

# The Partition Function and the Pentagonal Number Theorem

Manuel Eberl

February 6, 2026

## Abstract

The partition function  $p(n)$  [2, A000041] gives the number of ways to write a non-negative integer  $n$  as a sum of positive integers, without taking order into account.

This entry uses the Jacobi Triple Product (already available in the AFP) to give a short proof of the *Pentagonal Number Theorem*, which is the statement that the generating function  $F(X) = \sum_{n \geq 0} p(n)X^n$  of the partition function satisfies:

$$F(X)^{-1} = \sum_{k \in \mathbb{Z}} (-1)^k X^{k(3k-1)/2} = 1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + \dots$$

The numbers  $g_k = \frac{1}{2}k(3k-1)$  appearing in the exponents are the *generalised pentagonal numbers* [2, A001318].

As further corollaries of this, an upper bound for  $p(n)$  and the recurrence relation

$$p(n) = \sum_{\substack{k \in \mathbb{Z} \setminus \{0\} \\ g_k \leq n}} (-1)^{k+1} p(n - g_k)$$

are proved. The latter also yields an algorithm to compute the numbers  $p(0), \dots, p(n)$  simultaneously in time roughly  $n^{2+o(1)}$ . This algorithm is implemented and proved correct at the end of this entry using the Imperative-HOL Refinement Framework.

# Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Generalised pentagonal numbers</b>             | <b>3</b> |
| <b>2</b> | <b>The partition function</b>                     | <b>6</b> |
| 2.1      | Definition . . . . .                              | 6        |
| 2.2      | Generating function . . . . .                     | 7        |
| 2.3      | The Pentagonal Number Theorem . . . . .           | 7        |
| 2.3.1    | The analytic version . . . . .                    | 8        |
| 2.3.2    | The formal power series version . . . . .         | 9        |
| 2.4      | A recurrence for the partition function . . . . . | 10       |
| 2.5      | Upper bound . . . . .                             | 10       |
| 2.6      | Efficient implementation . . . . .                | 11       |
| 2.6.1    | The first sum . . . . .                           | 12       |
| 2.6.2    | The second sum . . . . .                          | 15       |
| 2.6.3    | Computing the next number . . . . .               | 15       |
| 2.6.4    | The full algorithm . . . . .                      | 16       |

# 1 Generalised pentagonal numbers

```
theory Pentagonal_Numbers
  imports Main "HOL-Library.FuncSet"
begin
```

The pentagonal numbers are defined as a sequence of natural numbers  $(g_k)_{n \in \mathbb{N}}$  with  $g_k = \frac{1}{2}k(3k - 1)$ . Visually, they count the number of spheres needed to fill a pentagon where each side consists of  $k$  spheres – similarly to how  $k^2$  is the number of spheres needed to fill up a square with  $k$  spheres on each side.

We define the *generalised* pentagonal numbers, where the only difference is that  $k$  may also be negative [2, A001318].

The function  $g_k$  (with  $k \in \mathbb{Z}$ ) is injective, since we have:

