From the Summation of the Quotient Function to the Partition of Integers

Darshan P.
Independent Researcher, Computer Programming Specialist
drshnp@bayesiancorp.com

1732891873

Abstract

This paper introduces a formula to compute $P(n,k)$, the number of partitions of an integer n into exactly k parts. The formula is derived using the quotient function $q(n,k) = \lfloor n/k \rfloor$ and simple summations. While the formula becomes recursive in nature, leveraging caching and memoization allows efficient computational verification. The results align with those derived from Euler's integer-partition identity, demonstrating the formula's validity and correctness.

1 Formula and Approach

The primary contribution of this paper is a formula to compute $p(n,k)$, the number of partitions of an integer n into exactly k parts. This formula is based on the quotient function $q(n,k) = \lfloor n/k \rfloor$, which forms the foundation of all computations. By leveraging this simple yet powerful operation, the formula employs summations and recursive constructs to evaluate partition counts without the need for traditional recursive breakdowns. To generalize, we define

$$p_k(n) = p(n,k),$$
$$Q_k(n) = \big\lfloor n/k \big\rfloor.$$

To begin, $p(n,2)$ is simply $Q_2(n)$:

$$p_2(n) = Q_2(n) = \big\lfloor n/2 \big\rfloor.$$

For three parts we obtain

$$p(n,3) = \sum_{i=1}^{Q_3(n)} \big(Q_2(n-i) - (i-1)\big),$$

and for four parts

$$p(n,4) = \sum_{u=0}^{Q_4(n)-1} p_3\big(n-4u-1\big).$$

A similar pattern gives, for $n > 3$,

$$p(n,5) = \sum_{u=0}^{Q_5(n)} p_4\big(n-5u-1\big),$$
$$p(n,6) = \sum_{u=0}^{Q_6(n)} p_5\big(n-6u-1\big),$$
$$p(n,7) = \sum_{u=0}^{Q_7(n)} p_6\big(n-7u-1\big),$$
$$p(n,8) = \sum_{u=0}^{Q_8(n)-1} p_7\big(n-8u-1\big),$$
$$p(n,9) = \sum_{u=0}^{Q_9(n)} p_8\big(n-9u-1\big).$$

2 Computational Verification

$$\boxed{\begin{aligned} p(n,k>3) &= \sum_{u=0}^{Q_k(n)-f_k} p_{k-1}(n-ku-1), \\ f_k &= \begin{cases} 1, & k \equiv 0 \pmod 2, \\ 0, & \text{otherwise}, \end{cases} \end{aligned}}$$

or equivalently

$$\boxed{p(n,k>3) = \sum_{u=0}^{Q_k(n)-f_k} p_{k-1}(n-ku-1), \quad f_k = \sin^2\!\Big(\frac{\pi(k-1)}{2}\Big).}$$

These recurrences are checked against Euler's identity [2]

