

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

1	Generalised pentagonal numbers	3
2	The partition function	9
2.1	Definition	9
2.2	Generating function	11
2.3	The Pentagonal Number Theorem	17
2.3.1	The analytic version	18
2.3.2	The formal power series version	22
2.4	A recurrence for the partition function	25
2.5	Upper bound	26
2.6	Efficient implementation	30
2.6.1	The first sum	31
2.6.2	The second sum	36
2.6.3	Computing the next number	37
2.6.4	The full algorithm	39

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"
  by (simp add: pent_num_def)
```

```
lemma pent_num_in_pent_nums [intro]: "pent_num k  $\in$  pent_nums"
  unfolding pent_nums_def by blast
```

```
lemma twice_pent_num_eq: "2 * int (pent_num k) = k * (3 * k - 1)"
proof -
  have "k * (3 * k - 1)  $\geq$  0"
  proof (cases "k > 0")
    case True
      thus ?thesis
        by (intro mult_nonneg_nonneg) auto
    next
      case False
      thus ?thesis
        by (intro mult_nonpos_nonpos) auto
  qed
  thus ?thesis
    unfolding pent_num_def by simp
qed
```

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

```

proof -
  have "2 * int (pent_num m) < 2 * int (pent_num n)"
  proof -
    have "m * (3 * m - 1) ≤ m * (3 * n - 1)"
      by (intro mult_left_mono) (use assms in auto)
    also have "... < n * (3 * n - 1)"
      by (intro mult_strict_right_mono) (use assms in auto)
    finally show ?thesis
      unfolding twice_pent_num_eq by simp
  qed
  thus ?thesis
    by simp
qed

lemma pent_num_less_iff_nonneg:
  assumes "i ≥ 0" "j ≥ 0"
  shows "pent_num i < pent_num j ↔ i < j"
  by (cases i j rule: linorder_cases)
    (use assms strict_mono_pent_num[of i j] strict_mono_pent_num[of j
i] in auto)

lemma pent_num_le_iff_nonneg:
  assumes "i ≥ 0" "j ≥ 0"
  shows "pent_num i ≤ pent_num j ↔ i ≤ j"
  by (cases i j rule: linorder_cases)
    (use assms strict_mono_pent_num[of i j] strict_mono_pent_num[of j
i] in auto)

lemma mono_pent_num:
  assumes "0 ≤ m" "m ≤ n"
  shows "pent_num m ≤ pent_num n"
  using strict_mono_pent_num[of m n] assms by (cases "m = n") auto

lemma strict_antimono_pent_num:
  assumes "m < n" "n ≤ 0"
  shows "pent_num m > pent_num n"
proof -
  have "2 * int (pent_num m) > 2 * int (pent_num n)"
  proof -
    have "n * (3 * n - 1) ≤ n * (3 * m - 1)"
      by (intro mult_left_mono_neg) (use assms in auto)
    also have "... < m * (3 * m - 1)"
      by (intro mult_strict_right_mono_neg) (use assms in auto)
    finally show ?thesis
      unfolding twice_pent_num_eq by simp
  qed
  thus ?thesis
    by simp
qed

```

```

lemma pent_num_less_iff_nonpos:
  assumes "i ≤ 0" "j ≤ 0"
  shows "pent_num i < pent_num j ↔ i > j"
  by (cases i j rule: linorder_cases)
    (use assms strict_antimono_pent_num[of i j] strict_antimono_pent_num[of
j i] in auto)

```

```

lemma pent_num_le_iff_nonpos:
  assumes "i ≤ 0" "j ≤ 0"
  shows "pent_num i ≤ pent_num j ↔ i ≥ j"
  by (cases i j rule: linorder_cases)
    (use assms strict_antimono_pent_num[of i j] strict_antimono_pent_num[of
j i] in auto)

```

```

lemma antimono_pent_num:
  assumes "m ≤ n" "n ≤ 0"
  shows "pent_num m ≥ pent_num n"
  using strict_antimono_pent_num[of m n] assms by (cases "m = n") auto

```

```

lemma pent_num_uminus: "int (pent_num (-n)) = int (pent_num n) + n"
proof -
  have "2 * (int (pent_num (-n)) - int (pent_num n)) = 2 * n"
    unfolding twice_pent_num_eq ring_distrib by (simp add: algebra_simps)
  thus ?thesis
    by simp
qed

```

```

lemma pent_num_uminus': "pent_num (-int n) = pent_num (int n) + n"
  using pent_num_uminus[of "int n"] by simp

```

```

lemma pent_num_neg_increment: "int (pent_num (n + 1)) = int (pent_num
(-n)) + 2 * n + 1"
proof -
  have "2 * (int (pent_num (n+1)) - int (pent_num (-n))) = 2 * (2 * n
+ 1)"
    unfolding twice_pent_num_eq ring_distrib by (simp add: algebra_simps)
  thus ?thesis
    by simp
qed

```

```

lemma pent_num_increment: "int (pent_num (n + 1)) = int (pent_num n)
+ 3 * n + 1"
proof -
  have "2 * (int (pent_num (n+1)) - int (pent_num n)) = 2 * (3 * n + 1)"
    unfolding twice_pent_num_eq ring_distrib by (simp add: algebra_simps)
  thus ?thesis
    by simp

```

qed

```
lemma pent_num_increment': "pent_num (int n + 1) = pent_num (int n) +
3 * n + 1"
  using pent_num_increment[of "int n"] by simp
```

```
lemma pent_num_decrement: "int (pent_num (n - 1)) = int (pent_num n)
- 3 * n + 2"
  using pent_num_increment[of "n-1"] by simp
```

```
lemma pent_num_decrement': "pent_num (int n - 1) = pent_num (int n) +
2 - 3 * n"
  using pent_num_decrement[of "int n"] by simp
```

```
lemma pent_num_less_pent_num_neg:
  assumes "n > 0"
  shows "pent_num n < pent_num (-n)"
proof -
  have "0 < n"
    using assms by simp
  also have "... = int (pent_num (-n)) - int (pent_num n)"
    by (simp add: pent_num_uminus)
  finally show ?thesis by simp
qed
```

```
lemma pent_num_neg_less_pent_num_increment:
  assumes "n ≥ 0"
  shows "pent_num (-n) < pent_num (n+1)"
proof -
  have "0 < 2 * n + 1"
    using assms by simp
  also have "... = int (pent_num (n + 1)) - int (pent_num (-n))"
    by (simp add: pent_num_uminus pent_num_increment)
  finally show ?thesis by simp
qed
```

```
lemma inj_pent_num_aux:
  assumes "m < 0" "n > 0" "pent_num m = pent_num n"
  shows False
proof (cases "-m ≥ n")
  case False
  have "pent_num m < pent_num (-m+1)"
    using pent_num_neg_less_pent_num_increment[of "-m"] assms by simp
  also have "pent_num (-m+1) ≤ pent_num n"
    by (rule mono_pent_num) (use assms False in auto)
  finally show False
    using assms by auto
next
  case True
```

```

have "pent_num n < pent_num (-n)"
  by (rule pent_num_less_pent_num_neg) (use assms in auto)
also have "... ≤ pent_num m"
  by (rule antimono_pent_num) (use assms True in auto)
finally show False
  using assms by auto
qed

lemma inj_pent_num: "inj pent_num"
proof
  fix m n :: int
  assume "pent_num m = pent_num n"
  hence False if "m ≠ n"
    using that
  proof (induction m n rule: linorder_wlog)
    case (le m n)
    hence "m < n"
      by auto
    have "m < 0" "n > 0"
      using strict_antimono_pent_num[of m n] strict_mono_pent_num[of m
n] <m < n> le.prem
      by linarith+
    thus False
      using inj_pent_num_aux[of m n] le.prem by simp
  qed (simp add: eq_commute)
  thus "m = n"
    by blast
qed

lemma pent_num_eq_iff: "pent_num m = pent_num n ↔ m = n"
  using inj_pent_num unfolding inj_on_def by blast

definition inv_pent_num :: "nat ⇒ int" where
  "inv_pent_num n = (if n ∈ pent_nums then THE k. pent_num k = n else
0)"

lemma pent_num_inv_pent_num:
  assumes "n ∈ pent_nums"
  shows "pent_num (inv_pent_num n) = n"
proof -
  have "∃ k. pent_num k = n"
    using assms by (auto simp: pent_nums_def)
  moreover have "k = k'" if "pent_num k = n" "pent_num k' = n" for k k'
    using inj_onD[OF inj_pent_num, of k k'] that by auto
  ultimately have "∃ !k. pent_num k = n"
    by blast
  from theI'[OF this] and assms show ?thesis
    by (simp add: inv_pent_num_def)

```

qed

```
lemma inv_pent_num_pent_num [simp]: "inv_pent_num (pent_num k) = k"
  by (metis pent_num_eq_iff pent_num_inv_pent_num pent_num_in_pent_nums)
```

```
lemma inv_pent_num_eqI: "pent_num k = n  $\implies$  inv_pent_num n = k"
  using inv_pent_num_pent_num by metis
```