$$g_0 < g_1 < g_{-1} < g_2 < g_{-2} < g_3 < g_{-3} < \dots$$

```
definition pent_num :: "int  $\Rightarrow$  nat" where
  "pent_num k = nat (k * (3 * k - 1) div 2)"
```

```
definition pent_nums :: "nat set" where
  "pent_nums = range pent_num"
```

```
lemma pent_num_0 [simp]: "pent_num 0 = 0"
  <proof>
```

```
lemma pent_num_in_pent_nums [intro]: "pent_num k  $\in$  pent_nums"
  <proof>
```

```
lemma twice_pent_num_eq: "2 * int (pent_num k) = k * (3 * k - 1)"
  <proof>
```

```
lemma strict_mono_pent_num:
  assumes "0  $\leq$  m" "m < n"
  shows "pent_num m < pent_num n"
  <proof>
```

```
lemma pent_num_less_iff_nonneg:
  assumes "i  $\geq$  0" "j  $\geq$  0"
  shows "pent_num i < pent_num j  $\longleftrightarrow$  i < j"
  <proof>
```

```
lemma pent_num_le_iff_nonneg:
  assumes "i  $\geq$  0" "j  $\geq$  0"
  shows "pent_num i  $\leq$  pent_num j  $\longleftrightarrow$  i  $\leq$  j"
  <proof>
```

```
lemma mono_pent_num:
```

```

    assumes "0 ≤ m" "m ≤ n"
    shows "pent_num m ≤ pent_num n"
    ⟨proof⟩

lemma strict_antimono_pent_num:
    assumes "m < n" "n ≤ 0"
    shows "pent_num m > pent_num n"
    ⟨proof⟩

lemma pent_num_less_iff_nonpos:
    assumes "i ≤ 0" "j ≤ 0"
    shows "pent_num i < pent_num j ↔ i > j"
    ⟨proof⟩

lemma pent_num_le_iff_nonpos:
    assumes "i ≤ 0" "j ≤ 0"
    shows "pent_num i ≤ pent_num j ↔ i ≥ j"
    ⟨proof⟩

lemma antimono_pent_num:
    assumes "m ≤ n" "n ≤ 0"
    shows "pent_num m ≥ pent_num n"
    ⟨proof⟩

lemma pent_num_uminus: "int (pent_num (-n)) = int (pent_num n) + n"
    ⟨proof⟩

lemma pent_num_uminus': "pent_num (-int n) = pent_num (int n) + n"
    ⟨proof⟩

lemma pent_num_neg_increment: "int (pent_num (n + 1)) = int (pent_num
(-n)) + 2 * n + 1"
    ⟨proof⟩

lemma pent_num_increment: "int (pent_num (n + 1)) = int (pent_num n)
+ 3 * n + 1"
    ⟨proof⟩

lemma pent_num_increment': "pent_num (int n + 1) = pent_num (int n) +
3 * n + 1"
    ⟨proof⟩

lemma pent_num_decrement: "int (pent_num (n - 1)) = int (pent_num n)
- 3 * n + 2"
    ⟨proof⟩

lemma pent_num_decrement': "pent_num (int n - 1) = pent_num (int n) +
2 - 3 * n"

```

*<proof>*

**lemma** *pent\_num\_less\_pent\_num\_neg*:  
 **assumes** "n > 0"  
 **shows** "pent\_num n < pent\_num (-n)"  
*<proof>*

**lemma** *pent\_num\_neg\_less\_pent\_num\_increment*:  
 **assumes** "n ≥ 0"  
 **shows** "pent\_num (-n) < pent\_num (n+1)"  
*<proof>*

**lemma** *inj\_pent\_num\_aux*:  
 **assumes** "m < 0" "n > 0" "pent\_num m = pent\_num n"  
 **shows** False  
*<proof>*

**lemma** *inj\_pent\_num*: "inj pent\_num"  
*<proof>*

**lemma** *pent\_num\_eq\_iff*: "pent\_num m = pent\_num n  $\longleftrightarrow$  m = n"  
*<proof>*

**definition** *inv\_pent\_num* :: "nat  $\Rightarrow$  int" **where**  
 "inv\_pent\_num n = (if n  $\in$  pent\_nums then THE k. pent\_num k = n else 0)"

**lemma** *pent\_num\_inv\_pent\_num*:  
 **assumes** "n  $\in$  pent\_nums"  
 **shows** "pent\_num (inv\_pent\_num n) = n"  
*<proof>*

**lemma** *inv\_pent\_num\_pent\_num [simp]*: "inv\_pent\_num (pent\_num k) = k"  
*<proof>*

**lemma** *inv\_pent\_num\_eqI*: "pent\_num k = n  $\implies$  inv\_pent\_num n = k"  
*<proof>*

**lemma** *pent\_num\_eq\_0\_iff*: "pent\_num k = 0  $\longleftrightarrow$  k = 0"  
*<proof>*

**lemma** *pent\_num\_pos\_iff*: "pent\_num k > 0  $\longleftrightarrow$  k  $\neq$  0"  
*<proof>*

An obvious but useful fact: only a finite number of pentagonal numbers are  $\leq k$  for any given  $k$ . In fact the number of pentagonal numbers  $\leq k$  is  $O(\sqrt{k})$ , but we will not show this here.