$$p(n,k) = p(n-1,k-1) + p(n-k,k).$$

2.1 Testing against Euler's identity

```python
from math import floor
import numpy as np
import random
from tabulate import tabulate

# Define the division logic
q = lambda n, k: n // k

# Recursive function to compute p_k(n)
def compute_p_k(n, k):
    if k == 3:
        # Base case for p_3(n)
        q_3 = q(n, 3)
```

```
        s = 0
        for i in range(1, q_3 + 1):
            s += q(n - i, 2) - (i - 1)
        return s
    else:
        # Recursive computation for p_k(n)
        q_k = q(n, k)
        s = 0
        for u in range(q_k):
            s += compute_p_k(n - k * u - 1, k - 1)
        return s

# Function to compute the number of partitions into exactly k parts
# Euler's Identity of Partition of Integers
def partitions_into_k_parts(n, k):
    dp = [[0] * (k + 1) for _ in range(n + 1)]
    dp[0][0] = 1  # Base case: 0 can be partitioned into 0 parts in 1 way

    for i in range(1, n + 1):
        for j in range(1, k + 1):
            if i >= j:
                dp[i][j] = dp[i - 1][j - 1] + dp[i - j][j]
            else:
                dp[i][j] = 0
    return dp[n][k]

# Test function to compare compute_p_k with partitions_into_k_parts
def test_compute_p_k():
    MAX_N = 100  # Reduce for quicker testing (adjustable)
    num_tests = 750  # Number of test cases
    random_values = []
    mismatches = []

    for _ in range(num_tests):
        n = random.randint(3, MAX_N)  # Random n between 3 and MAX_N
        k = random.randint(3, n)      # Random k between 3 and n

        # Compute using both methods
        result_p_k = compute_p_k(n, k)
        result_partition = partitions_into_k_parts(n, k)

        # Append to results
        random_values.append([n, k, result_p_k, result_partition])

        # Check for mismatches
        if result_p_k != result_partition:
            mismatches.append((n, k, result_p_k, result_partition))

    # Convert results to numpy array
    data = np.array(random_values, dtype=object)

    # Display results in a table
```

```
    headers = ["n", "k", "compute_p_k", "partitions_into_k_parts"]
    print(tabulate(data, headers=headers, tablefmt="grid"))

    # Print mismatches
    if mismatches:
        print("Mismatches Found:")
        print(tabulate(mismatches, headers=headers, tablefmt="grid"))
    else:
        print("All results match!")

if __name__ == "__main__":
    test_compute_p_k()
```

Listing 1: Code to test p_k against Euler's identity of integer partitions.

3 Recursive Implementation and Memoization

Although intuitive, a naive recursive evaluation repeats many sub-calls. Caching eliminates that duplication.

1. Initialize a cache for (n, k) pairs.
2. Check the cache before each computation.
3. Store every newly computed result.

3.1 Cache Memoization for $p(n)$

```
from math import floor
from functools import lru_cache

class PartitionCalculator:
    def __init__(self):
        # Memoized cache for compute_p_k
        self.cache = {}

    # Division logic
    def q(self, n, k):
        return n // k

    # Compute p_k(n) using memoization
    def compute_p_k(self, n, k):
        # Check for cached result
        if (n, k) in self.cache:
            return self.cache[(n, k)]

        if n < 0:
            return 0  # No partitions possible for negative n
        if k == 3:
            # Base case for p_3(n)
            q_3 = self.q(n, 3)
```

```
            s = 0
            for i in range(1, q_3 + 1):
                s += self.q(n - i, 2) - (i - 1)
            self.cache[(n, k)] = s
            return s
        else:
            # Recursive computation for p_k(n)
            q_k = self.q(n, k)
            s = 0
            for u in range(q_k):
                s += self.compute_p_k(n - k * u - 1, k - 1)
            self.cache[(n, k)] = s
            return s

    # Compute the summation of p_3(n) to p_n(n)
    def compute_sum_p_k(self, n):
        total = 0
        for k in range(3, n + 1):
            total += self.compute_p_k(n, k)
        total += self.q(n, 2) + 1  # The partitions into two parts and itself.
        return total

if __name__ == "__main__":
    # Print instructions for the user
    print("Enter the values of N:")
    print("- For a single value, just enter the integer (e.g., 42)")
    print("- For multiple values, enter them as a comma-separated list (e.g.,
        42, 53, 13423, 52)")

    # Input from the user
    inputs = input("Your input: ").strip()

    # Determine if the input is a single number or a list
    if ',' in inputs:
        # Convert input string to a list of integers if comma-separated
        n_values = list(map(int, inputs.split(',')))
    else:
        # Convert single input to a list with one element
        n_values = [int(inputs)]

    # Create an instance of PartitionCalculator
    calculator = PartitionCalculator()

    # Compute and display the results for each input
    for n in n_values:
        result = calculator.compute_sum_p_k(n)
        print(f"The sum of p_3({n}) to p_{n}({n}) is {result}")
```

Listing 2: Cache Memoization for $p(n)$.