```
lemma pent_num_eq_0_iff: "pent_num k = 0  $\longleftrightarrow$  k = 0"
  by (metis inv_pent_num_pent_num pent_num_0)
```

```
lemma pent_num_pos_iff: "pent_num k > 0  $\longleftrightarrow$  k  $\neq$  0"
  using pent_num_eq_0_iff[of k] by linarith
```

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 (rule finite_subset)
```

```
  show "{i. pent_num i  $\leq$  k}  $\subseteq$  {-int k..int k}"
```

```
  proof
```

```
    fix i assume i: "i  $\in$  {i. pent_num i  $\leq$  k}"
```

```
    have *: "k  $\leq$  pent_num (int k)"
```

```
    proof -
```

```
      have "2 * int k  $\leq$  int k * (3 * int k - 1)"
```

```
        by (cases "k = 0") auto
```

```
      also have "... = 2 * int (pent_num (int k))"
```

```
        by (simp add: twice_pent_num_eq)
```

```
      finally show "k  $\leq$  pent_num (int k)"
```

```
        by linarith
```

```
    qed
```

```
  have "i  $\leq$  int k"
```

```
  proof (cases "i  $\geq$  0")
```

```
    case True
```

```
    have "pent_num i  $\leq$  k"
```

```
      using i by simp
```

```
    also have "k  $\leq$  pent_num (int k)"
```

```
      by (fact *)
```

```
    finally show "i  $\leq$  int k"
```

```
      using True by (simp add: pent_num_le_iff_nonneg)
```

```
  qed auto
```

```
  moreover have "-int k  $\leq$  i"
```

```
  proof (cases "i  $\geq$  0")
```

```
    case False
```

```
    have "pent_num i  $\leq$  k"
```

```
      using i by simp
```

```

    also have "k ≤ pent_num (int k)"
      by fact
    also have "... ≤ pent_num (-int k)"
      by (subst pent_num_uminus') auto
    finally show "i ≥ -int k"
      using False by (simp add: pent_num_le_iff_nonpos)
qed auto

ultimately show "i ∈ {-int k..int k}"
  by auto
qed
qed auto

lemma finite_pent_num_minus_le: "finite {i. pent_num (-i) ≤ k}"
proof -
  have "bij_betw uminus {i. pent_num i ≤ k} {i. pent_num (-i) ≤ k}"
    by (rule bij_betwI[of _ _ _ uminus]) auto
  from bij_betw_finite[OF this] and finite_pent_num_le[of k] show ?thesis
    by simp
qed

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 = (∑ k ≤ n. Partition n k)"

```

```

lemma Partition'_0 [simp]: "Partition' 0 = 1"
  by (simp add: Partition'_def)

```

```

lemma Partition'_eq_card: "Partition' n = card {f. f partitions n}"
  using card_partitions[of n] by (simp add: Partition'_def)

lemma Partition'_pos: "Partition' n > 0"
proof -
  have *: "( $\lambda i. \text{if } i = 1 \text{ then } n \text{ else } 0$ ) partitions n"
    unfolding partitions_def
  proof
    have "( $\sum i \leq n. (\text{if } i = 1 \text{ then } n \text{ else } 0) * i$ ) = ( $\sum i \in (\text{if } n = 0 \text{ then } \{\} \text{ else } \{1::\text{nat}\}). n$ )"
      by (rule sum_mono_neutral_cong_right) (auto split: if_splits)
    also have "... = n"
      by auto
    finally show "( $\sum i \leq n. (\text{if } i = 1 \text{ then } n \text{ else } 0) * i$ ) = n" .
  qed auto
  have "Partition' n = card {f. f partitions n}"
    unfolding Partition'_eq_card ..
  also have "... > 0"
    by (subst card_gt_0_iff) (use * in <auto intro: finite_partitions>)
  finally show ?thesis .
qed

lemma Partition'_Suc_gt:
  assumes "0 < m"
  shows "Partition' m < Partition' (Suc m)"
proof -
  have "1 = Partition m 1"
    using assms Partition_parts1[of "m-1"] by simp
  also have "Partition m 1 = ( $\sum k \in \{0\}. \text{Partition } (m - k) (\text{Suc } k)$ )"
    by simp
  also have "( $\sum k \in \{0\}. \text{Partition } (m - k) (\text{Suc } k)$ )  $\leq$  ( $\sum k \leq m. \text{Partition } (m - k) (\text{Suc } k)$ )"
    by (intro sum_mono2) auto
  finally have pos: "( $\sum k \leq m. \text{Partition } (m - k) (\text{Suc } k)$ ) > 0"
    by linarith

  have "Partition' (Suc m) = ( $\sum k \leq \text{Suc } m. \text{Partition } (\text{Suc } m) k$ )"
    by (simp add: Partition'_def)
  also have "... = ( $\sum k \leq m. \text{Partition } (\text{Suc } m) (\text{Suc } k)$ )"
    by (subst sum.atMost_Suc_shift) simp
  also have "... = Partition' m + ( $\sum k \leq m. \text{Partition } (m - k) (\text{Suc } k)$ )"
    by (simp add: sum.distrib Partition'_def)
  finally have "Partition' (Suc m) = Partition' m + ( $\sum k \leq m. \text{Partition } (m - k) (\text{Suc } k)$ )" .
  with pos show ?thesis
    by linarith
qed

```

```

lemma Partition'_strict_mono:
  assumes "0 < m" "m < n"
  shows "Partition' m < Partition' n"
  using assms(2)
proof (induction n rule: less_induct)
  case (less n)
  show ?case
  proof (cases "n = Suc m")
    case True
    thus ?thesis
      using less.prem< m > 0> by (simp add: Partition'_Suc_gt)
  next
    case False
    have "Partition' m < Partition' (n - 1)"
      by (rule less.IH) (use False < m > 0> less.prem< m > 0> in auto)
    also have "... < Partition' (Suc (n - 1))"
      by (rule Partition'_Suc_gt) (use False < m > 0> less.prem< m > 0> in auto)
    also have "Suc (n - 1) = n"
      using False < m > 0> less.prem< m > 0> by simp
    finally show ?thesis .
  qed
qed

```

```

lemma Partition'_mono:
  assumes "m ≤ n"
  shows "Partition' m ≤ Partition' n"
  using Partition'_strict_mono assms sledgehammer
  by (metis Partition'_0 Partition'_pos bot_nat_0.extremum_strict
    less_one linorder_le_less_linear order_le_less)

```

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-once 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 -
  define f :: "nat multiset ⇒ nat ⇒ nat"
    where "f = (λX i. if i = 0 then 0 else count X (i - 1) div i)"
  define g :: "(nat ⇒ nat) ⇒ nat multiset"
    where "g = (λh. Abs_multiset (λi. h (Suc i) * Suc i))"
  have count_g: "count (g h) = (λi. h (Suc i) * Suc i)" if h: "h partitions

```

```

n" for h
  unfolding g_def
proof (rule count_Abs_multiset)
  have "finite {i. i > 0 ∧ h i > 0}"
    using _ partitions_imp_finite_elements[OF h]
    by (rule finite_subset) auto
  also have "bij_betw (λi. i-1) {i. i > 0 ∧ h i > 0} {i. h (Suc i) *
Suc i > 0}"
    by (rule bij_betwI[of _ _ _ "λi. i + 1"]) auto
  hence "finite {i. i > 0 ∧ h i > 0} ↔ finite {i. h (Suc i) * Suc
i > 0}"
    by (rule bij_betw_finite)
  finally show "finite {i. h (Suc i) * Suc i > 0}" .
qed

define A where "A = {X∈multisets_of_size UNIV n. ∀i. Suc i dvd count
X i}"
have "bij_betw f A {h. h partitions n}"
proof (rule bij_betwI)
  show f: "f ∈ A → {h. h partitions n}"
  proof safe
    fix X assume X: "X ∈ A"
    have X': "size X = n" "∧i. Suc i dvd count X i"
      using X by (auto simp: A_def multisets_of_size_def)
    have X_support: "i < n" if i: "i ∈# X" for i
    proof -
      have "Suc i ≤ count X i"
        using dvd_imp_le[OF X'(2)[of i]] i by (auto simp: f_def)
      also have "... ≤ size X"
        by (rule count_le_size)
      also have "... = n"
        using X'(1) by simp
      finally show "i < n"
        by simp
    qed
  qed
show "f X partitions n"
proof (rule partitionsI)
  fix i assume i: "f X i ≠ 0"
  hence "count X (i - 1) ≠ 0"
    by (intro notI) (auto simp: f_def split: if_splits)
  hence "(i - 1) ∈# X"
    by simp
  moreover have "i > 0"
    using i by (auto simp: f_def split: if_splits)
  ultimately show "1 ≤ i ∧ i ≤ n"
    using X_support[of "i-1"] by (auto simp: f_def)
next
  have "(∑ i≤n. f X i * i) = (∑ i=1..n. f X i * i)"

```