**lemma** *finite\_pent\_num\_le*: "finite {i. pent\_num i  $\leq$  k}"

*<proof>*

**lemma** *finite\_pent\_num\_minus\_le*: "finite {i. pent\_num (-i) ≤ k}"

*<proof>*

**end**

## 2 The partition function

**theory** *Partition\_Function*

**imports**

*Pentagonal\_Numbers*

*"Theta\_Functions.Jacobi\_Triple\_Product"*

*"Card\_Number\_Partitions.Card\_Number\_Partitions"*

**begin**

### 2.1 Definition

In the AFP, there is already a definition of the restricted partition number  $p_k(n)$ , which counts the number of ways to write the integer  $n$  as a sum of exactly  $k$  positive integers (without taking order into account). More formally, it counts the number of multisets  $X$  such that  $X \subseteq \mathbb{N} \setminus \{0\}$  and  $|X| = k$  and  $\sum X = n$ .

We now define the partition function  $p(n)$ , which counts the number of unrestricted number partitions, i.e. it removes the condition that there be exactly  $k$  parts [2, A000041]. In other words:  $p(n) = \sum_{k \leq n} p_k(n)$ .

**definition** *Partition'* :: "nat ⇒ nat"

**where** "*Partition'* n = ( $\sum_{k \leq n} \text{Partition } n \ k$ )"

**lemma** *Partition'\_0 [simp]*: "*Partition'* 0 = 1"

*<proof>*

**lemma** *Partition'\_eq\_card*: "*Partition'* n = card {f. f partitions n}"

*<proof>*

**lemma** *Partition'\_pos*: "*Partition'* n > 0"

*<proof>*

**lemma** *Partition'\_Suc\_gt*:

**assumes** "0 < m"

**shows** "*Partition'* m < *Partition'* (Suc m)"

*<proof>*

**lemma** *Partition'\_strict\_mono*:

**assumes** "0 < m" "m < n"

**shows** "*Partition'* m < *Partition'* n"

*<proof>*

```

lemma Partition'_mono:
  assumes "m ≤ n"
  shows "Partition' m ≤ Partition' n"
  ⟨proof⟩

```

## 2.2 Generating function

Next, we will move on to derive a closed-form expression for the generating function  $\sum_{n \geq 0} p(n)X^n$  of  $p(n)$  in terms of an infinite product.

The first step is the following lemma: there is a one-to-one correspondence between the number partitions of  $n$  and the ways to distribute  $n$  balls onto the natural numbers such that the number of balls in the  $i$ -th bin is a multiple of  $i + 1$ .

```

lemma bij_betw_multisets_of_size_partitions:
  "bij_betw (λX i. if i = 0 then 0 else count X (i - 1) div i)
    {X ∈ multisets_of_size UNIV n. ∀ i. Suc i dvd count X i} {h. h partitions
  n}"
  ⟨proof⟩

```

We can now easily derive our closed-form expression, namely:

$$\sum_{n \geq 0} p(n)X^n = \prod_{k \geq 1} \frac{1}{1 - X^k} = \frac{1}{\phi(X)}$$

where  $\phi(X)$  is Euler's function (not to be confused with Euler's totient function  $\varphi(n)$ ).

```

lemma has_prod_Partition'_aux:
  "(λk. ∏ i ≤ k. 1 / (1 - fps_X ^ Suc i)) → Abs_fps (λn. of_nat (Partition'
  n)
  :: 'a :: {field, t2_space})" (is "?F → ?G")
  ⟨proof⟩

```

```

theorem has_prod_Partition':
  "(λi. 1 / (1 - fps_X ^ Suc i)) has_prod Abs_fps (λn. of_nat (Partition'
  n)
  :: 'a :: {field, t2_space})" (is "?F has_prod ?G")
  ⟨proof⟩