4 Validate partition of N in K

Validating the results of $p(n, k)$ against Euler's identity for random inputs without memoization. For large numbers, it is recommended to use memoization and caching to store results.

4.1 $p(n, k)$ against Euler's identity for random inputs without memoization

```python
import random
from math import sin, pi
from tabulate import tabulate
import numpy as np

q = lambda n, k: n // k if k > 0 else 0

def p_of_n_k(n, k):
    if k == 3:
        q_3 = q(n, 3)
        s = 0
        for j in range(1, q_3 + 1):
            s += q(n - j, 2) - (j - 1)
        return s
    if k == 2:
        return q(n, 2)
    if k == 1:
        return 1
    q_k_of_n = q(n, k)
    delta = int((sin(pi * (k - 1) * 0.5)) ** 2)  # Compute delta
    sm = 0
    for u in range(q_k_of_n - delta + 1):
        sm += p_of_n_k(n - k * u - 1, k - 1)
    return sm

def compute_euler_identity(N, K):
    p = np.zeros((N + 1, N + 1), dtype=int)
    for i in range(N + 1):
        p[i, 1] = 1
        p[i, i] = 1
    for n in range(2, N + 1):
        for k in range(2, n + 1):
            p[n, k] = p[n - 1, k - 1] + p[n - k, k]
    sum_of_row_N = np.sum(p[N, :])  # Sums over all columns of row N
    p_N_K = p[N][K]
    return sum_of_row_N, p_N_K

def generate_random_values(num_tests=100, max_n=100):
    values = []
    for _ in range(num_tests):
        N = random.randint(1, max_n)
        K = random.randint(1, N)
        values.append((N, K))
    return values

```

```python
def validate(N, K):
    _, euler_value = compute_euler_identity(N, K)
    custom_value = p_of_n_k(N, K)
    return euler_value, custom_value, euler_value == custom_value

if __name__ == "__main__":
    max_n = 100
    num_tests = 501  # To avoid repetition num_tests <= max_n^2
    test_values = generate_random_values(num_tests, max_n)
    results = []
    print(f"Running {num_tests} validations...")
    for N, K in test_values:
        euler_value, custom_value, is_equal = validate(N, K)
        # Format values to avoid scientific notation
        euler_value_fmt = f"{euler_value:.0f}" if euler_value >= 1e5 else str(
            euler_value)
        custom_value_fmt = f"{custom_value:.0f}" if custom_value >= 1e5 else
            str(custom_value)
        results.append([N, K, euler_value_fmt, custom_value_fmt, "Pass" if
            is_equal else "Fail"])
    # Print results in tabular format
    headers = ["N", "K", "Euler Value", "Custom Value", "Status"]
    print(tabulate(results, headers=headers, tablefmt="grid"))
    all_tests_passed = all(row[4] == "Pass" for row in results)
    if all_tests_passed:
        print("\nAll tests passed! The formula matches Euler's identity.")
    else:
        print("\nSome tests failed. Investigate the discrepancy.")
```

Listing 3: $p(n, k)$ against Euler's identity for random inputs without memoization.

5 Conclusion

In this study, we established how the partition of N into two parts $P(n, 2)$ relates to the quotient function. Subsequently, we derived $p_3(n)$ as a series of sums of $p_2(n)$ or $Q_2(n)$, and extended this to $p_4(n)$ as a series of $p_3(n)$. This iterative approach was used to formulate a general expression for $p_k(n)$. The results were validated against Euler's identity using Python, and the recursion was optimized through caching and memoization. All tests were passed successfully.

References

[1] B. C. Berndt, *Ramanujan's Notebooks, Part III*, Springer, 1991.

[2] Equation Number 59 https://mathworld.wolfram.com/PartitionFunctionP.html