of elements, and this number is linked with the number of turns.

Theorem 1 states that given a lattice $L(n_1, n_2)$, the number of paths with a given number of turns $t$, $P(n_1, n_2, t)$, is $P(n_1, n_2, 2k) = \binom{n_1-1}{k} \cdot \binom{n_2-1}{k-1} + \binom{n_2-1}{k} \cdot \binom{n_1-1}{k-1}$ when $t$ is even and $P(n_1, n_2, 2k-1) = 2 \cdot \binom{n_1-1}{k-1} \cdot \binom{n_2-1}{k-1}$, when $t$ is odd.

Suppose that the number of turns is even $(2k)$. When the first step is eastward the number of different $E$-steps permutations, $|perms_A|$, is $\binom{n_1-1}{k}$ and the number of different $N$-steps permutations, $|perms_B|$, is $\binom{n_2-1}{k-1}$. When the

first step is northward the number of different $E$-steps permutations, $|perms_C|$, is $\binom{n_1-1}{k-1}$ and the number of different $N$-steps permutations, $|perms_D|$, is $\binom{n_2-1}{k}$.

Suppose that the number of turns is odd $(2k-1)$. No matter if the first step is eastward or northward, the number of different $E$-steps permutations, $|perms_A|$ and $|perms_D|$, is $\binom{n_1-1}{k-1}$ and the number of different $N$-steps permutations, $|perms_B|$ and $|perms_C|$, is $\binom{n_2-1}{k-1}$.

Using the *Stirling* approximation, $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, we can be confident that, for a fixed a number of turns, the Theorem 1 formula generally grows exponentially with respect to $n_1$ and $n_2$, with the following few borderline cases exception:

- $turns = 2k - 1 = 1$

  The number of paths is constant and equal to 2.

  $P(n_1, n_2, t = 2k - 1 = 1) = 2 \cdot \binom{n_1-1}{0} \cdot \binom{n_2-1}{0} = 2$

- $turns = 2k = 2$

  The number of paths is a linear function of $n_1$ and $n_2$.

  $P(n_1, n_2, t = 2k) = \binom{n_1-1}{1} \cdot \binom{n_2-1}{0} + \binom{n_2-1}{1} \cdot \binom{n_1-1}{0} = n_1 + n_2 - 2$

- $turns = 2k - 1 = 2n_1 - 1,\ n_1 = n_2$

  The number of paths is constant and equal to 2.

  $P(n_1, n_2, t = 2k - 1 = 2n_1 - 1) = 2 \cdot \binom{n_1-1}{n_1-1} \cdot \binom{n_2-1}{n_2-1} = 2$

- $turns = 2k = 2n_1 - 2,\ n_1 = n_2$

  The number of paths is a linear function of $n_1$ and $n_2$.

  $P(n_1, n_2, t = 2k = 2n_1 - 2) = \binom{n_1-1}{n_1-1} \cdot \binom{n_2-1}{n_2-2} + \binom{n_2-1}{n_2-1} \cdot \binom{n_1-1}{n_1-1} = n_1 + n_2 - 2$

It is worth noting that the exponential complexity of such a problem is not linked with the algorithm but is strictly bound to the problem solution, that is, if the number of paths with a given number of turns are an exponential function of the lattice size then the algorithm that generates such paths should be exponential.

**Worst case.** Given a lattice $L(n_1, n_2)$, the problem worst case, that is the one that generates the bigger number of paths, is obtained by finding the maximum for the Theorem 1 formula in function of the number of turns. The exact problem solution can be reconducted to the maximization of a function of the form $h(k) = \binom{n_1}{k} \cdot \binom{n_2}{k}$, for two fixed non negative integers $n_1$ and $n_2$ and a non negative integer $k$, with $0 \leq k \leq min(n_1, n_2)$.