```

end

## 2.3 The Pentagonal Number Theorem

```

theory Pentagonal_Number_Theorem
imports
  Partition_Function
  Theta_Functions.Jacobi_Triple_Product
begin

```

### 2.3.1 The analytic version

The analytic version of the pentagonal number theorem states that:

$$\phi(q) = (q; q)_{\infty} = f(-q, -q^2) = \theta(-\sqrt{q}, q\sqrt{q})$$

where  $f(a, b)$  denotes the Ramanujan theta function and  $\theta(w, q)$  denotes the Jacobi theta function (both in terms of the nome).

This is an easily corollary from the Jacobi triple product, which is already in the AFP.

```

theorem pentagonal_number_theorem_complex:
  fixes q :: complex
  assumes q: "norm q < 1"
  shows "euler_phi q = ramanujan_theta (-q) (-(q^2))"
  <proof>
  include qepochhammer_inf_notation
  <proof>

```

```

lemma pentagonal_number_theorem_real:
  fixes q :: real
  assumes q: "|q| < 1"
  shows "euler_phi q = ramanujan_theta (-q) (-(q^2))"
  <proof>
  include qepochhammer_inf_notation
  <proof>

```

As a corollary, we get the following power series representation for  $\phi(q)$ .

```

lemma has_sum_euler_phi_complex:
  fixes q :: complex
  assumes q: "norm q < 1"
  shows "((λk. (-1) powi k * q ^ pent_num k) has_sum euler_phi q) UNIV"
  <proof>
  include qepochhammer_inf_notation
  <proof>

```

The following is the more explicit form of the Pentagonal Number Theorem usually found in textbooks:

$$\prod_{n=1}^{\infty} (1 - q^n) = \sum_{k=-\infty}^{\infty} (-1)^k q^{k(3k-1)/2}$$

The exponents  $g_k$  are the generalised pentagonal numbers we defined earlier.

```

lemma pentagonal_number_theorem_complex':
  fixes q :: complex
  assumes q: "norm q < 1"
  shows "abs_convergent_prod (λn. 1 - q^(n+1))" (is ?th1)
  and "(λk. (-1) powi k * q ^ pent_num k) abs_summable_on UNIV" (is
  ?th2)

```

```

    and "( $\prod_{n::\text{nat.}} 1 - q^{(n+1)}$ ) = ( $\sum_{\infty}(k::\text{int}). (-1)^{\text{powi } k} * q^{\text{pent\_num } k}$ )" (is ?th3)
  <proof>
    include qepochhammer_inf_notation
  <proof>

```

### 2.3.2 The formal power series version

We now rephrase the above analytic result in terms of formal power series. First, we give the generating function of  $\phi(q)$  in terms of the generalised pentagonal numbers.

```

definition fps_euler_phi :: "'a :: ring_1 fps" where
  "fps_euler_phi =
    Abs_fps ( $\lambda n.$  if  $n \in \text{pent\_nums}$  then if even ( $\text{inv\_pent\_num } n$ ) then
    1 else -1 else 0)"

```

```

lemma sums_euler_phi_complex:
  fixes q :: complex and f :: "nat  $\Rightarrow$  complex"
  assumes q: "norm q < 1"
  defines "f  $\equiv$  ( $\lambda n.$  if  $n \in \text{pent\_nums}$  then if even ( $\text{inv\_pent\_num } n$ ) then
  1 else -1 else 0)"
  shows "( $\lambda k.$  f k * q ^ k) sums euler_phi q"
<proof>

```

```

theorem has_fps_expansion_euler_phi_complex [fps_expansion_intros]:
  "euler_phi has_fps_expansion (fps_euler_phi :: complex fps)"
<proof>

```

The formal power series version of the pentagonal number theorem states that the power series  $\sum_{n \geq 0} p(n)X^n$  and  $\sum_{k \in \mathbb{Z}} (-1)^k X^{k(3k-1)/2}$  are multiplicative inverses of one another.