```

    by (rule sum.mono_neutral_right) auto
  also have "... = ( $\sum_{i < n}. f X (Suc i) * Suc i$ )"
    by (intro sum.reindex_bij_witness[of _ " $\lambda i. i+1$ " " $\lambda i. i-1$ "])
auto
  also have "... = ( $\sum_{i < n}. count X i$ )"
  proof (rule sum.cong)
    fix i assume "i  $\in$  {.. $n$ }"
    have "f X (Suc i) * Suc i = count X i div Suc i * Suc i"
      by (simp add: f_def)
    also have "... = count X i"
      using X'(2)[of i] by (rule dvd_div_mult_self)
    finally show "f X (Suc i) * Suc i = count X i" .
  qed auto
  also have "... = ( $\sum_{i \in set\_mset X}. count X i$ )"
    by (intro sum.mono_neutral_right) (use X_support in <auto simp:
not_in_iff>)
  also have "... = size X"
    by (simp add: size_multiset_overloaded_eq)
  also have "... = n"
    using X'(1) by simp
  finally show "( $\sum_{i \leq n}. f X i * i$ ) = n" .
  qed
qed

show g: "g  $\in$  {h. h partitions n}  $\rightarrow$  A"
  unfolding A_def
proof safe
  fix h assume h: "h partitions n"
  note count_g = count_g[OF h]

  show "Suc i dvd count (g h) i" for i
    unfolding count_g by (rule dvd_triv_right)
  have support: "{i. count (g h) i > 0}  $\subseteq$  {.. $n$ }"
  proof safe
    fix i assume "count (g h) i > 0"
    hence "h (Suc i) > 0"
      by (auto simp: count_g)
    hence "Suc i  $\leq$  n"
      using partitionsE(1)[OF h] by blast
    thus "i < n"
      by auto
  qed

  have "size (g h) = ( $\sum i \mid count (g h) i > 0. count (g h) i$ )"
    unfolding size_multiset_overloaded_eq by (rule sum.cong) auto
  also have "... = ( $\sum_{i < n}. count (g h) i$ )"
    by (rule sum.mono_neutral_left) (use support in <auto simp: not_in_iff>)
  also have "... = ( $\sum_{i=1..n}. count (g h) (i-1)$ )"
    by (rule sum.reindex_bij_witness[of _ " $\lambda i. i-1$ " " $\lambda i. i+1$ "]) auto

```

```

also have "... = ( $\sum_{i=1..n} h\ i * i$ )"
  by (intro sum.cong) (auto simp: count_g algebra_simps)
also have "... = ( $\sum_{i \leq n} h\ i * i$ )"
  by (intro sum.mono_neutral_left) auto
also have "... = n"
  using h by (simp add: partitions_def)
finally show "g h  $\in$  multisets_of_size UNIV n"
  unfolding multisets_of_size_def by simp
qed

show "g (f X) = X" if X: "X  $\in$  A" for X
proof (rule multiset_eqI)
  fix i :: nat
  have dvd: "Suc i dvd count X i"
    using that by (auto simp: A_def)
  have "f X partitions n"
    using f X by blast
  hence "count (g (f X)) i = f X (Suc i) * Suc i"
    by (simp add: count_g)
  also have "... = count X i div Suc i * Suc i"
    by (simp add: f_def)
  also have "... = count X i"
    using dvd by (rule dvd_div_mult_self)
  finally show "count (g (f X)) i = count X i" .
qed

show "f (g h) = h" if h: "h  $\in$  {h. h partitions n}" for h
proof
  fix i :: nat
  have count_g: "count (g h) = ( $\lambda i. h\ (Suc\ i) * Suc\ i$ )"
    using count_g[of h] h by simp
  show "f (g h) i = h i"
  proof (cases "i = 0")
    case True
      have "h 0 = 0"
        using that by (auto simp: partitions_def)
      thus ?thesis
        using True by (auto simp: f_def)
    next
      case False
      thus ?thesis
        by (simp add: f_def count_g algebra_simps)
  qed
qed
qed
thus ?thesis
  unfolding f_def A_def .
qed

```

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*:

"($\lambda k. \prod_{i \leq k}. 1 / (1 - \text{fps}_X \wedge \text{Suc } i)$) \longrightarrow *Abs_fps* ($\lambda n. \text{of_nat } (\text{Partition}' n)$)

:: 'a :: {field, t2_space}" (is "?F \longrightarrow ?G")

proof -

define *F* :: "nat \Rightarrow 'a *fps*" **where** "*F* = ($\lambda i. 1 / (1 - \text{fps}_X \wedge \text{Suc } i)$)"

have *F_altdef*: "*F* *i* = *fps_compose* (*Abs_fps* ($\lambda_. 1$)) ($\text{fps}_X \wedge \text{Suc } i$)"

for *i*

by (*simp add: fps_compose_sub_distrib gp F_def del: power_Suc*)

have *fps_nth_F*: "*fps_nth* (*F* *i*) *j* = (if *Suc* *i* *dvd* *j* then 1 else 0)" **for**

i j

by (*auto simp: F_altdef fps_nth_compose_X_power simp del: power_Suc*)

have *subdegree_F*: "*subdegree* ($1 - F$ *i*) = *Suc* *i*" **for** *i*

proof -

have "*F* *i* - 1 = *fps_compose* (*Abs_fps* ($\lambda_. 1$) - 1) ($\text{fps}_X \wedge \text{Suc } i$)"

by (*simp add: F_altdef fps_compose_sub_distrib*)

also have "*subdegree* ... = *subdegree* (*Abs_fps* ($\lambda_. 1$:: 'a) - 1) *"

Suc *i*"

by (*subst subdegree_fps_compose*) *auto*

also have "*subdegree* (*Abs_fps* ($\lambda_. 1$:: 'a) - 1) = 1"

by (*rule subdegreeI*) *auto*

finally show ?thesis

by *simp*

qed

have [*simp*]: "*F* *i* \neq 0" **for** *i*

using *subdegree_F*[*of* *i*] **by** *auto*

have *ev*: "*eventually* ($\lambda i. N \leq \text{subdegree } (F \ i - 1)$) *at_top*" **for** *N*

using *eventually_ge_at_top*[*of* *N*]

by *eventually_elim* (*auto simp del: power_Suc simp: subdegree_F*)

have " $(\lambda n. \text{prod } F \ \{..n\}) \longrightarrow$

Abs_fps ($\lambda n. \sum_{X \in \text{multisets_of_size } \{..n\}} n. \prod_{i \leq n}. \text{fps_nth}$

(*F* *i*) (count *X* *i*)")

proof (*rule tendsto_prod_fps*)

have "*Abs_fps* ($\lambda_. 1$:: 'a) \neq 0"

by (*auto simp: fps_eq_iff*)

thus "*F* *k* \neq 0" **for** *k*

unfolding *F_altdef* **by** (*auto simp: fps_compose_eq_0_iff*)

next

fix *n k* :: nat

assume "*n* < *k*"

thus "*n* < *subdegree* (*F* *k* - 1)"

```

    by (auto simp: subdegree_F)
  qed

  also have "( $\lambda n. \sum_{X \in \text{multisets\_of\_size } \{..n\} n. \prod_{i \leq n. \text{fps\_nth } (F i)} (\text{count } X i)$ ) = ( $\lambda n. \text{of\_nat } (\text{Partition}' n)$ )"
  proof
    fix n :: nat
    have "( $\sum_{X \in \text{multisets\_of\_size } \{..n\} n. \prod_{i \leq n. \text{fps\_nth } (F i)} (\text{count } X i)$ ) =
      ( $\sum_{X \in \text{multisets\_of\_size } \{..n\} n. \text{if } \exists i \leq n. \neg \text{Suc } i \text{ dvd count } X i \text{ then } 0 \text{ else } 1::'a$ )"
      by (rule sum.cong) (auto simp: fps_nth_F)
    also have "( $\sum_{X \in \text{multisets\_of\_size } \{..n\} n. \text{if } \exists i \leq n. \neg \text{Suc } i \text{ dvd count } X i \text{ then } 0 \text{ else } 1::'a$ ) =
      ( $\sum_{X \mid X \in \text{multisets\_of\_size } \{..n\} n \wedge (\forall i \leq n. \text{Suc } i \text{ dvd count } X i). 1$ )"
      by (rule sum.mono_neutral_cong_right) auto
    also have "... = of_nat (card {X. X  $\in$  multisets_of_size {..n} n  $\wedge$  ( $\forall i \leq n. \text{Suc } i \text{ dvd count } X i$ )})"
      by simp
    also have "{X. X  $\in$  multisets_of_size {..n} n  $\wedge$  ( $\forall i \leq n. \text{Suc } i \text{ dvd count } X i$ )} =
      {X. X  $\in$  multisets_of_size UNIV n  $\wedge$  ( $\forall i. \text{Suc } i \text{ dvd count } X i$ )}" (is "?lhs = ?rhs")
    proof (intro equalityI subsetI)
      fix X assume X: "X  $\in$  ?lhs"
      hence X: "X  $\in$  multisets_of_size {..n} n" " $\wedge i. i \leq n \implies \text{Suc } i \text{ dvd count } X i$ "
      by blast+
      have dvd: "Suc i dvd count X i" for i
      proof (cases "i  $\leq$  n")
        case False
        hence "i  $\notin$  X"
        using X(1) by (auto simp: multisets_of_size_def)
        thus ?thesis
        by (simp add: not_in_iff)
      qed (use X(2) in auto)
      thus "X  $\in$  ?rhs"
      using X multisets_of_size_mono[of "{..n}" UNIV] by auto
    next
    fix X assume X: "X  $\in$  ?rhs"
    hence X: "size X = n" " $\wedge i. \text{Suc } i \text{ dvd count } X i$ "
    unfolding multisets_of_size_def by auto
    have "set_mset X  $\subseteq$  {..n}"
    proof safe
      fix i assume "i  $\in$  X"
      hence "Suc i  $\leq$  count X i"
      by (intro dvd_imp_le[OF X(2)]) auto
      also have "...  $\leq$  size X"

```