That value can be proven to be $k = \left\lceil \frac{n_1 n_2 - 1}{n_1 + n_2 + 2} \right\rceil$

*Proof.*

We have that $k(k) < h(k+1)$ if

$$\binom{n_1}{k} \cdot \binom{n_2}{k} < \binom{n_1}{k+1} \cdot \binom{n_2}{k+1}$$

That is if

$$\frac{n_1!}{k!(n_1-k)!}\frac{n_2!}{k!(n_2-k)!} < \frac{n_1!}{(k+1)!(n_1-k-1)!}\frac{n_2!}{(k+1)!(n_2-k-1)!}$$

Equivalent to

$$(k+1)!(k+1)!(n_1-k-1)!(n_2-k-1)! < k!k!(n_1-k)!(n_2-k)!$$

And simplifying

$$(k+1)^2 < (n_1-k)(n_2-k)$$

That is true for

$$k < \frac{n_1 n_2 - 1}{n_1+n_2+2}$$

So the maximum is reached for the value immediately following that bound, that is for $k = \left\lceil \frac{n_1 n_2 - 1}{n_1+n_2+2} \right\rceil$.

□

**Further analysis.** Can be interesting understand how the numbers from the partitioning and permutation algorithms relates to and the *G2DLP* ones.

Given a lattice $L(n_1, n_2)$, the number of paths from $P_0 = (0, 0)$ to $P_f = (n_1, n_2)$ in function of the number of turns $t$ graph is in general a bell shaped graph as a direct consequence of the binomial coefficients in the Theorem 1 formula.

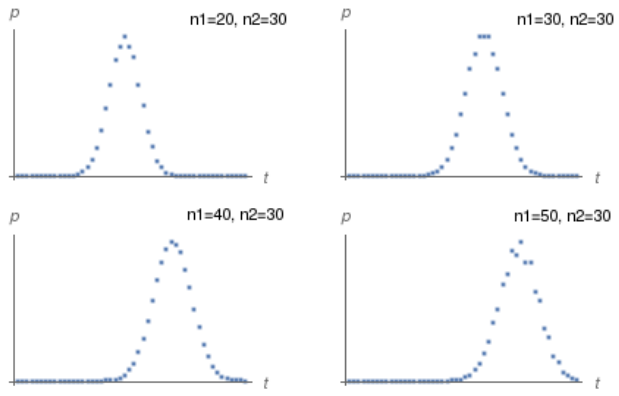


Figure 16: $paths(n_1, n_2, t)$

To understand how the partitioning algorithm, whose complexity function $p(n, t)$ graph looks like a binomial distribution with $p \approx 0.2$ (skewed to the right), can be part of an algorithm which resulting paths numbers distributes like a

binomial coefficient we must apply the permutation algorithm to the sequences generated by the partitioning function. Indeed, It has been empirically observed that if we partition a fixed number $n$ into $m$ parts, and then we permute each obtained sequence, then the number of such permutations graph has a perfect bell shape.

In short, if $p(n, m)$ is the function counting the number of partitions of an integer $n$ with a given number of parts $m$, then there exists a function $f$ and two non negative natural numbers $a$ and $b$ such that

$$f(p(n, m)) = \binom{a}{b}$$

Follows an experiment with $n = 30$ and $1 \leq m \leq n$.