We just determined that  $\sum_{n \geq 0} p(n)q^n = \prod_{k \geq 1} \frac{1}{1-q^k}$  holds analytically, so all we have to do is to lift this to the level of formal power series.

```

theorem fps_Partition'_eq:
  "Abs_fps ( $\lambda n.$  of_nat (Partition' n) :: complex) = inverse fps_euler_phi"
<proof>

```

```

lemma fps_nth_euler_phi_0 [simp]: "fps_nth fps_euler_phi 0 = 1"
<proof>

```

```

lemma has_fps_expansion_inverse_euler_phi_complex:
  "( $\lambda q.$  1 / euler_phi q :: complex)
  has_fps_expansion Abs_fps ( $\lambda n.$  of_nat (Partition' n))"
<proof>

```

```

lemma conv_radius_Partition':
  "conv_radius ( $\lambda n.$  of_nat (Partition' n) :: 'a :: {banach,real_normed_algebra_1})
   $\geq 1$ "

```

*<proof>*

**lemma** *sums\_inverse\_euler\_phi\_complex:*

**assumes** "norm z < 1"

**shows** "( $\lambda k.$  of\_nat (Partition' k) \* z ^ k :: complex) sums (1 / euler\_phi z)"

*<proof>*

**lemma** *sums\_inverse\_euler\_phi\_real:*

**assumes** "|x| < (1::real)"

**shows** "( $\lambda k.$  of\_nat (Partition' k) \* x ^ k) sums (1 / euler\_phi x)"

*<proof>*

## 2.4 A recurrence for the partition function

A direct consequence of this is the following recurrence for the partition numbers:

$$p(n) = \sum_{\substack{k \in \mathbb{Z} \\ g_k \in [1, n]}} p(n - g_k)$$

where  $n > 0$  and  $g_k = k(3k - 1)/2$  are the generalised pentagonal numbers. Note that the sum on the right-hand side has  $O(\sqrt{n})$  terms.

**theorem** *Partition'\_recurrence:*

**assumes**  $n: "n > 0"$

**shows** "int (Partition' n) =

$(\sum i \mid i \in \{1..n\} \wedge i \in \text{pent\_nums}.$

$(\text{if even (inv\_pent\_num } i) \text{ then } -1 \text{ else } 1) * \text{int (Partition'}$

$(n - i)))"$

*<proof>*

## 2.5 Upper bound

Lastly, we prove an upper bound for the partition function based on a lower bound for  $\phi(x)$ .

We follow Apostol's presentation of the proof by van Lint [1, Theorem 14.5].

The basic idea is to recall that

$$\sum_{k \geq 0} p(k)x^k = \frac{1}{\phi(x)}$$

and note that due to the monotonicity of  $\phi$  we have:

$$\sum_{k \geq 0} p(k)x^k \geq \sum_{k \geq n} p(k)x^k = p(n) \frac{x^n}{1-x}$$

After combining the two, we take logarithms and rearrange to get:

$$\log p(n) \leq -\log \phi(x) - n \log x + \log(1-x)$$

We then use the bound  $\phi(x) \geq -\frac{\pi^2}{6} \frac{x}{1-x}$  and make the change of variables  $t = \frac{1-x}{x}$  and some other small estimates to get:

$$\log p(n) \leq \frac{\pi^2}{6t} + (n-1)t + \log t$$

We then plug in the optimal value for  $t$  and make some more final estimates to get the final result:

$$p(n) \leq \frac{\pi \exp(K\sqrt{n})}{\sqrt{6(n-1)}} \quad \text{where } K = \sqrt{\frac{2}{3}}\pi$$

Note that asymptotically,  $p(n) \sim \exp(K\sqrt{n})/(4\sqrt{3}n)$ , so the inequality is off by roughly a factor of  $3\sqrt{n}$ , which is relatively small considering that  $p(n)$  grows superpolynomially.

```

theorem Partition'_le:
  assumes n: "n > 1"
  shows "real (Partition' n) ≤ pi * exp (pi * sqrt (2 / 3 * real n))
/ sqrt (6 * (real n - 1))"
  ⟨proof⟩

