On the number of strictly decreasing and increasing tuples

Matthias Schulte, M.Sc.*

October 1, 2023

Abstract

We derive a recursive formula for the number of strictly decreasing n-tuples with values in $\{1, \ldots, m\}$. We then implement a recursive algorithm in Python suitable to calculate this number in reasonable time. Furthermore we show that the same formula holds true asking for the number if strictly *increasing* tuples in the same setting. We then give a stochastic interpretation of these results.

Contents

*Email:

1 Strictly decreasing and increasing tuples

1.1 Notations

For $k \in \mathbb{N}$ we denote

$$[k] := \{1, \dots, k\} \subset \mathbb{N}$$

and

$$\Omega_m^n := \{\omega = (\omega_1, \dots, \omega_n) \colon \omega_i \in [m] \ \forall i \in [n]\} = [m]^n, n, m \in \mathbb{N}.$$

Definition 1.1.
A tuple $\omega = (\omega_1, \dots, \omega_n) \in \Omega_m^n$ is **strictly decreasing** if

$$\omega_i > \omega_{i+1} \ \forall i \in [n-1].$$

It is **strictly increasing** if

$$\omega_i < \omega_{i+1} \ \forall i \in [n-1].$$

By

$$S_m^n := \{\omega \in \Omega_m^n \colon \omega \text{ is strictly decreasing}\}$$

we denote the set of all strictly decreasing tuples in Ω_m^n. Furthermore let $s_{m,n}$ be the number of strictly decreasing tuples in Ω_m^n:

$$s_{m,n} := \#S_m^n.$$

The aim of this paper is to calculate $s_{m,n}$. To do so, we introduce an auxiliary size that measures the number of strictly decreasing tuples starting with a specified number:

$$f_{k,n} := \#\left\{\omega := (\omega_1, \dots, \omega_n) \in S_m^n \colon \omega_1 = k\right\}, \ k \in [m].$$

1.2 Calculation of $s_{m,n}$

We observe that all strictly decreasing tuples $\omega = (\omega_1, \ldots, \omega_n)$ of length n can be found recursively in the following way:
Fix $\omega_1 \in [m]$. Then append all strictly decreasing tuples of length $n-1$ to ω_1. Repeat this process for all possible values in $[m]$.

This observation leads us to the following recursive formula for $s_{m,n}$.
Theorem 1.2.
We have

$$f_{k,n} = \begin{cases} 0 & : \quad k < n \\ 1 & : \quad k = n \\ \sum_{j=1}^{k-1} f_{j,n-1} & : \quad k > n \end{cases}, \; n \geq 2; \; f_{k,1} := 1. \tag{1}$$

Therefore,

$$s_{m,n} = \sum_{k=1}^{m} f_{k,n}. \tag{2}$$

Remark 1.3.
Let's take a closer look at the cases of the calculation of $f_{k,n}$ in (1).

(1) We begin with $f_{k,1}$ for any $k \in [m]$. This is the number of 1-tuples starting with k; therefore we have only one tuple to count: (k). Hence, $f_{k,1} = 1$.

(2) In the case $k < n$ we need to count any strictly decreasing n-tuples starting with k. Due to the restriction $k < n$, there cannot be such ones; hence, $f_{k,n} = 0$ for $k < n$.

(3) In the case $k = n$ we need to count any strictly decreasing n-tuples starting with n. There is exactly one tuple satisfying this condition; namely

$$(n, n-1, \ldots, 2, 1).$$

Hence, $f_{n,n} = 1$.

(4) In the case $k > n$, formula (1) mirrors the above observation. We fix $\omega_1 = k$ and then append all strictly decreasing $(n-1)$-tuples, counted as $f_{j,n-1}$ depending on ω_2.