```

        by (simp add: count_le_size)
    also have "... = n"
        using X(1) by simp
    finally show "i ≤ n"
        by simp
qed
thus "X ∈ ?lhs"
    using X by (simp_all add: multisets_of_size_def)
qed
also have "card ... = card {h. h partitions n}"
    by (rule bij_betw_same_card, rule bij_betw_multisets_of_size_partitions)
also have "... = Partition' n"
    by (simp add: Partition'_eq_card)
finally show "( $\sum_{X \in \text{multisets\_of\_size } \{..n\} n. \prod_{i \leq n. \text{fps\_nth } (F i)}$ 
(count X i)) =
    of_nat (Partition' n)" .

qed
finally show ?thesis
    unfolding F_def .
qed

theorem has_prod_Partition':
  "( $\lambda i. 1 / (1 - \text{fps\_X } ^ \text{Suc } i)$ ) has_prod Abs_fps ( $\lambda n. \text{of\_nat } (\text{Partition}' n)$ )
  :: 'a :: {field, t2_space}" (is "?F has_prod ?G")
proof -
  have "fps_nth (Abs_fps ( $\lambda n. \text{of\_nat } (\text{Partition}' n)$ )) 0  $\neq$  fps_nth 0 0"
    by (auto simp: Partition'_def)
  hence "Abs_fps ( $\lambda n. \text{of\_nat } (\text{Partition}' n)$ ) :: 'a)  $\neq$  0"
    by metis
  hence "raw_has_prod ?F 0 ?G"
    unfolding raw_has_prod_def using has_prod_Partition'_aux[where ?'a
= 'a] by simp
  thus ?thesis
    unfolding has_prod_def by blast
qed

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
  have "ramanujan_theta (-q) (-(q^2)) = (∏ i<3. (q * q ^ i; q^3)_∞)"
    by (subst ramanujan_theta_triple_product_complex)
      (use q in <simp_all flip: power_Suc add: norm_power power_less_one_iff
eval_nat_numeral>)
  also have "... = (q ; q)_∞"
    by (rule prod_qepochhammer_inf_group) (use q in auto)
  finally show ?thesis by (simp add: euler_phi_def)
qed

```

```

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
  have "complex_of_real ((q; q)_∞) = (of_real q; of_real q)_∞"
    by (subst qepochhammer_inf_of_real) (use q in auto)
  also have "... = euler_phi (of_real q)"
    by (simp add: euler_phi_def)
  also have "... = complex_of_real (ramanujan_theta (-q) (-(q^2)))"
    by (subst pentagonal_number_theorem_complex)
      (use q in <auto simp flip: ramanujan_theta_of_real>)
  finally show ?thesis
    by (simp only: of_real_eq_iff euler_phi_def)
qed

```

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 -

```

```

have *: "q ^ pent_num n = q powi (n * (3 * n - 1) div 2)" for n
proof -
  have "n * (3 * n - 1) ≥ 0"
    by (cases "n > 0") (auto intro: mult_nonpos_nonpos)
  hence "q powi (n * (3 * n - 1) div 2) = q ^ nat (n * (3 * n - 1) div
2)"
    unfolding power_int_def by auto
  thus ?thesis
    unfolding pent_num_def by simp
qed

show ?thesis
proof (cases "q = 0")
  case [simp]: True
  have "((λn. (-1) powi n * q powi (n*(3*n-1) div 2))
    has_sum ramanujan_theta (-q) (-q2)) {0}"
    by (intro has_sum_finiteI) auto
  also have "?this ↔ ((λn. (-1) powi n * q ^ pent_num n)
    has_sum ramanujan_theta (-q) (-q2)) UNIV"
    unfolding * by (intro has_sum_cong_neutral) (auto simp: dvd_div_eq_0_iff)
  finally show ?thesis
    by (auto simp: has_sum_iff summable_on_iff_abs_summable_on_complex)
next
  include qepochhammer_inf_notation
  case [simp]: False
  have "(λn. 1 + norm q ^ Suc n) has_prod (-norm q; norm q)∞"
    using has_prod_qepochhammer_inf[of "norm q" "-norm q"] q by simp
  hence th1: "abs_convergent_prod (λn. 1 - q ^ (n+1))"
    by (simp add: abs_convergent_prod_def norm_mult norm_power has_prod_iff)

  have prod: "(λn. 1 - q ^ Suc n) has_prod (q; q)∞"
    using has_prod_qepochhammer_inf[of q q] q by simp
  have "((λn. (-q) powi (n*(n+1) div 2) * (-q2) powi (n*(n-1) div
2))
    has_sum ramanujan_theta (-q) (-q2)) UNIV"
    by (rule has_sum_ramanujan_theta)
    (auto simp: norm_power power_less_one_iff q simp flip: power_Suc)
  also have "(λn. (-q) powi (n*(n+1) div 2) * (-q2) powi (n*(n-1)
div 2)) =
    (λn. (- 1) powi n * q powi (n*(3*n-1) div 2))"
    (is "?lhs = ?rhs")
  proof
    fix n :: int
    have "(-q) powi (n*(n+1) div 2) * (-q2) powi (n*(n-1) div 2)
=
    (-1) powi (n*(n+1) div 2 + n*(n-1) div 2) *
    (q powi (n*(n+1) div 2) * (q2) powi (n*(n-1) div 2))"
    by (auto simp: power_int_minus_left)
    also have "n*(n+1) div 2 + n*(n-1) div 2 = (n*(n+1) + n*(n-1)) div

```

```

2"
  by (rule div_plus_div_distrib_dvd_left [symmetric]) auto
also have "(n*(n+1) + n*(n-1)) div 2 = n ^ 2"
  by (simp add: algebra_simps power2_eq_square)
also have "(-1) powi (n ^ 2) = (-1::complex) powi n"
  by (auto simp: power_int_minus_left)
also have "(q^2) powi (n*(n-1) div 2) = q powi (n*(n-1))"
  by (simp add: power_int_power)
also have "q powi (n * (n + 1) div 2) * q powi (n * (n - 1)) =
          q powi (n * (n + 1) div 2 + (2 * n * (n - 1)) div 2)"
  by (simp add: power_int_add)
also have "n * (n + 1) div 2 + (2 * n * (n - 1)) div 2 = (n*(n+1)
+ 2*n*(n-1)) div 2"
  by (rule div_plus_div_distrib_dvd_left [symmetric]) auto
also have "n*(n+1) + 2*n*(n-1) = n * (3 * n - 1)"
  by (simp add: algebra_simps)
finally show "?lhs n = ?rhs n" .
qed
finally have sum: "((λn. (-1) powi n * q powi (n*(3*n-1) div 2))
  has_sum ramanujan_theta (-q) (-q^2)) UNIV" .
also have "ramanujan_theta (-q) (-q^2) = euler_phi q"
  using pentagonal_number_theorem_complex[of q] assms by (simp add:
euler_phi_def)
finally show ?thesis unfolding * .
qed
qed

```

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 "(∏ n::nat. 1 - q^(n+1)) = (∑ ∞ (k::int). (-1) powi k * q ^ pent_num
k)" (is ?th3)
proof -
  include qpothhammer_inf_notation
  have *: "q ^ pent_num n = q powi (n * (3 * n - 1) div 2)" for n
proof -
  have "n * (3 * n - 1) ≥ 0"
    by (cases "n > 0") (auto intro: mult_nonpos_nonpos)

```