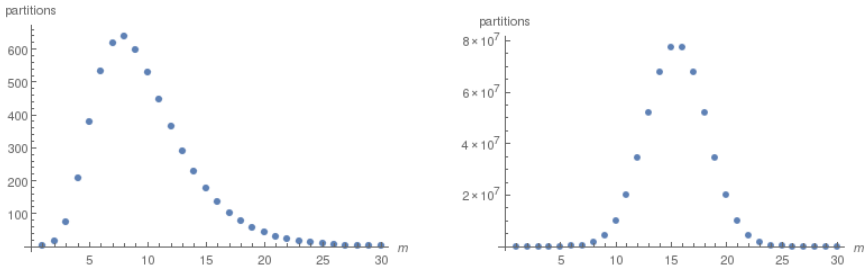


Figure 17: $p(n, m)$ and $f(p(n, m))$

| m | p(30,m) | f(p(30,m)) | m | p(30,m) | f(p(30,m)) |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 30 | 1 | 1 |
| 2 | 15 | 29 | 29 | 1 | 29 |
| 3 | 75 | 406 | 28 | 2 | 406 |
| 4 | 206 | 3654 | 27 | 3 | 3654 |
| 5 | 377 | 23751 | 26 | 5 | 23751 |
| 6 | 532 | 118755 | 25 | 7 | 118755 |
| 7 | 618 | 475020 | 24 | 11 | 475020 |
| 8 | 638 | 1560780 | 23 | 15 | 1560780 |
| 9 | 598 | 4292145 | 22 | 22 | 4292145 |
| 10 | 530 | 10015005 | 21 | 30 | 10015005 |
| 11 | 445 | 20030010 | 20 | 42 | 20030010 |
| 12 | 366 | 34597290 | 19 | 56 | 34597290 |
| 13 | 290 | 51895935 | 18 | 77 | 51895935 |
| 14 | 229 | 67863915 | 17 | 101 | 67863915 |
| 15 | 176 | 77558760 | 16 | 135 | 77558760 |

Table 3:

### 4.2.3 Implementation

In this implementation the fixed width partitions are generated via the *Partition* algorithm and each one is permuted using the *Permute* algorithm.

At the end the paths are generated using a function that performs the *ToSteps* and *AltMerge* algorithms.

```
//! Partitions list type alias
using parts_list = vector<vector<int>>;

//! Generates all the partitions permutations
//! @param n
//!    Integer value to partition.
//! @param m
//! Partition sequence length.
//! @param parts
//! Output partitions list.
static void partition(int n, int m, parts_list &parts)
{
    vector<int> a(m);    // Working sequence
    first_partition(n, a.begin(), a.end());
    do {
        do {
            parts.push_back(vector<int>(a));  // Copy
        } while (next_permute(a.begin(), a.end()));
    } while (next_partition(a.begin(), a.end()));
}

//! Alternate merge the two partition lists
//! @param step1
//!     Step type contained in the parts1 list
//! @param step2
//!     Step type contained in the parts2 list
//! @param parts1
//!     First partition list.
//! @param parts1
//!     Second partition list.
static void altmerge(char step1, char step2,
        parts_list &parts1, parts_list &parts2)
{
    for (auto p1 = parts1.begin(); p1 != parts1.end(); p1++)
    {
        for (auto p2 = parts2.begin(); p2 != parts2.end(); p2++)
        {
            string s;
            for (unsigned i = 0; i < p1->size()+p2->size(); i++)
            {
                auto q = i/2;
                if (i % 2 == 0)
                    for (auto j = 0; j < (*p1)[q]; j++)
                        s += step1;
                else
                    for (auto j = 0; j < (*p2)[q]; j++)
                        s += step2;
            }
            cout << s << endl;
```

```
        }
    }
}

//! G2DLP Algorithm as described by Kuo.
//! @param n1
//!     Lattice x-axis dimension.
//! @param n2
//!     Lattice y-axis dimension.
//! @param turns
//!     Number of turns.
void g2dlp(int n1, int n2, int turns)
{
    int k = (turns+1)/2;
    parts_list parts_a, parts_b, parts_c, parts_d;

    if (turns % 2 == 0)
    {
        // First step is Eastward
        partition(n1, k+1, parts_a);
        partition(n2, k,   parts_b);
        // First step is Northward
        partition(n2, k+1, parts_c);
        partition(n1, k,   parts_d);
    }
    else
    {
        // First step is Eastward
        partition(n1, k, parts_a);
        partition(n2, k, parts_b);
        // First step is Northward
        parts_c = parts_b;
        parts_d = parts_a;
    }

    altmerge('E', 'N', parts_a, parts_b);
    altmerge('N', 'E', parts_c, parts_d);
}
```

#### 4.2.4 Empirical results

**Variables**

- *n1* : lattice horizontal dimension;
- *n2* : lattice vertical dimension;
- *turns* : number of turns;
- *paths* : number of paths with a given number of turns;
- *time* : computation time in milliseconds.

**Experiment 1.** $2 \leq n_1 \leq 15$, $n_2 = n_1 = t$ . Table 4 resumes the experimental results for the *G2DLP* algorithm with $n_1 = n_2$ and with a number of *turns* equal to one of the dimensions.