```

**end**

## 2.6 Efficient implementation

```

theory Partition_Number_Imperative
imports
  Pentagonal_Number_Theorem
  "Refine_Monadic.Refine_Monadic"
  "Refine_Imperative_HOL.IICF"
begin

```

Using the aforementioned recurrence, we can easily compute  $p(n)$  given  $p(0), \dots, p(n-1)$ . We split the sum into one sum for  $k > 0$  and one for  $k < 0$ . In the first one, we start with  $k = 1$ , add  $(-1)^{k+1}p(n-g_k)$  to our accumulator, and increment  $k$ , until  $g_k > n$ . The second sum is analogous except that we start with  $k = -1$  and decrement in every step.

We do not show it, but the number of terms to sum over is roughly  $\frac{\sqrt{24n}}{3} \approx 1.633\sqrt{n} \in O(n^{1/2})$ . Since  $p(n)$  has roughly  $3.7\sqrt{n} \in \Theta(n^{1/2})$  bits, this means that in order to compute  $p(0), \dots, p(n)$  we have to do add  $O(n^{3/2})$  numbers with up to  $O(n^{1/2})$  bits each, which gives the total algorithm a running time of roughly  $O(n^2)$ . Since the output itself has  $O(n^{3/2})$  bits, this is not too bad.

What we are doing here to compute  $p(0), \dots, p(n)$  is in fact to take the power series of  $\phi(X)$  and compute  $(1/\phi(X)) \bmod X^{n+1}$  (i.e. the first  $n$  coefficients of its multiplicative inverse) in the naïve way. This works very well, since

the power series of  $\phi(X)$  is easy to compute and also very sparse (i.e. out of the first  $n$  coefficients, only  $O(\sqrt{n})$  are non-zero).

One way to get a better running time would be to use a more sophisticated method of computing reciprocals of power series, e.g. by Hensel lifting. In principle, this is very easy to implement and prove correct, but to be faster than the present method in practice it requires a fast algorithm for polynomial multiplication: if used with a polynomial multiplication algorithm that uses  $O(M(n))$  coefficient operations (i.e. addition/multiplication of coefficients) for a polynomial of degree  $n$ , Hensel lifting also uses  $O(M(n))$  coefficient operations. Since the coefficients have size up to  $O(n^{1/2})$  bits, this gives us a total running time of  $O(M(n)n^{1/2})$ . For an FFT-based  $O(n \log n)$  multiplication algorithm, we therefore have the quasi-optimal  $O(n^{3/2} \log(n))$ . However, for less sophisticated algorithms such as Karatsuba multiplication or naïve multiplication, this actually ends up being less efficient than the above approach.

Note that if one only wants to compute a single  $p(n)$ , this can be done much faster, namely in  $O(\sqrt{n})$  time, using Rademacher's sum. However, this requires a significant additional amount of mathematics and, more importantly, infrastructure to approximate transcendental functions to high precision.

```
definition Partition'_list :: "nat  $\Rightarrow$  int list"
  where "Partition'_list n = map (int  $\circ$  Partition') [0.. $n+1$ ]"
```

### 2.6.1 The first sum

```
definition partition_impl1_inner1_aux where
  "partition_impl1_inner1_aux n ps k =
    ( $\sum$  i=1.. $k$ . (if even i then -1 else 1) * ps ! nat (n - int (pent_num i)))"
```

```
definition partition_impl1_inner1_aux' where
  "partition_impl1_inner1_aux' n ps =
    ( $\sum$  i | i  $\geq$  1  $\wedge$  int (pent_num i)  $\leq$  n. (if even i then -1 else 1)
    * ps ! nat (n - int (pent_num i)))"
```

```
definition partition_impl1_inner1_invar where
  "partition_impl1_inner1_invar n ps acc0 = ( $\lambda$ (k,acc).
    k > 0  $\wedge$  int (pent_num (k - 1))  $\leq$  n  $\wedge$ 
    acc = acc0 + partition_impl1_inner1_aux n ps k)"
```

```
definition partition_impl1_inner1 :: "int  $\Rightarrow$  int list  $\Rightarrow$  int  $\Rightarrow$  int nres"
where
  "partition_impl1_inner1 n ps acc = do {
    (k,acc)  $\leftarrow$ 
    WHILE_T partition_impl1_inner1_invar n ps acc ( $\lambda$ (k,_). int (pent_num
    k)  $\leq$  n) ( $\lambda$ (k,acc). do {
```

```

    ASSERT (n - int (pent_num k) ∈ {0..