```

    hence "q powi (n * (3 * n - 1) div 2) = q ^ nat (n * (3 * n - 1) div
2)"
      unfolding power_int_def by auto
      thus ?thesis
      unfolding pent_num_def by simp
qed

have "?th1 ∧ ?th2 ∧ ?th3"
proof (cases "q = 0")
  case [simp]: True
  have "((λn. (-1) powi n * q powi (n*(3*n-1) div 2))
    has_sum ramanujan_theta (-q) (-q2)) {0}"
    by (intro has_sum_finiteI) auto
  also have "?this ↔ ((λn. (-1) powi n * q ^ pent_num n)
    has_sum ramanujan_theta (-q) (-q2)) UNIV"
    unfolding * by (intro has_sum_cong_neutral) (auto simp: dvd_div_eq_0_iff)
  finally show ?thesis
    by (auto simp: abs_convergent_prod_def has_sum_iff summable_on_iff_abs_summable_on_co
next
  case [simp]: False
  have "(λn. 1 + norm q ^ Suc n) has_prod (-norm q; norm q)∞"
    using has_prod_qepochhammer_inf[of "norm q" "-norm q"] q by simp
  hence th1: "abs_convergent_prod (λn. 1 - q ^ (n+1))"
    by (simp add: abs_convergent_prod_def norm_mult norm_power has_prod_iff)

  have prod: "(λn. 1 - q ^ Suc n) has_prod (q; q)∞"
    using has_prod_qepochhammer_inf[of q q] q by simp
  have sum: "((λn. (-1) powi n * q ^ pent_num n) has_sum euler_phi q)
UNIV"
    using has_sum_euler_phi_complex[of q] assms by simp

  have "(λn. (-1) powi n * q ^ pent_num n) summable_on UNIV"
    using sum by (rule has_sum_imp_summable)
  hence th2: "(λn. (-1) powi n * q ^ pent_num n) abs_summable_on UNIV"
    by (simp add: summable_on_iff_abs_summable_on_complex)

  have th3: "(∏ n. 1 - q ^ (n+1)) = (∑∞ (k::int). (-1) powi k * q ^
pent_num k)"
    using sum prod by (simp add: has_prod_iff has_sum_iff euler_phi_def)

  show ?thesis
    using th1 th2 th3 by blast
qed
thus ?th1 ?th2 ?th3
  by blast+
qed

```

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 (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 (inv_pent_num n) then
  1 else -1 else 0)"
  shows " $(\lambda k. f k * q ^ k)$  sums euler_phi q"

```

```

proof -
  have " $(\lambda k. (-1) \text{powi } k * q ^ \text{pent\_num } k)$  has_sum euler_phi q" UNIV"
    using q by (rule has_sum_euler_phi_complex)

  also have "?this  $\longleftrightarrow$  ( $(\lambda k. f k * q ^ k)$  has_sum euler_phi q) pent_nums"
    by (rule has_sum_reindex_bij_witness[of _ inv_pent_num pent_num])
      (auto simp: pent_num_inv_pent_num f_def)
  also have "...  $\longleftrightarrow$  ( $(\lambda k. f k * q ^ k)$  has_sum euler_phi q) UNIV"
    by (rule has_sum_cong_neutral) (auto simp: f_def)
  finally show " $(\lambda k. f k * q ^ k)$  sums euler_phi q"
    by (simp add: has_sum_imp_sums)

```

qed

```

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

```

```

proof (rule has_fps_expansionI)
  have "eventually ( $\lambda q. q \in \text{ball } 0 \ 1$ ) (nhds (0::complex))"
    by (rule eventually_nhds_in_open) auto
  thus "eventually ( $\lambda q. (\lambda n. \text{fps\_nth } \text{fps\_euler\_phi } n * q ^ n)$  sums euler_phi
  q) (nhds (0::complex))"
  proof eventually_elim
    case (elim q)
    thus ?case
      using sums_euler_phi_complex[of q] by (simp add: fps_euler_phi_def)
  qed

```

qed

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 (λn. of_nat (Partition' n) :: complex) = inverse fps_euler_phi"
proof -
  define f :: "nat ⇒ complex fps" where "f = (λn. (∏ i<n. (1 - fps_X
  ^ Suc i)))"
  define h :: "complex fps" where "h = inverse (Abs_fps (λn. of_nat (Partition'
  n)))"
  define f' :: "nat ⇒ complex ⇒ complex" where "f' = (λn z. (∏ i<n.
  (1 - z ^ Suc i)))"

  have "fps_euler_phi = h"
  proof (rule uniform_limit_imp_fps_expansion_eq)
    have "(λk. ∏ i<k. 1 / (1 - fps_X ^ Suc i)) ⟶ Abs_fps (λn. complex_of_nat
  (Partition' n))"
      using has_prod_Partition'_aux[where ?'a = complex] LIMSEQ_lessThan_iff_atMost
  by blast
    hence "(λk. inverse (∏ i<k. 1 / (1 - fps_X ^ Suc i))) ⟶ h"
      unfolding h_def by (intro tendsto_intros) auto
    also have "(λk. inverse (∏ i<k. 1 / (1 - fps_X ^ Suc i))) = f"
      by (simp add: f_def inverse_prod_fps fps_divide_unit)
    finally show "(f ⟶ h) sequentially" .
  next
    show "open (ball 0 (1/2) :: complex set)" "0 ∈ (ball 0 (1/2) :: complex
  set)"
      by auto
  next
    show "∀F x in sequentially. f' x has_fps_expansion f x"
      unfolding f'_def f_def by (intro always_eventually allI fps_expansion_intros)
  next
    show "euler_phi has_fps_expansion (fps_euler_phi :: complex fps)"
      by (rule has_fps_expansion_euler_phi_complex)
  next
    define r where "r = (λn (a,q). ∏ k<n. 1 - a * q ^ k :: complex)"
    define s where "s = (λ(a,q). qepochhammer_inf a q :: complex)"
    have "uniform_limit (cball 0 (1/2) × cball 0 (1/2)) r s sequentially"
      unfolding r_def s_def by (rule uniform_limit_qepochhammer_inf) (auto
  simp: compact_Times)
    hence 1: "uniform_limit (ball 0 (1/2)) (λn q. r n (q, q)) (λq. s
  (q, q)) sequentially"
      by (rule uniform_limit_compose) auto
    thus "uniform_limit (ball 0 (1 / 2)) f' euler_phi sequentially"
      by (simp add: f'_def r_def s_def euler_phi_def [abs_def])
  next
    show "∀F x in sequentially. f' x holomorphic_on ball 0 (1 / 2)"
      unfolding f'_def by (intro always_eventually allI holomorphic_intros)
  qed auto

  hence "inverse fps_euler_phi =
    inverse (inverse (Abs_fps (λn. of_nat (Partition' n) :: complex)))"

```

```

    unfolding h_def by simp
    also have "... = Abs_fps ( $\lambda n. \text{of\_nat (Partition' n)}$ )"
      by (rule fps_inverse_idempotent) auto
    finally show ?thesis ..
qed

lemma fps_nth_euler_phi_0 [simp]: "fps_nth fps_euler_phi 0 = 1"
proof -
  have "pent_num 0  $\in$  pent_nums"
    by blast
  moreover have "inv_pent_num 0 = 0"
    by (intro inv_pent_num_eqI) auto
  ultimately show ?thesis
    unfolding fps_euler_phi_def by simp
qed

lemma has_fps_expansion_inverse_euler_phi_complex:
  " $(\lambda q. 1 / \text{euler\_phi } q :: \text{complex})$ 
  has_fps_expansion Abs_fps ( $\lambda n. \text{of\_nat (Partition' n)}$ )"
proof -
  have " $(\lambda q. \text{inverse (euler\_phi } q :: \text{complex})$ ) has_fps_expansion (inverse
  fps_euler_phi)"
    by (intro fps_expansion_intros has_fps_expansion_euler_phi_complex)
  auto
  also have "... = Abs_fps ( $\lambda n. \text{of\_nat (Partition' n)}$ )"
    unfolding fps_Partition'_eq ..
  finally show ?thesis
    by (simp add: field_simps)
qed

lemma conv_radius_Partition':
  " $\text{conv\_radius } (\lambda n. \text{of\_nat (Partition' n)} :: 'a :: \{\text{banach, real\_normed\_algebra\_1}\})$ 
 $\geq 1$ "
proof -
  have "fps_conv_radius (Abs_fps ( $\lambda n. \text{of\_nat (Partition' n)} :: \text{complex}$ ))
 $\geq 1$ "
    using has_fps_expansion_inverse_euler_phi_complex
    by (rule holomorphic_on_imp_fps_conv_radius_ge) (auto intro!: holomorphic_intros)
  thus ?thesis
    by (simp add: fps_conv_radius_def conv_radius_def)
qed

lemma sums_inverse_euler_phi_complex:
  assumes "norm z < 1"
  shows " $(\lambda k. \text{of\_nat (Partition' k)} * z ^ k :: \text{complex})$  sums (1 / euler_phi
  z)"
proof -
  define F where "F = Abs_fps ( $\lambda k. \text{of\_nat (Partition' k)} :: \text{complex}$ )"
  have " $(\lambda n. \text{fps\_nth } F n * z ^ n)$  sums (1 / euler_phi z)"

```