Unlike the trivial approach, the number of turns is important for the result, and in this experiment we are going to set $turns = n_1$ with the effect to evaluate the worst case, that is the one that generates the maximum number of paths.

| n1 | n2 | turns | paths | g2dlp time | brute time |
|---|---|---|---|---|---|
| 2 | 2 | 2 | 2 | 0.014685 | 0.004969 |
| 3 | 3 | 3 | 8 | 0.03006 | 0.021282 |
| 4 | 4 | 4 | 18 | 0.046747 | 0.090375 |
| 5 | 5 | 5 | 72 | 0.129795 | 0.390691 |
| 6 | 6 | 6 | 200 | 0.35597 | 1.71823 |
| 7 | 7 | 7 | 800 | 1.45058 | 7.17305 |
| 8 | 8 | 8 | 2450 | 4.47957 | 18.6175 |
| 9 | 9 | 9 | 9800 | 8.32935 | 43.3554 |
| 10 | 10 | 10 | 31752 | 24.2522 | 176.945 |
| 11 | 11 | 11 | 127008 | 102.117 | 739.125 |
| 12 | 12 | 12 | 426888 | 371.213 | 3036.63 |
| 13 | 13 | 13 | 1707552 | 1602.57 | 12550.7 |
| 14 | 14 | 14 | 5889312 | 5992.7 | 51858.6 |
| 15 | 15 | 15 | 23557248 | 25269. | 213612. |

Table 4: G2DLP results, $n1 = n2 = t$

The results numbers grows very fast, to allow a better view of the graph the $x$ and $y$ axis were logarithmically scaled.

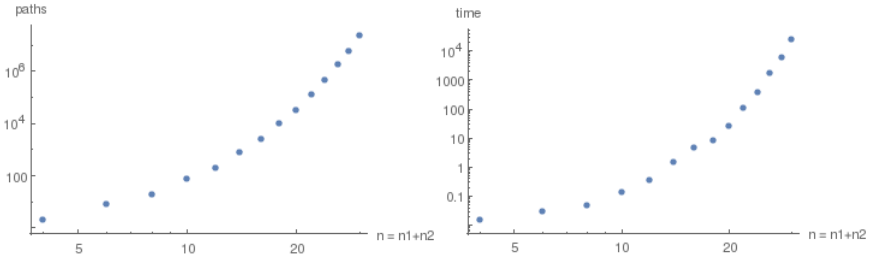


Figure 18: $paths = f(n_1 + n_2, t = n_1)$, $time = g(n_1 + n_2, t = n_1)$

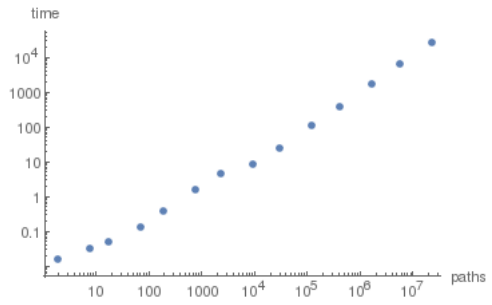


Figure 19: $time = h(paths)$

Table 4 also compares the *G2DLP* algorithm numbers with the the trivial ones. From the results is evident that the former increases the overall performances in the generation of the lattice paths with a given number of turns even in the worst case ($n_1 = n_2 = t$), anyway the algorithm is still exponential in complexity and not linear as the author asserts.

Figure 20 shows how, for two fixed values $n_1$ and $n_2$, the running time is bound to the number of generated paths and how these are distributed with respect to the number of turns $t$. Clearly the maximum number of paths, and so the generation time, is given for $t = n_1 = n_2$. This result is coherent with the Theorem 1 and the algorithm theoretical analysis.