```

shows "partition\_impl2\_inner1 n ps acc  $\leq$  SPEC ( $\lambda$ acc'. acc' = acc + partition\_impl1\_inner1\_aux' n ps)"  
 <proof>

**definition** partition\_impl1\_inner2\_aux where  
 "partition\_impl1\_inner2\_aux n ps k =  
 ( $\sum_{i=1..<k. (if\ even\ i\ then\ -1\ else\ 1) * ps ! nat (n - int (pent\_num (-i)))}$ )"

**definition** partition\_impl1\_inner2\_aux' where  
 "partition\_impl1\_inner2\_aux' n ps =  
 ( $\sum_{i \mid i \geq 1 \wedge int (pent\_num (-i)) \leq n. (if\ even\ i\ then\ -1\ else\ 1) * ps ! nat (n - int (pent\_num (-i)))}$ )"

**definition** partition\_impl1\_inner2\_invar where  
 "partition\_impl1\_inner2\_invar n ps acc0 = ( $\lambda(k, acc).$   
 $k > 0 \wedge int (pent\_num (-(k - 1))) \leq n \wedge$   
 $acc = acc0 + partition\_impl1\_inner2\_aux\ n\ ps\ k$ )"

**definition** partition\_impl1\_inner2 :: "int  $\Rightarrow$  int list  $\Rightarrow$  int  $\Rightarrow$  int nres"  
 where  
 "partition\_impl1\_inner2 n ps acc = do {  
 (k, acc)  $\leftarrow$   
 WHILE\_T partition\_impl1\_inner2\_invar n ps acc ( $\lambda(k, _). int (pent\_num (-k)) \leq n$ ) ( $\lambda(k, acc). do \{$   
 ASSERT ( $n - int (pent\_num (-k)) \in \{0..<int (length\ ps)\}$ );  
 let x = ps ! nat (n - int (pent\\_num (-k)));  
 RETURN (k + 1, if even k then acc - x else acc + x)  
 })  
 (1, acc);  
 RETURN acc  
 }"

**lemma** partition\_impl1\_inner2\_aux\_rec:  
 assumes k: "k  $\geq$  1"  
 shows "partition\_impl1\_inner2\_aux n ps (1 + k) =  
 partition\_impl1\_inner2\_aux n ps k + (if even k then -1 else 1) \* ps ! nat (n - int (pent\\_num (-k)))"  
 <proof>

**lemma** partition\_impl1\_inner2\_final:  
 assumes "partition\_impl1\_inner2\_invar n ps acc (k, acc'" "int (pent\\_num (-k)) > n"  
 shows "acc' = acc + partition\_impl1\_inner2\_aux' n ps"  
 <proof>

**lemma** partition\_impl1\_inner2\_correct [refine\_vcg]:  
 assumes "n  $\in \{1..int (length\ ps)\}$ "

```

shows "partition_impl1_inner2 n ps acc ≤
      SPEC (λacc'. acc' = acc + partition_impl1_inner2_aux' n ps)"
⟨proof⟩

```

### 2.6.2 The second sum

```

definition partition_impl2_inner2 :: "int ⇒ int list ⇒ int ⇒ int nres"
where

```

```

"partition_impl2_inner2 n ps acc = do {
  (_, _, acc') ←
    WHILE_T (λ(k,i,_). i ≥ 0) (λ(k,i,acc). do {
      ASSERT (nat i < length ps);
      let x = ps ! nat i;
      RETURN (k + 1, i - (3 * k + 2), if even k then acc - x else acc
+ x)
    })
  (1, n - 2, acc);
  RETURN acc'
}"