Before we give a formal proof of Theorem 1.2, let's check one concrete example.
Example 1.4.
Suppose $m = 6$ and $n = 4$. In this small case, we can count all strictly decreasing n-tuples by just listing all of them:

$(6,5,4,3), (6,5,4,2), (6,5,4,1), (6,5,3,2), (6,5,3,1), (6,5,2,1), (6,4,3,2),$
$(6,4,3,1), (6,4,2,1), (6,3,2,1),$
$(5,4,3,2), (5,4,3,1), (5,4,2,1), (5,3,2,1),$
$(4,3,2,1)$

Therefore, $s_{6,4} = 15$. Now we calculate $s_{6,4}$ using formula (1) and (2).
We have

$$s_{6,4} \overset{(2)}{=} \underbrace{f_{1,4}}_{=0} + \underbrace{f_{2,4}}_{=0} + \underbrace{f_{3,4}}_{=0} + \underbrace{f_{4,4}}_{=1} + \underbrace{f_{5,4}}_{=4} + \underbrace{f_{6,4}}_{\to(*_1)} = 1 + 4 + 10 = 15,$$

$$f_{6,4} \overset{k \geq n}{=} \underbrace{f_{1,3}}_{=0} + \underbrace{f_{2,3}}_{=0} + \underbrace{f_{3,3}}_{=1} + \underbrace{f_{4,3}}_{\to(*_2)} + \underbrace{f_{5,3}}_{\to(*_4)} = 1 + 3 + 6 = 10, \qquad (*_1)$$

$$f_{4,3} \overset{k \geq n}{=} \underbrace{f_{1,2}}_{=0} + \underbrace{f_{2,2}}_{=1} + \underbrace{f_{3,2}}_{\to(*_3)} = 1 + 2 = 3, \qquad (*_2)$$

$$f_{3,2} \overset{k \geq n}{=} \underbrace{f_{1,1}}_{=1} + \underbrace{f_{2,1}}_{=1} = 1 + 1 = 2, \qquad (*_3)$$

$$f_{5,3} \overset{k \geq n}{=} \underbrace{f_{1,2}}_{=0} + \underbrace{f_{2,2}}_{=1} + \underbrace{f_{3,2}}_{=2,\ (*_3)} + \underbrace{f_{4,2}}_{\to(*_5)} = 1 + 2 + 3 = 6, \qquad (*_4)$$

$$f_{4,2} \overset{k \geq n}{=} \underbrace{f_{1,1}}_{=1} + \underbrace{f_{2,1}}_{=1} + \underbrace{f_{3,1}}_{=1} = 1 + 1 + 1 = 3, \qquad (*_5)$$

so (2) checks out.

Remark 1.5.
As you can see in above example, the recursion computes values multiple times; in above example the value $f_{3,2}$. This indicates that the usage of a *cache* implementing this formula should be considered later on.

Exercise 1.6.
Proof $f_{5,4} = 4$ rigorously using formula (1).

Proof of Theorem 1.2.
We prove formula (1) by induction on n.
Base cases: $n = 1, 2$.
The case $n = 1$ is obviously true.
For $n = 2$ we have to show the following:

$$f_{k,2} = \begin{cases} 0 & : \quad k < 2 \\ 1 & : \quad k = 2 \\ \sum_{j=1}^{k-1} f_{j,1} & : \quad k > 2 \end{cases} = \begin{cases} 0 & : \quad k < n \\ 1 & : \quad k = n \\ k - 1 & : \quad k > n \end{cases}.$$

For $k < 2$ this is correct. For $k = 2$ we only have on valid tuple, namely $(2, 1)$; therefore, $f_{2,2} = 1$. For $k > 2$ we seek tuples of the form $\omega := (k, l)$ and for ω

to be strictly decreasing, $l \leq k-1$ is necessary; hence, $f_{k,2} = k-1$. ✓
INDUCTION HYPOTHESIS: Suppose

$$f_{k,n-1} = \begin{cases} 0 & : \quad k < n-1 \\ 1 & : \quad k = n-1 \\ \sum_{j=1}^{k-1} f_{j,n-2} & : \quad k > n-1 \end{cases}$$

for $n \in \mathbb{N}$.
INDUCTION STEP: $n-1 \to n$.