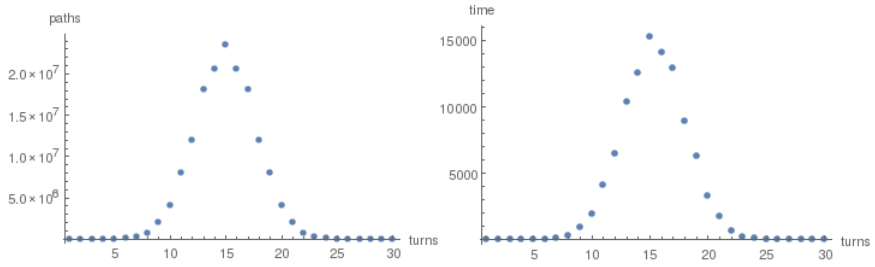


Figure 20: $paths = f(turns)$, $time = g(turns)$, $n_1 = n_2 = 15$

Using the experimental results from the *G2DLP worse* case $n_1 = n_2 = t$, the following linear relationships are found

$$time_{g2dlp} \approx 0.00107114 \cdot paths - 45.3796$$
$$time_{trivial} \approx 0.00906556 \cdot paths - 415.267$$

With a limit of the ratio $time_{trivial}/time_{g2dlp} \approx 8.46347$, meaning that, in the worst case, the *G2DLP* algorithm is still $\sim 8.5$ times faster than the trivial approach.

Is worth noting that this ratio grows exponentially while the number of turns moves away from the lattice dimensions.

**Experiment 2.** $2 \leq n_1 \leq 15$, $2 \leq n_2 \leq 15$ The results table is not reported here for space reasons, anyway from the graphs we can clearly see the exponential growth linked to the value of $n_1$ and $n_2$.