```

```

lemma partition_impl2_inner2_refine [refine]:
  "partition_impl2_inner2 n ps acc ≤ ↓Id (partition_impl1_inner2 n ps
acc)"
⟨proof⟩

```

### 2.6.3 Computing the next number

```

definition partition_impl2_inner where
  "partition_impl2_inner n ps =
  do {acc ← partition_impl2_inner1 n ps 0;
  partition_impl2_inner2 n ps acc}"

```

```

lemma partition_impl2_inner2_correct [refine_vcg]:
  assumes "n ∈ {1..int (length ps)}"
  shows "partition_impl2_inner2 n ps acc ≤ SPEC (λacc'. acc' = acc
+ partition_impl1_inner2_aux' n ps)"
⟨proof⟩

```

```

lemma partition_impl2_inner_correct_aux:
  assumes "n ∈ {1..int (length ps)}" "∧i. int i < n ⇒ ps ! i = int
(Partition' i)"
  shows "partition_impl1_inner1_aux' (int n) ps + partition_impl1_inner2_aux'
(int n) ps =
  int (Partition' n)" (is "?lhs = ?rhs")
⟨proof⟩

```

```

lemma partition_impl2_inner_correct [refine_vcg]:
  assumes "n ∈ {1..int (length ps)}" "∧i. int i < n ⇒ ps ! i = int
(Partition' i)"
  shows "partition_impl2_inner n ps ≤ SPEC (λx. x = Partition' n)"

```

*<proof>*

#### 2.6.4 The full algorithm

**definition** `partition_impl2` where

```
"partition_impl2 n = do {
  ps ← RETURN (op_array_replicate (n+1) 0);
  ASSERT (length ps > 0);
  let ps' = ps[0 := 1];
  (_, ps'') ←
    WHILE_T λ(m,ps). m∈{1..n+1} ∧ length ps = n+1 ∧ (∀i<m. ps ! i = int (Partition' i))
      (λ(m,ps). m ≤ n)
      (λ(m,ps).
        do {
          x ← partition_impl2_inner m ps;
          ASSERT (m < length ps);
          RETURN (m+1, ps[m := x])
        }) (1, ps');
  RETURN ps''
}"
```

**lemma** `partition_impl2_correct` [`refine_vcg`]:

```
"partition_impl2 n ≤ SPEC (λxs. xs = Partition'_list n)"
<proof>
```

**lemma** `param_dvd_nat` [`param`, `sepref_import_param`]:

```
"((dvd), (dvd)) ∈ nat_rel → nat_rel → bool_rel"
"((dvd), (dvd)) ∈ int_rel → int_rel → bool_rel"
<proof>
```

**sepref\_definition** `partition_impl3` is

```
"partition_impl2" :: "nat_assnd →a array_assn int_assn"
<proof>
```

**lemma** `partition_impl3_correct'`:

```
"(partition_impl3, λn. RETURN (Partition'_list n)) ∈ nat_assnd →a array_assn
int_assn"
<proof>
```

**theorem** `partition_impl3_correct`:

```
"<nat_assn n n> partition_impl3 n <array_assn int_assn (Partition'_list
n)>t"
<proof>
```

**end**

**theory** `Partition_Number_Imperative_Test`

**imports**

```
Partition_Number_Imperative
"HOL-Library.Code_Target_Numeral"
```

**begin**

```
definition "partition_impl3_test n =  
  do {  
    a ← partition_impl3 (nat_of_integer n);  
    xs ← Array.freeze a;  
    return (map integer_of_int xs)  
  }"
```

⟨ML⟩

**end**

## References

- [1] T. M. Apostol. *Introduction to Analytic Number Theory*. Undergraduate Texts in Mathematics. Springer, 1976.
- [2] OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2025. Published electronically at <http://oeis.org>.