Case 1: $k < n$.
We can only choose

$$(k, k-1, \ldots, k-(n-1)) = (k, k-1, \ldots, \underbrace{k-n+1}_{\leq 0}),$$

which is not a valid tuple in Ω_m^n. Hence, $f_{k,n} = 0$ for $k < n$. ✓

Case 2: $k = n$.
We can only choose

$$(k, k-1, \ldots, k-(n-1)) = (k, k-1, \ldots, \underbrace{k-n+1}_{=1}) \in \S_m^n,$$

hence, $f_{n,n} = 1$. ✓

Case 3: $k > n$.
By definition of $f_{k,n}$ we look at n-tuples of the form

$$(k, \omega_2, \ldots, \omega_n) \in S_m^n.$$

For these to be strictly decreasing,

$$(\omega_2, \ldots, \omega_n) \in S_m^{n-1}$$

is necessary. Due to $\omega_j \in [m]$ for all $j \in [n]$, ω_2 can be replaced by $k-1$ numbers. For each of these numbers j there are $f_{j,n-1}$ strictly decreasing tuples using the INDUCTION HYPOTHESIS.
To arrive at $f_{k,n}$ as result we just need to add up all those numbers:

$$f_{k,n} = f_{k-1,n-1} + f_{k-2,n-1} + \ldots + f_{2,n-1} + f_{1,n-1} = \sum_{j=1}^{k-1} f_{j,n-1}.$$

This is the expected result we needed to show. ✓

Hereby, formula (1) is proven. Formula (2) is therefore correct by construction. □

Corollary 1.7.
By construction and Theorem 1.2 we have

$$f_{k,n} = \begin{cases} 0 & : \quad k < n \\ 1 & : \quad k = n \\ \sum_{j=n-1}^{k-1} f_{j,n-1} & : \quad k > n \end{cases} = \begin{cases} 0 & : \quad k < n \\ 1 & : \quad k = n \\ 1 + \sum_{j=n}^{k-1} f_{j,n-1} & : \quad k > n \end{cases}$$

due to all the missing summands being 0.

1.3 A recursive algorithm for computing $s_{m,n}$

The recursive algorithm we are going to present is essentially just the exact implementation of formulae (1) and (2). We will break this down in several pieces and show the complete code in the end. Furthermore we give experimental runtimes in seconds for each method.

We begin with an implementation to calculate $f_{k,n}$. We write this method using the recursive formula (1) passing k and n as parameters and using a built-in Python-cache to memorize already computed values.

```
from functools import cache

@cache
def f(k: int, n: int) -> int:
  if k < n:
    return 0
  if k == n or n == 1:
    return 1
  res = 0
  for j in range(n - 1, k):
    res += f(j, n - 1)
  return res
```

To give an experimental runtime overview, we will run this function with the values

$$k \in \{10, 20, 50, 100, 500, 1000\}, n \in \{5, 10, 15, 20, 25, 30\},$$

using the following code:

```
import time

def time_calculation_of_f_k_n():
  ks = [10, 20, 50, 100, 500, 1000]
  ns = [10, 15, 20, 25, 30, 50]
  for k in ks:
    for n in ns:
      f_k_n, t = execute_f_with_timing(k, n)
      print(f"k: {k},  \t\t\t n: {n}; \t\t time: {round(t, 2)}
  seconds.")


def execute_f_with_timing(k: int, n: int) -> [int, float]:
  time_start = time.time()
  f_k_n = f(k, n)
  time_end = time.time()
  time_total = time_end - time_start
  return f_k_n, time_total
```