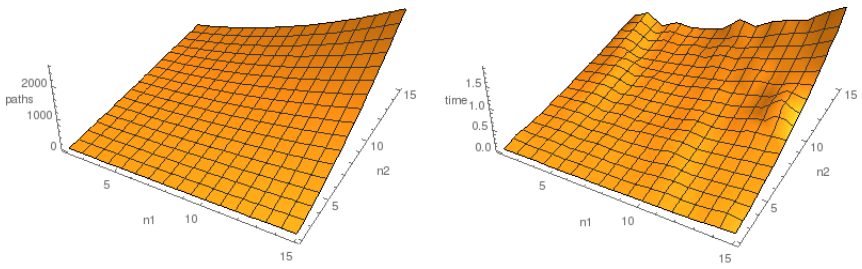


Figure 21: $paths = f(n1, n2, t = 4)$, $time = g(n1, n2, t = 4)$

,

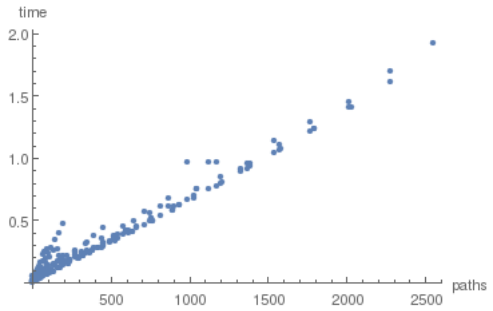


Figure 22: $time = h(paths, t = 4)$

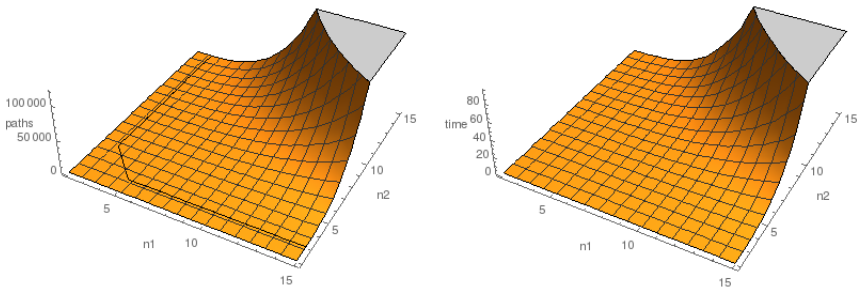


Figure 23: $paths = f(n1, n2, t = 8)$, $time = g(n1, n2, t = 8)$

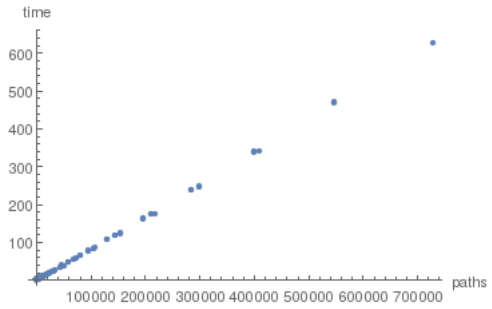


Figure 24: $time = h(paths, t = 8)$

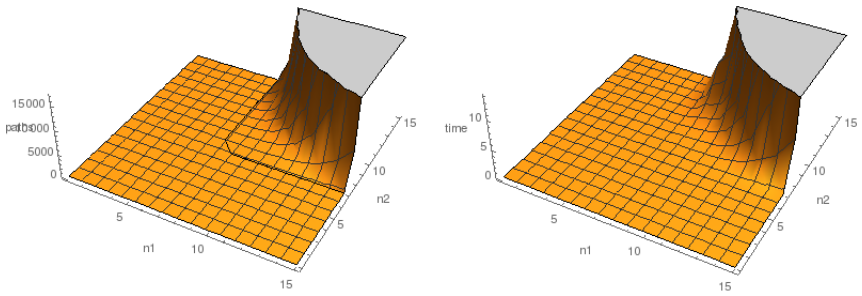


Figure 25: $paths = f(n1, n2, t = 15)$, $time = g(n1, n2, t = 15)$

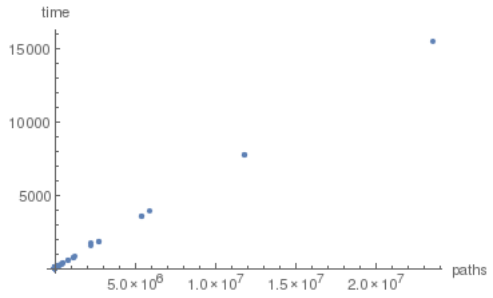


Figure 26: $time = h(paths, t = 15)$

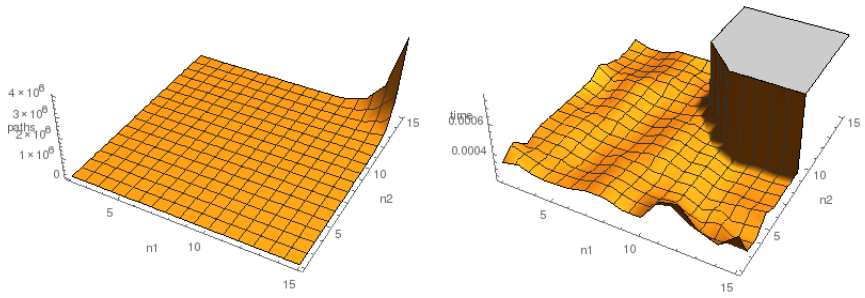


Figure 27: $paths = f(n1, n2, t = 20)$, $time = g(n1, n2, t = 20)$

,

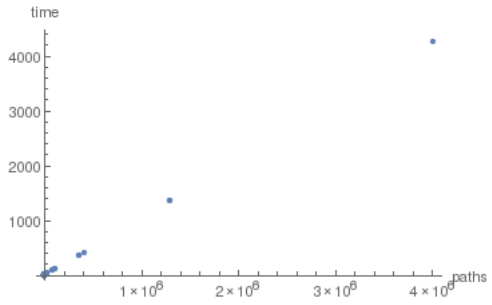


Figure 28: $time = h(paths, t = 20)$

## 4.3 Applications

### 4.3.1 Cryptography

The application has been presented by Kuo [1] in its paper. Here we are going to revisit the application with some improvements, especially related to the application of the theoretical results to an effective and usable implementation.

The proposed application combines a substitution cipher system with the *G2DLP* algorithm to become a product cipher system.

Given an ordered set of symbols $A = \{a_1, a_2, \ldots, a_n\}$ and a one-to-one function $f : A \to \mathbb{N}$, we shuffle the set $A$ by selecting one of its $|A|!$ permutations, for example by using an unranking algorithm proposed by Myrvold and Ruskey [7]. The cleartext is then processed by substituting each symbol $a_i$ with the positionally correspondent symbol found in the selected permutation of $A$, this produces the ciphertext.

Example. If $A = \{x, y, z, w\}$and $Permute(A) = \{w, z, y, x\}$. Then, the cleartext $xxzwwwzy$ is translated into $wwyxxxyz$.