```

proof (rule has_fps_expansion_imp_sums_complex)
  show "(λz. 1 / euler_phi z) has_fps_expansion F"
    unfolding F_def using has_fps_expansion_inverse_euler_phi_complex
by simp
next
  show "(λz. 1 / euler_phi z) holomorphic_on eball 0 1"
    by (auto intro!: holomorphic_intros)
qed (use assms in auto)
thus ?thesis
  by (simp add: F_def)
qed

lemma sums_inverse_euler_phi_real:
  assumes "|x| < (1::real)"
  shows "(λk. of_nat (Partition' k) * x ^ k) sums (1 / euler_phi x)"
proof -
  have "(λk. of_nat (Partition' k) * (of_real x) ^ k :: complex) sums
(1 / euler_phi (of_real x))"
    by (rule sums_inverse_euler_phi_complex) (use assms in auto)
  also have "(λk. of_nat (Partition' k) * (of_real x) ^ k :: complex)
=
      (λk. of_real (of_nat (Partition' k) * x ^ k) :: complex)"
    by simp
  also have "1 / euler_phi (of_real x :: complex) = of_real (1 / euler_phi
x)"
    using assms by (simp add: euler_phi_of_real)
  finally show ?thesis
    unfolding sums_of_real_iff .
qed

```

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) =
      (∑ i | i ∈ {1..n} ∧ i ∈ pent_nums.
      (if even (inv_pent_num i) then -1 else 1) * int (Partition'
(n - i)))"
proof -
  define F where "F = Abs_fps (λn. of_nat (Partition' n) :: complex)"
  define G where "G = (fps_euler_phi :: complex fps)"

```

```

have "complex_of_int 0 = fps_nth (1 :: complex fps) n"
  using n by simp
also have "1 = G * F"
  unfolding F_def G_def fps_Partition'_eq by (simp add: inverse_mult_eq_1')
also have "fps_nth (G * F) n = (∑ i=0..n. fps_nth G i * fps_nth F (n
- i))"
  by (subst fps_mult_nth) auto
also have "... = (∑ i=0..n. of_int (fps_nth fps_euler_phi i * int (Partition'
(n - i))))"
  by (intro sum.cong) (auto simp: G_def F_def fps_euler_phi_def)
also have "... = of_int (∑ i=0..n. fps_nth fps_euler_phi i * int (Partition'
(n - i)))"
  by simp
finally have "0 = (∑ i = 0..n. fps_nth fps_euler_phi i * int (Partition'
(n - i)))"
  by (simp only: of_int_eq_iff)
also have "... = (∑ i∈{0..n}-{0}. fps_nth fps_euler_phi i * int (Partition'
(n - i))) + Partition' n"
  by (subst sum.remove[of _ 0]) auto
also have "(∑ i∈{0..n}-{0}. fps_nth fps_euler_phi i * int (Partition'
(n - i))) =
  (∑ i | i ∈ {1..n} ∧ i ∈ pent_nums. fps_nth fps_euler_phi
i * int (Partition' (n - i)))"
  by (intro sum.mono_neutral_right) (auto simp: fps_euler_phi_def)
finally have "int (Partition' n) =
  (∑ i | i ∈ {1..n} ∧ i ∈ pent_nums. -fps_nth fps_euler_phi
i * int (Partition' (n - i)))"
  by (auto simp: add_eq_0_iff2 sum_negf)
also have "... = (∑ i | i ∈ {1..n} ∧ i ∈ pent_nums.
  (if even (inv_pent_num i) then -1 else 1) * int (Partition'
(n - i)))"
  by (intro sum.cong) (auto simp: fps_euler_phi_def)
finally show ?thesis .
qed

```

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 ϕ we have:

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

After combining the two, we take logarithms and rearrange to get 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 " $\text{real (Partition' } n) \leq \text{pi} * \exp(\text{pi} * \text{sqrt}(2 / 3 * \text{real } n)) / \text{sqrt}(6 * (\text{real } n - 1))$ "

proof -

define c where " $c = \text{real } n - 1$ "

have c : " $c > 0$ "

using n by (*simp add: c_def*)

define u where " $u = \text{sqrt}(1 + 2/3 * c * \text{pi} ^ 2)$ "

have u : " $u > 1$ "

using c by (*simp add: u_def*)

define t :: *real* where " $t = (u - 1) / (2 * c)$ "

have t : " $t > 0$ "

using c *unfolding t_def* by (*intro divide_pos_pos*) (*auto simp: u_def*)

define x where " $x = 1 / (1 + t)$ "

have x : " $x \in \{0 < .. < 1\}$ "

using t by (*auto simp: x_def*)

have t_{conv_x} : " $t = (1 - x) / x$ "

using t by (*simp add: x_def field_simps*)

define g where " $g = (\lambda x. \text{pi} ^ 2 / (6 * x) + c * x + \ln x)$ "

define f where " $f = (\lambda x :: \text{real}. 1 / \text{euler_phi } x)$ "

define p where " $p = (\lambda n. \text{real (Partition' } n))$ "

have p_{pos} : " $p n > 0$ " for n

by (*simp add: p_def Partition'_pos*)

have " $p n * x ^ n / (1 - x) \leq f x$ "

```

proof (rule sums_le)
  show "( $\lambda k. p k * x ^ k$ ) sums f x"
    unfolding p_def f_def by (rule sums_inverse_euler_phi_real) (use
x in auto)
  next
    have "( $\lambda k. p n * x ^ n * x ^ k$ ) sums (p n * x ^ n * (1 / (1 - x)))"
      by (intro sums_mult geometric_sums) (use x in auto)
    also have "( $\lambda k. p n * x ^ n * x ^ k$ ) = ( $\lambda k. p n * x ^ (k + n)$ )"
      by (simp add: power_add mult_ac)
    also have "p n * x ^ n * (1 / (1 - x)) = p n * x ^ n / (1 - x)"
      by simp
    finally have "( $\lambda k. \text{if } k + n \geq n \text{ then } p n * x ^ (k + n) \text{ else } 0$ ) sums
(p n * x ^ n / (1 - x))"
      by simp
    thus "( $\lambda k. \text{if } k \geq n \text{ then } p n * x ^ k \text{ else } 0$ ) sums (p n * x ^ n /
(1 - x))"
      by (subst (asm) sums_zero_iff_shift) auto
  next
    fix k :: nat
    show "(if n ≤ k then p n * x ^ k else 0) ≤ p k * x ^ k"
      unfolding p_def using x by (auto intro!: Partition'_mono)
qed
hence "ln (p n * x ^ n / (1 - x)) ≤ ln (f x)"
  unfolding f_def using x
  by (subst ln_le_cancel_iff)
  (auto intro!: euler_phi_pos_real divide_pos_pos mult_pos_pos p_pos)
also have "ln (p n * x ^ n / (1 - x)) = ln (p n) + real n * ln x - ln
(1 - x)"
  using x p_pos[of n] by (simp add: ln_mult ln_div ln_realpow)
finally have "ln (p n) ≤ ln (f x) + real n * (-ln x) + ln (1 - x)"
  by simp

also have "ln (f x) ≤ pi ^ 2 / (6 * t)"
  using ln_euler_phi_ge[of x] x by (simp add: f_def ln_div t_conv_x)
also have "1 - x = t * x"
  using x by (simp add: t_conv_x)
also have "ln (t * x) = ln t + ln x"
  using t x by (simp add: ln_mult)
also have "pi^2 / (6 * t) + real n * -ln x + (ln t + ln x) =
pi^2 / (6 * t) + c * (-ln x) + ln t"
  by (simp add: algebra_simps c_def)
also have "-ln x ≤ t"
proof -
  have "-ln x = ln (1 / x)"
    by (subst ln_div) (use x in auto)
  also have "1 / x = 1 + t"
    using x by (simp add: t_conv_x field_simps)
  also have "ln (1 + t) ≤ t"
    by (rule ln_add_one_self_le_self) (use x in <auto simp: t_conv_x>)

```

```

    finally show ?thesis .
qed
hence "c * (-ln x) ≤ c * t"
  using c by (intro mult_left_mono) auto
finally have "ln (p n) ≤ g t"
  unfolding g_def c_def by linarith
hence "exp (ln (p n)) ≤ exp (g t)"
  by (subst exp_le_cancel_iff)
hence "p n ≤ exp (g t)"
  by (simp add: p_pos)

also have "... = exp (pi2 / (6 * t) + c * t) * t"
  using t by (simp add: g_def exp_add)
also have "c * t = (u - 1) / 2"
  using c by (simp add: t_def)
also have "pi2 / (6 * t) = pi2 / 3 * c / (u - 1)"
  unfolding t_def by (simp add: divide_simps)
also have "pi2 / 3 * c / (u - 1) + (u - 1) / 2 = sqrt (1 + 2 * c * pi2 / 3)"
  using c by (simp add: u_def field_simps)
also have "... ≤ sqrt (2 / 3 * pi2 * real n)"
proof -
  have "1 + 2 * c * pi2 / 3 = 2 / 3 * pi2 * real n + (1 - 2 / 3 * pi2)"
    by (simp add: c_def field_simps)
  also have "1 - 2 / 3 * pi2 ≤ 1 - 2 / 3 * 32"
    by (intro diff_left_mono mult_left_mono power_mono less_imp_le[OF pi_gt3]) auto
  also have "... ≤ 0"
    by simp
  finally have "1 + 2 * c * pi2 / 3 ≤ 2 / 3 * pi2 * real n"
    by simp
  thus ?thesis
    by (rule real_sqrt_le_mono)
qed
also have "... = pi * sqrt (2 / 3 * real n)"
  by (simp add: real_sqrt_mult real_sqrt_divide)
also have "t = (sqrt (1 + 2 * c * pi2 / 3) - 1) / (2 * c)"
  by (simp add: t_def u_def)
also have "... ≤ (sqrt (2 * c * pi2 / 3)) / (2 * c)"
  using c sqrt_add_le_add_sqrt[of 1 "2 * c * pi2 / 3"]
  by (intro divide_right_mono) simp_all
also have "... = pi / (sqrt 2 * sqrt 3 * sqrt c)"
  using c by (simp add: real_sqrt_mult real_sqrt_divide field_simps power2_eq_square)
also have "... = pi / sqrt (6 * c)"
  using c by (simp flip: real_sqrt_mult)

finally show ?thesis