These are the results. We exclude the values of $f_{k,n}$ due to the numbers getting big pretty fast.

```
k: 10,          n: 10;       time: 0.0 seconds.
k: 10,          n: 15;       time: 0.0 seconds.
k: 10,          n: 20;       time: 0.0 seconds.
k: 10,          n: 25;       time: 0.0 seconds.
k: 10,          n: 30;       time: 0.0 seconds.
k: 10,          n: 50;       time: 0.0 seconds.
k: 20,          n: 10;       time: 0.0 seconds.
k: 20,          n: 15;       time: 0.0 seconds.
k: 20,          n: 20;       time: 0.0 seconds.
k: 20,          n: 25;       time: 0.0 seconds.
```

```
k: 20,          n: 30;      time: 0.0 seconds.
k: 20,          n: 50;      time: 0.0 seconds.
k: 50,          n: 10;      time: 0.0 seconds.
k: 50,          n: 15;      time: 0.0 seconds.
k: 50,          n: 20;      time: 0.0 seconds.
k: 50,          n: 25;      time: 0.0 seconds.
k: 50,          n: 30;      time: 0.0 seconds.
k: 50,          n: 50;      time: 0.0 seconds.
k: 100,         n: 10;      time: 0.0 seconds.
k: 100,         n: 15;      time: 0.0 seconds.
k: 100,         n: 20;      time: 0.0 seconds.
k: 100,         n: 25;      time: 0.0 seconds.
k: 100,         n: 30;      time: 0.0 seconds.
k: 100,         n: 50;      time: 0.0 seconds.
k: 500,         n: 10;      time: 0.09 seconds.
k: 500,         n: 15;      time: 0.06 seconds.
k: 500,         n: 20;      time: 0.06 seconds.
k: 500,         n: 25;      time: 0.05 seconds.
k: 500,         n: 30;      time: 0.07 seconds.
k: 500,         n: 50;      time: 0.22 seconds.
k: 1000,        n: 10;      time: 0.32 seconds.
k: 1000,        n: 15;      time: 0.2 seconds.
k: 1000,        n: 20;      time: 0.2 seconds.
k: 1000,        n: 25;      time: 0.21 seconds.
k: 1000,        n: 30;      time: 0.21 seconds.
k: 1000,        n: 50;      time: 0.83 seconds.
```

As you can see, the algorithm works pretty fast even for large values of k. To scale this further we ran a test with increased recursion depth and increased numbers.

```
  import time

  def time_calculation_of_f_n_k_edge_case():
    sys.setrecursionlimit(10000)
    k = 2000
    n = 1000
    time_start = time.time()
    result = f(k=k, n=n)
    time_end = time.time()
    time_total = time_end - time_start
    print(f"k: {k}, n: {n}; time: {round(time_total, 2)} seconds")
```

This resulted in the following logging:

```
  k: 2000, n: 1000; time: 216.82 seconds
```

In summary it seems safe to say that the algorithm works in reasonable time for reasonable big input numbers k and n.

The consequential algorithm for the computation of $s_{m,n}$ can be derived pretty simply from formula (2) and the above algorithm for $f_{k,n}$:

```
  def s(m: int, n: int) -> int:
    return sum([f(k, n) for k in range(1, m + 1)])
```

We need to pass $m + 1$ in the range function because it excludes the upper bound:

$$range(1, m + 1) \simeq [m].$$

We test this algorithm in the same way as the above algorithm for $f_{k,n}$.

```
def time_calculation_of_s_m_n():
  ms = [10, 20, 50, 100, 500, 1000]
  ns = [10, 15, 20, 25, 30, 50]
  for m in ms:
    for n in ns:
      s_m_n, t = execute_f_with_timing(m, n)
      print(f"m: {m},  \t\t\t n: {n}; \t\t time: {round(t, 2)}
  seconds.")


def execute_s_with_timing(m: int, n: int) -> [int, float]:
  time_start = time.time()
  s_m_n = s(m, n)
  time_end = time.time()
  time_total = time_end - time_start
  return s_m_n, time_total
```