Next, we process the ciphertext using the *G2DLP* algorithm to produce the final result.

Let $L(n_1, n_2)$ be a $2D$ lattice. If $M = max[f(A)]$ and $k = \lfloor \frac{M+1}{2} \rfloor$ then, to be sure that the *G2DLP* algorithm produces at least one path for each symbol $a_i$, it should be $(k < n_1) \wedge (k \leq n_2)$ or $(k \leq n_1) \wedge (k < n_2)$[3].

Each input symbol $a_i$ is encoded as a bit string of length $n_1+n_2$ representing one of the possible lattice paths from $P_0 = (0, 0)$ to $P_f = (n_1, n_2)$ with a number of turns equal to $f(a_i)$. Each $E$-step the path is encoded with a 0 (or a 1) and each $N$-step is encoded with the complementary value 1 (or 0). Which path to use from the possible ones is randomly selected each time the symbol $a_i$ is processed.

Example. If $L(3, 3)$ and $A = \{x = 1, y = 2, z = 3, w = 4\}$ then the following 6 bits strings can be used to encode each symbol

$f(x) = 1 \Rightarrow$ select from $\{000111, 111000\}$

$f(y) = 2 \Rightarrow$ select from $\{011100, 001110, 100011, 110001\}$

$f(z) = 3 \Rightarrow$ select from $\{010011, 011001, 001011, 001101, 101100$
$100110, 110100, 110010\}$

$f(w) = 4 \Rightarrow$ select from $\{010110, 011010, 101001, 100101\}$

The input string is then each time encrypted in a different way. The reverse process, from a ciphertext to a cleartext, is always possible if we know the values of $n_1$,$n_2$ and the substitution permutation.

To achieve a more effective result the lattice size can be arbitrary increased (only limited by the computing power of the machine), with the consequence to increment the number of possible paths, eg. if $f(a_i) = 13$ and the lattice size is $13 \times 13$ then each instance of the symbol $a_i$ can be encoded as one of the 1707552 possible 26 bits strings.

**Implementation notes.**

It was already proven that the complexity of the *G2DLP* algorithm is exponential. As an implementation note, is suggested to skip the *AltMerge* phase and select rendomly the paths directly from the vectors containing the separated $E$-steps and $N$-steps. This optimization skips all the cartesian product phase of the algorithm, that is indeed the most

Another observation can be made regarding the algorithm encoded data overhead. Assuming that what we want to transmit a sequence of bytes, then the symbols set cardinality is 256 and the simplest associated integer mapping function is $f : A \rightarrow [1 : 256]$. Since $max[f(A)] = 256$ then $k = 128$ and the lattice minimum dimensions are $129 \times 128$ or $128 \times 129$. Such a lattice is useless for the majority number of turns:

$>$ turns: 1, paths: 2, time: 0.023300

$>$ turns: 2, paths: 255, time: 1.123182

[3]These values are extracted from the *G2DLP* algorithm preconditions in case of an even number of turns

> turns: 3, paths: 32512, time: 31.987583
> turns: 4, paths: 2056384, time: 1099.612278
> turns: 5, paths: 130064256, time: 62135.459239
*...practically useless zone...*
> turns: 253, paths: 2064512, time: 13592.381973
> turns: 254, paths: 24384, time: 179.455214
> turns: 255, paths: 256, time: 2.084976
> turns: 256, paths: 1, time: 0.032493

The worst case is given for $k \approx \left\lceil \frac{n_1 n_2 - 1}{n_1 + n_2 + 2} \right\rceil = 64$.

$$P(n_1 = 128, n_2 = 129, t = 2k = 128) = 5.69245 \cdot 10^{74}$$

Just to have an idea of what we are talking about, if we use the interpolated linear function to get an extimated computation time we have

$$\begin{aligned} time &\approx 0.00107114 \cdot 5.69245 \cdot 10^{74} - 45.3796\, ms = \\ &= 6.09741 \cdot 10^{71}\, ms = 1.93348 \cdot 10^{61}\, years \end{aligned}$$

So ridiculously large numbers that are comparable with a *googol* ($10^{100}$).