```

```

    using t by (simp add: mult_ac c_def p_def)
qed

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)  $\in$  {0.. $\text{int}(\text{length } ps)$ });
  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_inner1_aux_rec:
  assumes k: "k  $\geq$  1"
  shows "partition_impl1_inner1_aux n ps (1 + k) =
  partition_impl1_inner1_aux n ps k + (if even k then -1 else
1) * ps ! nat (n - int (pent_num k))"
proof -
  have "partition_impl1_inner1_aux n ps (1 + k) =
  ( $\sum$  i $\in$ insert k {1.. $k$ }. (if even i then - 1 else 1) * ps ! nat
(n - int (pent_num i)))"
  unfolding partition_impl1_inner1_aux_def using k by (intro sum.cong
refl) auto
  also have "... = partition_impl1_inner1_aux n ps k + (if even k then
```

```

-1 else 1) * ps ! nat (n - int (pent_num k))"
  by (subst sum.insert) (auto simp: partition_impl1_inner1_aux_def add_ac)
  finally show ?thesis .
qed

```

```

lemma partition_impl1_final:
  assumes "partition_impl1_inner1_invar n ps acc (k, acc'" "int (pent_num
k) > n"
  shows "acc' = acc + partition_impl1_inner1_aux' n ps"
proof -
  have "partition_impl1_inner1_aux n ps k = partition_impl1_inner1_aux'
n ps"
    unfolding partition_impl1_inner1_aux_def partition_impl1_inner1_aux'_def
  proof (intro sum.cong refl)
    have "pent_num (i - 1) < pent_num i" if "i > 0" for i
      by (rule strict_mono_pent_num) (use that in auto)
    show "{1..<k} = {i. 1 ≤ i ∧ int (pent_num i) ≤ n}"
    proof (intro equalityI subsetI)
      fix i assume i: "i ∈ {1..<k}"
      have "pent_num i ≤ pent_num (k-1)"
        using i by (auto simp: pent_num_le_iff_nonneg)
      also have "... ≤ n"
        using assms by (auto simp: partition_impl1_inner1_invar_def)
      finally show "i ∈ {i. 1 ≤ i ∧ int (pent_num i) ≤ n}"
        using i by auto
    next
      fix i assume i: "i ∈ {i. 1 ≤ i ∧ int (pent_num i) ≤ n}"
      have k: "int (pent_num (k - 1)) ≤ n" "k > 0"
        using assms by (auto simp: partition_impl1_inner1_invar_def)
      have "int (pent_num i) ≤ n"
        using i by simp
      also have "n < int (pent_num k)"
        using assms by auto
      finally have "pent_num i < pent_num k"
        by linarith
      hence "i < k"
        using i <k > 0> by (simp add: pent_num_less_iff_nonneg)
      with i show "i ∈ {1..<k}"
        by auto
    qed
  qed
  thus ?thesis
    using assms(1) by (simp add: partition_impl1_inner1_invar_def)
qed

```

```

lemma partition_impl1_inner1_correct [refine_vcg]:
  assumes "n ∈ {1..int (length ps)}"
  shows "partition_impl1_inner1 n ps acc ≤
    SPEC (λacc'. acc' = acc + partition_impl1_inner1_aux' n ps)"

```

```

unfolding partition_impl1_inner1_def
apply refine_vcg
  apply (rule wf_measure[of "\λ(k,_). nat n + 1 - pent_num k"])
subgoal
  using assms
  by (auto simp: partition_impl1_inner1_invar_def partition_impl1_inner1_aux_def)
subgoal for k_acc' k acc
  using assms strict_mono_pent_num[of 0 k]
  by (auto simp: partition_impl1_inner1_invar_def)
subgoal for k_acc' k acc'
  using assms
  by (auto simp: partition_impl1_inner1_invar_def partition_impl1_inner1_aux_rec
algebra_simps
      nat_diff_distrib)
subgoal for k_acc' k acc'
  using assms
  by (auto simp: partition_impl1_inner1_invar_def intro!: diff_less_mono2
simp: strict_mono_pent_num)
subgoal for k_acc' k acc'
  using partition_impl1_final[of n ps acc k acc'] by simp
done

```

definition partition_impl2_inner1 :: "int \Rightarrow int list \Rightarrow int \Rightarrow int nres"
where

```

"partition_impl2_inner1 n ps acc = do {
  (_, _, acc')  $\leftarrow$ 
  WHILET ( $\lambda(k,i,_). i \geq 0$ ) ( $\lambda(k,i,acc). do \{$ 
    ASSERT (nat i < length ps);
    let x = ps ! nat i;
    RETURN (k + 1, i - (3 * k + 1), if even k then acc - x else acc
+ x)
  })
  (1, n - 1, acc);
  RETURN acc'
}"

```

lemma partition_impl2_inner1_refine [refine]:

```

"partition_impl2_inner1 n ps acc  $\leq$   $\Downarrow$ Id (partition_impl1_inner1 n ps
acc)"

```

proof -

```

define R where "R = br ( $\lambda(k,i,acc). (k,acc)$ ) ( $\lambda(k,i,acc::int). i =
n - int (pent_num k)$ )"

```

```

note [refine_dref_RELATES] = RELATESI[of R]

```

```

define w2 where "w2 = WHILET ( $\lambda(k, i, _). 0 \leq i$ )
( $\lambda(k, i, acc).$ 
  ASSERT (nat i < length ps)  $\gg$ 
  ( $\lambda_. let x = ps ! nat i$ 

```

```

      in RETURN (k + 1, i - (3 * k + 1), if even k then acc -
x else acc + x))) (1, n - 1, acc)"
  define w1 where "w1 = WHILE_T partition_impl1_inner1_invar n ps acc
    (λ(k, _). int (pent_num k) ≤ n)
    (λ(k, acc).
      ASSERT (n - int (pent_num k) ∈ {0..<int (length ps)}) >>=
        (λ_. let x = ps ! nat (n - int (pent_num k))
          in RETURN (k + 1, if even k then acc - x else acc
+ x))) (1, acc)"
  have [refine_vcg]: "w2 ≤ ↓R w1"
    unfolding w1_def w2_def
    by refine_rcg (auto simp: R_def br_def pent_num_def[of 1] pent_num_increment)
  have "do {(_,_,acc') ← w2; RETURN acc'} ≤ ↓int_rel (do {(_,acc') ←
w1; RETURN acc'})"
    by refine_vcg (auto simp: R_def br_def)
  thus ?thesis
    unfolding partition_impl2_inner1_def partition_impl1_inner1_def w1_def
w2_def .
qed

```

```

lemma partition_impl2_inner1_correct [refine_vcg]:
  assumes "n ∈ {1..int (length ps)}"
  shows "partition_impl2_inner1 n ps acc ≤ SPEC (λacc'. acc' = acc
+ partition_impl1_inner1_aux' n ps)"
proof (rule order.trans[OF _ partition_impl1_inner1_correct[OF assms]])
  show "partition_impl2_inner1 n ps acc ≤ partition_impl1_inner1 n ps
acc"
  using partition_impl2_inner1_refine[of n ps acc] by simp
qed

```

```

definition partition_impl1_inner2_aux where
  "partition_impl1_inner2_aux n ps k =
    (∑ 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 =
    (∑ i | i ≥ 1 ∧ int (pent_num (-i)) ≤ 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 = (λ(k, acc).
  k > 0 ∧ int (pent_num -(k - 1)) ≤ n ∧
  acc = acc0 + partition_impl1_inner2_aux n ps k)"

```

```

definition partition_impl1_inner2 :: "int ⇒ int list ⇒ int ⇒ int nres"
where
  "partition_impl1_inner2 n ps acc = do {

```

```

      (k, acc) ←
        WHILE_T partition_impl1_inner2_invar n ps acc (λ(k, _). int (pent_num
(-k)) ≤ n) (λ(k, acc). do {
          ASSERT (n - int (pent_num (-k)) ∈ {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 ≥ 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 -
    have "partition_impl1_inner2_aux n ps (1 + k) =
          (∑ i ∈ insert k {1..<k}. (if even i then - 1 else 1) * ps ! nat
(n - int (pent_num (-i))))"
      unfolding partition_impl1_inner2_aux_def using k by (intro sum.cong
refl) auto
    also have "... = partition_impl1_inner2_aux n ps k + (if even k then
-1 else 1) * ps ! nat (n - int (pent_num (-k)))"
      by (subst sum.insert) (auto simp: partition_impl1_inner2_aux_def add_ac)
    finally show ?thesis .
  qed