This results in the following values:

```
m: 10,          n: 10;       time: 0.0 seconds.
m: 10,          n: 15;       time: 0.0 seconds.
m: 10,          n: 20;       time: 0.0 seconds.
m: 10,          n: 25;       time: 0.0 seconds.
m: 10,          n: 30;       time: 0.0 seconds.
m: 10,          n: 50;       time: 0.0 seconds.
m: 20,          n: 10;       time: 0.0 seconds.
m: 20,          n: 15;       time: 0.0 seconds.
m: 20,          n: 20;       time: 0.0 seconds.
m: 20,          n: 25;       time: 0.0 seconds.
m: 20,          n: 30;       time: 0.0 seconds.
m: 20,          n: 50;       time: 0.0 seconds.
m: 50,          n: 10;       time: 0.0 seconds.
m: 50,          n: 15;       time: 0.0 seconds.
m: 50,          n: 20;       time: 0.0 seconds.
m: 50,          n: 25;       time: 0.0 seconds.
m: 50,          n: 30;       time: 0.0 seconds.
m: 50,          n: 50;       time: 0.0 seconds.
m: 100,         n: 10;       time: 0.0 seconds.
m: 100,         n: 15;       time: 0.0 seconds.
m: 100,         n: 20;       time: 0.0 seconds.
m: 100,         n: 25;       time: 0.0 seconds.
m: 100,         n: 30;       time: 0.0 seconds.
m: 100,         n: 50;       time: 0.0 seconds.
m: 500,         n: 10;       time: 0.09 seconds.
m: 500,         n: 15;       time: 0.06 seconds.
m: 500,         n: 20;       time: 0.06 seconds.
m: 500,         n: 25;       time: 0.06 seconds.
m: 500,         n: 30;       time: 0.06 seconds.
m: 500,         n: 50;       time: 0.22 seconds.
m: 1000,        n: 10;       time: 0.31 seconds.
m: 1000,        n: 15;       time: 0.2 seconds.
m: 1000,        n: 20;       time: 0.21 seconds.
m: 1000,        n: 25;       time: 0.2 seconds.
m: 1000,        n: 30;       time: 0.21 seconds.
m: 1000,        n: 50;       time: 0.83 seconds.
```

Again it is safe to say that this algorithm is reasonably fast.
Furthermore we ran a similar edge case test for the calculation of $s_{m,n}$ using the following code:

```
def time_calculation_of_s_m_n_edge_case():
  sys.setrecursionlimit(10000)
  m = 2000
```

```
    n = 1000
    time_start = time.time()
    result = s(m=m, n=n)
    time_end = time.time()
    time_total = time_end - time_start
    print(f"m: {m}, n: {n}; time: {round(time_total, 2)} seconds")
```

This results in the following numbers:

```
m: 2000, n: 1000; time: 198.61 seconds
```

To sum this section up we give the complete code of the algorithms for $f_{k,n}$ and $s_{m,n}$. The executable py-file including the testing methods for timing is included in this papers github.[1]

```
import time
from functools import cache

@cache
def f(k: int, n: int) -> int:
  if k < n:
    return 0
  if k == n or n == 1:
    return 1
  res = 0
  for j in range(n - 1, k):
    res += f(j, n - 1)
  return res


def s(m: int, n: int) -> int:
  return sum([f(k, n) for k in range(1, m + 1)])
```

[1] **TODO!**

1.4 Strictly increasing tuples

In this section we want to generalize our results on strictly *increasing* tuples. Let $\omega := (\omega_1, \ldots, \omega_n) \in \Omega_m^n$. By $\overleftarrow{\omega}$ we denote the *reverse tuple* of ω:

$$\overleftarrow{\omega} := (\omega_n, \ldots, \omega_1).$$

Furthermore we set

$$\overleftarrow{S}_m^n := \{\overleftarrow{\omega} \colon \omega \in S_m^n\}.$$

Proposition 1.8.
We have $\#S_m^n = \#\overleftarrow{S}_m^n$.

Proof. If we choose $\omega \in S_m^n$, by definition we have $\overleftarrow{\omega} \in \overleftarrow{S}_m^n$. □

Exercise 1.9.
Show rigorously that the mapping

$$\varphi : S_m^n \to \overleftarrow{S}_m^n,\ \varphi(\omega) := \overleftarrow{\omega},$$

is a bijection.

Let $\overleftarrow{s}_{m,n}$ be the number of strictly increasing n-tuples with values in $[m]$ and $\overleftarrow{f}_{k,n}$ the number of strictly increasing n-tuples with values in $[m]$ starting with $k \in [m]$.