Such an algorithm is realistically usable to encode just alphabets with a limited number of symbols, such as the 26 English letters and by using lookup tables.

### 4.3.2 Probability and game theory

Mohanty [5] proposed the following game. Take two coins 1 and 2 with probabilities $p_1$ and $p_2$ of obtaining heads, respectively. The rules of the game are:

1. start with a coin $i$, $i = 1, 2$;
2. if the last trial was a tail, then make the next trial with coin 1, otherwise with coin 2;
3. stop making further trials when for the first time the total number of heads exceeds $\mu$ times the total number of tails by exactly $a$, with a fixed $a > 0$.

The question is: provided the game was started by tossing coin $i$, $i = 1$ or 2, what is the distribution of the duration of the game?

It is an easy observation that any game can be represented in terms of a lattice path, by starting in $(0, 0)$ and proceding with an $E$-step if tail ($T$) was tossed and by an $N$-step if head ($H$) was tossed. If $n$ and $h$ are the current number of tails and heads, respectivelly, then the game continues while $h < \mu n + a$.

Thus, the game $THHHTHTHHHH$ (which is a game for $\mu = 2$ and $a = 2$) would be represented by the lattice path $P$ in Figure 29.

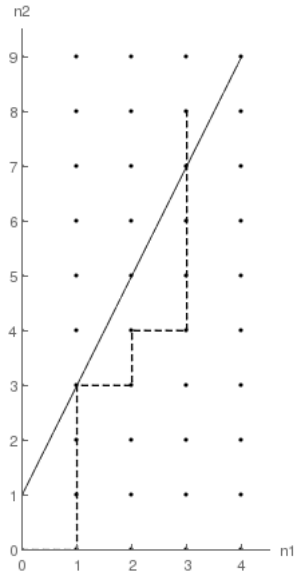


Figure 29: Dices Game

The probability of a game of length $n + h = n + \mu n + a = (\mu + 1)n + a$ ($n$ tails and $h = \mu n + a$ heads) is given as follows.

If the first toss was with coin 1, then the probability of a game, corresponding to a path $P$ as described above, is

$$p_1^{k+1}(1 - p_1)^{n-k} p_2^{\mu n+a-k-1}(1 - p_2)^k$$

where $k$ denotes the number of $E$-turns of the path.

On the other hand, if the first toss was with coin 2, then the probability of a game, corresponding to path $P$, is

$$p_1^{k+1}(1 - p_1)^{n-k-1} p_2^{\mu n+a-k-1}(1 - p_2)^{k+1},$$

if the first toss resulted in tail, and

$$p_1^{k}(1 - p_1)^{n-k} p_2^{\mu n+a-k}(1 - p_2)^{k},$$

if the first toss resulted in head, respectively.

Therefore, to determine the probability of games of length $(\mu + 1)n + a$, we need to enumerate lattice paths from $(0, 0)$ to $(n, \mu n + a - 1)$ staying below the line $y = \mu x + a - 1$, being allowed to touch it, which have a given number of $E$-turns.

To solve problems such as the one proposed it is sufficient to concentrate on the enumeration of lattice paths with a given starting and end points, satisfying certain restrictions, and with a given number of turns.

The game has been faced, along with some other interesting problems involving lattice paths enumeration with respect to their number of turns, by Krattenthaler [3].

# References


[1] D. E. Knuth. The art of computer programming, generating all paritions. Volume 4, Fascicle 3, 2005.

[2] D. E. Knuth. The art of computer programming, generating all tuples and permutations. Volume 4, Fascicle 2, 2005.

[3] C. Krattenthale. The enumeration of lattice paths with respect to their number of turns. 1997.

[4] T. Kuo. From enumerating to generating: A linear time algorithm for generating 2d lattice paths with a given number of turns. May 2015. 190-208.

[5] G. Mohanty. Lattice path counting and applications. January 1979.

[6] R. P. Stanley. Enumerative combinatorics, second edition. July 2011.

[7] F. Ruskey W. Myrvold. Ranking and unranking permutations in linear times. April 2000.

[8] H. S. Wilf. Lectures on integer partitions. July 2000.