```

```

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 -
    have "partition_impl1_inner2_aux n ps k = partition_impl1_inner2_aux'
n ps"
      unfolding partition_impl1_inner2_aux_def partition_impl1_inner2_aux'_def
proof (intro sum.cong refl)
    have "pent_num (-(i-1)) < pent_num (-i)" if "i > 0" for i
      by (rule strict_antimono_pent_num) (use that in auto)
    show "{1..<k} = {i. 1 ≤ i ∧ int (pent_num (-i)) ≤ n}"
      proof (intro equalityI subsetI)
        fix i assume i: "i ∈ {1..<k}"
        have "pent_num (-i) ≤ pent_num (-(k-1))"
          using i by (auto simp: pent_num_le_iff_nonpos)
        also have "... ≤ n"
          using assms by (auto simp: partition_impl1_inner2_invar_def)
        finally show "i ∈ {i. 1 ≤ i ∧ int (pent_num (-i)) ≤ n}"
          using i by auto
      next

```

```

fix i assume i: "i ∈ {i. 1 ≤ i ∧ int (pent_num (-i)) ≤ n}"
have k: "int (pent_num (-(k - 1))) ≤ n" "k > 0"
  using assms by (auto simp: partition_impl1_inner2_invar_def)
have "int (pent_num (-i)) ≤ n"
  using i by simp
also have "n < int (pent_num (-k))"
  using assms by auto
finally have "pent_num (-i) < pent_num (-k)"
  by linarith
hence "i < k"
  using i <k > 0> by (simp add: pent_num_less_iff_nonpos)
with i show "i ∈ {1..<k}"
  by auto
qed
qed
thus ?thesis
  using assms(1) by (simp add: partition_impl1_inner2_invar_def)
qed

lemma partition_impl1_inner2_correct [refine_vcg]:
  assumes "n ∈ {1..int (length ps)}"
  shows "partition_impl1_inner2 n ps acc ≤
    SPEC (λacc'. acc' = acc + partition_impl1_inner2_aux' n ps)"
  unfolding partition_impl1_inner2_def
  apply refine_vcg
  apply (rule wf_measure[of "λ(k,_). nat n + 1 - pent_num (-k)"])
  subgoal
    using assms
    by (auto simp: partition_impl1_inner2_invar_def partition_impl1_inner2_aux_def)
  subgoal for k_acc' k acc'
    using assms strict_antimono_pent_num[of "-k" 0]
    by (auto simp: partition_impl1_inner2_invar_def)
  subgoal for k_acc' k acc'
    using assms
    by (auto simp: partition_impl1_inner2_invar_def partition_impl1_inner2_aux_rec
      algebra_simps
      nat_diff_distrib)
  subgoal for k_acc' k acc'
    using assms
    by (auto simp: partition_impl1_inner2_invar_def intro!: diff_less_mono2
      simp: strict_antimono_pent_num)
  subgoal for k_acc' k acc'
    using partition_impl1_inner2_final[of n ps acc k acc'] by simp
  done

```

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 -

```

define R where "R = br (λ(k,i,acc). (k,acc)) (λ(k,i,acc::int). i =
n - int (pent_num (-k)))"

```

```

note [refine_dref_RELATES] = RELATESI[of R]

```

```

define w2 where "w2 = WHILE_T (λ(k, i, _) . 0 ≤ i)
(λ(k, i, acc).

```

```

  ASSERT (nat i < length ps) >>=

```

```

  (λ_. let x = ps ! nat i

```

```

    in RETURN (k + 1, i - (3 * k + 2), if even k then acc -
x else acc + x))) (1, n - 2, acc)"

```

```

define w1 where "w1 = WHILE_T partition_impl1_inner2_invar n ps acc

```

```

  (λ(k, _) . int (pent_num (-k)) ≤ n)

```

```

  (λ(k, acc).

```

```

    ASSERT (n - int (pent_num (-k)) ∈ {0..<int (length ps)})

```

```

>>=

```

```

  (λ_. let x = ps ! nat (n - int (pent_num (-k)))

```

```

    in RETURN (k + 1, if even k then acc - x else acc

```

```

+ x))) (1, acc)"

```

```

have [refine_vcg]: "w2 ≤ ↓R w1"

```

```

unfolding w1_def w2_def

```

```

by refine_rcg (auto simp: R_def br_def pent_num_def[of "-1"] pent_num_decrement)

```

```

have "do {(_,_,acc') ← w2; RETURN acc'} ≤ ↓int_rel (do {(_,acc') ←
w1; RETURN acc'})"

```

```

by refine_vcg (auto simp: R_def br_def)

```

```

thus ?thesis

```

```

unfolding partition_impl2_inner2_def partition_impl1_inner2_def w1_def

```

```

w2_def .

```

```

qed

```

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 (rule order.trans[OF _ partition_impl1_inner2_correct[OF assms]])
  show "partition_impl2_inner2 n ps acc ≤ partition_impl1_inner2 n ps
acc"
  using partition_impl2_inner2_refine[of n ps acc] by simp
qed

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 -
  define S :: "int set ⇒ int ⇒ int" where
    "S = (λA c. (∑ i | i ∈ A ∧ int (pent_num (c*i)) ≤ int n.
      (if even (c*i) then -1 else 1) * ps ! nat (int n - int (pent_num
(c * i))))))"

  have "?lhs = S {1..} 1 + S {1..} (-1)"
    unfolding partition_impl1_inner1_aux'_def partition_impl1_inner2_aux'_def
S_def by simp
  also have "S {1..} (-1) = S {..-1} 1"
    unfolding S_def by (rule sum.reindex_bij_witness[of _ uminus uminus])
  auto
  also have "S {1..} 1 + S {..-1} 1 = S (-{0}) 1" unfolding S_def
    by (subst sum.union_disjoint [symmetric])
    (auto intro!: sum.cong finite_subset[OF _ finite_pent_num_le[of
n]])
  also have "... = (∑ i | i ∈ - {0} ∧ int (pent_num (1 * i)) ≤ int n.
      (if even (1 * i) then - 1 else 1) * Partition' (n
- pent_num i))"
    unfolding S_def by (intro sum.cong) (use assms in <auto simp: nat_diff_distrib
pent_num_pos_iff>)
  also have "... = (∑ i | i ∈ {1..n} ∧ i ∈ pent_nums.
      (if even (inv_pent_num i) then -1 else 1) * int (Partition'
(n - i)))"
    by (rule sum.reindex_bij_witness[of _ inv_pent_num pent_num])
    (auto simp: Suc_le_eq pent_num_pos_iff pent_nums_def)
  also have "... = int (Partition' n)"
    by (subst (2) Partition'_recurrence) (use assms(1) in auto)
  finally show ?thesis .
qed

```

```

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)"
  unfolding partition_impl2_inner_def
  apply refine_vcg
  subgoal
    using assms by simp
  subgoal
    using assms by simp
  subgoal for acc acc'
    using partition_impl2_inner_correct_aux[of n ps] assms by simp
  done

```

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)"
  unfolding partition_impl2_def Partition'_list_def
  apply refine_vcg
  apply simp
  apply (rule wf_measure[of "λ(m,_). n + 1 - m"])
  apply (auto simp: nth_list_update' simp del: upt_Suc intro!:
nth_equalityI)
  done

```

```

lemma param_dvd_nat [param, seprel_import_param]:
  "(dvd), (dvd) ∈ nat_rel → nat_rel → bool_rel"
  "(dvd), (dvd) ∈ int_rel → int_rel → bool_rel"
  by simp_all

```

```

sepref_definition partition_impl3 is
  "partition_impl2" :: "nat_assnd →a array_assn int_assn"
  unfolding partition_impl2_def partition_impl2_inner_def
             partition_impl2_inner1_def partition_impl2_inner2_def
  by sepref

lemma partition_impl3_correct':
  "(partition_impl3, λn. RETURN (Partition'_list n)) ∈ nat_assnd →a array_assn
  int_assn"
proof -
  have *: "(partition_impl2, λn. RETURN (Partition'_list n)) ∈ nat_rel
  → ⟨Id⟩nres_rel"
  by refine_vcg simp?
  show ?thesis
  using partition_impl3.refine[FCOMP *] .
qed

theorem partition_impl3_correct:
  "<nat_assn n n> partition_impl3 n <array_assn int_assn (Partition'_list
n)>t"
proof -
  have [simp]: "nofail (partition_impl2 n)"
  using partition_impl2_correct[of n] le_RES_nofailI by blast
  have 1: "xs = Partition'_list n" if "RETURN xs ≤ partition_impl2 n"
  for xs
  using that partition_impl2_correct[of n] by (simp add: pw_le_iff)
  note r1 = partition_impl3.refine[THEN hfrefD, of n n, THEN hn_refineD,
simplified]
  show ?thesis
  apply (rule cons_rule[OF _ _ r1])
  apply (sep_auto simp: pure_def)
  apply (sep_auto simp: pure_def dest!: 1)
  done
qed

end
theory Partition_Number_Imperative_Test
imports
  Partition_Number_Imperative
  "HOL-Library.Code_Target_Numerals"
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_val <@{code partition_impl3_test} 100 ()>
```

```
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>.