Corollary 1.10.
We have $\overleftarrow{f}_{k,n} = f_{k,n}$ for $k, n \in \mathbb{N}$.

Theorem 1.11.
We have the following results:

(1) $\overleftarrow{s}_{m,n} = \sum_{k=1}^{m} \overleftarrow{f}_{k,n}$.

(2) $\overleftarrow{s}_{m,n} = s_{m,n}$.

(3) The same algorithm as before can be applied calculating $\overleftarrow{s}_{m,n}$.

Proof.

(1) This is true by construction.

(2) Using Corollary 10 we have

$$\overleftarrow{s}_{m,n} \overset{(1)}{=} \sum_{k=1}^{m} \overleftarrow{f}_{k,n} \overset{\text{Cor. } 10}{=} \sum_{k=1}^{m} f_{k,n} \overset{(2)}{=} s_{m,n}.$$

(3) This follows directly from (2).

□

1.5 Stochastic interpretation

To put our results in a stochastic interpretation, consider the following setup: We throw a fair m-sided dice n times and denote the thrown numbers as $\omega := (\omega_1, \ldots, \omega_n)$. What is the probability of ω to be strictly decreasing?

We first build a stochastic model of this situation. Let X_i be the random variable describing the result of the i-th dice throw:

$$X_i \in [m] \ \forall i \in [n].$$

Then the random vector $X := (X_1, \ldots, X_n)$ describes the tuple of all n dice throws. We suppose all dice throws to be independent from each other and the dice to be fair:

$$X_i \sim \mathcal{U}([m]) \Rightarrow \mathbb{P}(X_i = k) = \frac{1}{m} \ \forall k \in [m],$$

where $\mathbb{P}$ is the discrete uniform distribution on Ω_m^n:

$$\mathbb{P}(A) := \frac{\#A}{\#\Omega_m^n} = \#A \cdot m^{-n}, \ A \in \mathcal{P}(\Omega_m^n).$$

In summary, our underlying probability space is given by

$$(\Omega_m^n, \mathcal{P}(\Omega_m^n), \mathbb{P}).$$

We are interested in the event that our random vector X lies in S_m^n; using above notations we want to calculate

$$p_{m,n} := \mathbb{P}(X \in S_m^n).$$

Theorem 1.12.
We have $p_{m,n} = s_{m,n} \cdot m^{-n}$.

Proof.
The event "$X \in S_m^n$" translates to the following calculation:

$$\mathbb{P}(X \in S_m^n) = \mathbb{P}(S_m^n) = \#S_m^n \cdot m^{-n} = s_{m,n} \cdot m^{-n}.$$

□

In this given setup it is irrelevant whether we ask for ω to be strictly increasing or decreasing. We will give a precision of this statement using the following notation:

$$\overleftarrow{p}_{m,n} := \mathbb{P}(\overleftarrow{X} \in \overleftarrow{S}_m^n).$$

Theorem 1.13.
We have the following results:

(1) $\overleftarrow{p}_{m,n} = \overleftarrow{s}_{m,n} \cdot m^{-n}$.

(2) $\overleftarrow{p}_{m,n} = p_{m,n}$.

Proof.

(1) The event "$\overleftarrow{X} \in \overleftarrow{S}_m^n$" translates to the following calculation:

$$\mathbb{P}(\overleftarrow{X} \in \overleftarrow{S}_m^n) = \mathbb{P}(\overleftarrow{S}_m^n) = \#\overleftarrow{S}_m^n \cdot m^{-n} = \overleftarrow{s}_{m,n} \cdot m^{-n}.$$

(2) We have

$$\overleftarrow{p}_{m,n} \overset{(1)}{=} \overleftarrow{s}_{m,n} \cdot m^{-n} \overset{\text{Thm. 11 (2)}}{=} s_{m,n} \cdot m^{-n} \overset{Def.}{=} p_{m,n}.$$

□

To close this paper we will give an algorithm calculating $p_{m,n}$ using the established algorithms above calculating $s_{m,n}$ and $f_{k,n}$.

Remark 1.14.

The probability $p_{m,n}$ (and by Theorem 13 also $\overleftarrow{p}_{m,n}$) can be calculated using the following algorithm:

```
    def p(m: int, n: int) -> float:
      return s(m, n) / pow(m, n)
```

We measure the runtime of this using the same methods as before:

```
    def time_calculation_of_p_m_n():
      ms = [10, 20, 50, 100, 500, 1000]
      ns = [10, 15, 20, 25, 30, 50]
      for m in ms:
      for n in ns:
      p_m_n, t = execute_f_with_timing(m, n)
      print(f"m: {m},  \t\t\t n: {n}; \t\t time: {round(t, 2)}
    seconds.")


    def execute_p_with_timing(m: int, n: int) -> [int, float]:
      time_start = time.time()
      p_m_n = p(m, n)
      time_end = time.time()
      time_total = time_end - time_start
      return p_m_n, time_total
```

The results are the following:

```
    m: 10,          n: 10;       time: 0.0 seconds.
    m: 10,          n: 15;       time: 0.0 seconds.
    m: 10,          n: 20;       time: 0.0 seconds.
    m: 10,          n: 25;       time: 0.0 seconds.
    m: 10,          n: 30;       time: 0.0 seconds.
    m: 10,          n: 50;       time: 0.0 seconds.
    m: 20,          n: 10;       time: 0.0 seconds.
    m: 20,          n: 15;       time: 0.0 seconds.
    m: 20,          n: 20;       time: 0.0 seconds.
    m: 20,          n: 25;       time: 0.0 seconds.
    m: 20,          n: 30;       time: 0.0 seconds.
    m: 20,          n: 50;       time: 0.0 seconds.
    m: 50,          n: 10;       time: 0.0 seconds.
    m: 50,          n: 15;       time: 0.0 seconds.
    m: 50,          n: 20;       time: 0.0 seconds.
    m: 50,          n: 25;       time: 0.0 seconds.
    m: 50,          n: 30;       time: 0.0 seconds.
    m: 50,          n: 50;       time: 0.0 seconds.
    m: 100,         n: 10;       time: 0.0 seconds.
    m: 100,         n: 15;       time: 0.0 seconds.
```

```
    m: 100,         n: 20;      time: 0.0 seconds.
    m: 100,         n: 25;      time: 0.0 seconds.
    m: 100,         n: 30;      time: 0.0 seconds.
    m: 100,         n: 50;      time: 0.0 seconds.
    m: 500,         n: 10;      time: 0.08 seconds.
    m: 500,         n: 15;      time: 0.04 seconds.
    m: 500,         n: 20;      time: 0.05 seconds.
    m: 500,         n: 25;      time: 0.04 seconds.
    m: 500,         n: 30;      time: 0.04 seconds.
    m: 500,         n: 50;      time: 0.22 seconds.
    m: 1000,        n: 10;      time: 0.26 seconds.
    m: 1000,        n: 15;      time: 0.17 seconds.
    m: 1000,        n: 20;      time: 0.17 seconds.
    m: 1000,        n: 25;      time: 0.17 seconds.
    m: 1000,        n: 30;      time: 0.17 seconds.
    m: 1000,        n: 50;      time: 0.75 seconds.
```

As above this is reasonably fast.
The edge case simulation works as follows:

```
    def time_calculation_of_p_m_n_edge_case():
      sys.setrecursionlimit(10000)
      m = 2000
      n = 1000
      time_start = time.time()
      result = p(m=m, n=n)
      time_end = time.time()
      time_total = time_end - time_start
      print(f"m: {m}, n: {n}; time: {round(time_total, 2)} seconds"
    )
```

This is the result:

```
    m: 2000, n: 1000; time: 143.38 seconds
```

2 Open questions

We finish with some open questions which will be tackled in future papers (if any progress can be achieved).

(1) Is there a closed formula for $f_{n,k}$, enabling an iterative algorithm with (potentially) faster runtime?

(2) What is if the tuple is just required to be "weakly" decreasing / increasing?

(3) What is the theoretical runtime of the recursive algorithm?

We leave this questions open to discussion.