Verification and generation of unrefinable partitions

Riccardo Aragona, Lorenzo Campioni, Roberto Civino

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica - Università dell'Aquila, Italy

Massimo Lauria

Dipartimento di Scienze Statistiche - Sapienza Università di Roma, Italy

Abstract

Unrefinable partitions are a subset of partitions into distinct parts which satisfy an additional unrefinability property. More precisely, being an unrefinable partition means that none of the parts can be written as the sum of smaller integers without introducing a repetition. We address the algorithmic aspects of unrefinable partitions, such as testing whether a given partition is unrefinable or not and enumerating all the partitions whose sum is a given integer. We design two algorithms to solve the two mentioned problems and we discuss their complexity.

Keywords: integer partitions into distinct part, minimal excludant, algorithms.

1. Introduction

Given $N \in \mathbb{N}$, a *partition* of N is a finite sequence of positive integers $\lambda_1, \lambda_2, \ldots, \lambda_k$ such that

$$N = \lambda_1 + \lambda_2 + \cdots + \lambda_k.$$

Each λ_i is called a *part* of the partition, and a partition into *distinct parts* is a partition where all parts are distinct integers. In this work we only discuss partitions into distinct parts and, among them, we are interested in the study of *unrefinable* partitions, where refining a partition means splitting any of its parts as the sum of smaller pieces. When the repetition of elements is allowed, the process of refining a partition can proceed until one obtains

$$N = \underbrace{1 + 1 + \cdots + 1}_{N \text{ times}}$$

and for this reason it is of little interest.

The problem becomes more interesting if we require the considered partitions, after the refinement, to still be partitions into distinct parts. Consider for example the following partition of 50:

$$50 = 1 + 2 + 3 + 4 + 6 + 7 + 11 + 16.$$

Is there a way to refine any of the parts without introducing a repetition? In this case the answer is no, but how can this be tested and what effort is required?

There are well known algorithms and formulas to enumerate or count all partitions of a given N [And76], but none is known, to our knowledge, for unrefinable partitions. In order to close this gap, we address in this paper some algorithmic aspects related to unrefinable partitions. We present two algorithms: one which verifies whether a partition is unrefinable or not, and one which recursively enumerates all unrefinable partition of a given N. More precisely, we discuss a naïve $O(\ell^3)$-algorithm which determines if an increasing sequence of integers with maximum element equal to ℓ represents an unrefinable partition and show that it can be improved by means of simple arithmetical arguments. We prove that the verification problem can be actually solved in $O(\ell + \mu^2)$ steps, where μ is the *minimal excludant* of the sequence, i.e. the least integer that is not a part, which is defined in detail in the next section. Following Aragona et al. [ACCL22] we have that ℓ and μ are upper bounded by some functions in $O(\sqrt{N})$, therefore the verification algorithm is linear in N in the worst case.

1.1. Related works
The notion of unrefinable partition is at least as old as the OEIS entry A179009 [OEI] (due to David S. Newman in 2011) and has been formally introduced in a paper by Aragona et al. [ACGS22], where unrefinability appeared in a natural way in connection to some subgroups in a chain of normalizers [ACGS21]. The authors proved that the generators of such subgroups are parametrized by some unrefinable partitions satisfying additional conditions on the minimal excludant. Some first combinatorial equalities regarding unrefinable partitions for triangular and non-triangular numbers have been shown recently [ACCL22, ACC22]. The notion of minimal excludant, which frequently appears also in combinatorial game theory [Gur12, FP15], has been studied in the context of integers partition by other authors [AN19, BM20, HSS22].

1.2. Organization of the paper
The remainder of this document is arranged as follows. In Sec. 2 we introduce our notation and show some preliminary results. In Sec 3 we present the verification algorithm (cf. Algorithm 1), prove its correctness and discuss its complexity (cf. Theorem 13). The enumerating algorithm (cf. Algorithms 3 and 4) and the relative complexity analysis (cf. Theorem 18) are presented in Sec 4, which concludes the paper.

2. Notation and preliminaries

In this paper we use a non-conventional representation for partitions into distinct parts. Namely, together with the integers which belongs to the partition, we explicitly mark all the missing parts up to a certain value. Formally, we call a *sequence of parts* a sequence of integers $\lambda = (v_i)_{i \geq 1}$ such that

1. there exists $\ell \geq 1$ such that for $i > \ell$ we have $v_i = 0$,
2. for each $1 \leq i \leq \ell$ we have that v_i is either i or 0.

Each null v_i is displayed using the symbol $\star$, and we denote for brevity $\lambda = (v_1, v_2, \ldots, v_\ell)$, where ℓ is as above, and i is called the *index* of v_i. Defining

$$\mathrm{sum}(\lambda) = \sum_{i=1}^{\infty} v_i = \sum_{v_i=i} v_i < \infty,$$

where the symbol $\star$ is treated as zero, any such sequence naturally represents a partition of the integer $N = \mathrm{sum}(\lambda)$ into distinct parts. Therefore, when it is not ambiguous, we use the terms partition and sequence of parts interchangeably. We call *length* of the sequence of parts the integer $\ell = |\lambda|$. If $x \leq |\lambda|$ and v_x is equal to x then we say that x is a part of λ, and say that $x \in \lambda$. Otherwise we say that $x \notin \lambda$.

The representation of a partition as a sequence of parts is clearly not unique and this is exactly the reason why we need to introduce this formalism. We use this representation to distinguish, e.g., that $(1, 2, \star, \star, 5)$ is a prefix of $(1, 2, \star, \star, 5, 6, 7, \star, 9)$ while $(1, 2, \star, \star, 5, \star)$ is not, even though they both represent the partition of 8 with parts 1, 2, and 5. This distinction will be useful while discussing the process of verifying and enumerating partitions. Indeed, our algorithms will take sequences of parts as inputs and will

1. test whether a sequence of parts is unrefinable,
2. enumerate all unrefinable sequences of parts of a given N which does not end with $\star$.

Our representation highlights the numbers that are not included among the parts of the partitioned number N. We say that x is a *missing part* of λ when $x \leq |\lambda|$ and the xth coordinate of λ is $\star$.[1] The smallest missing part in λ (i.e., the index of the leftmost $\star$ in the sequence) is called the *minimal excludant* of λ, and is denoted by $\mathrm{mex}(\lambda)$. It is customary in the relevant literature to set $\mathrm{mex}(\lambda) = 0$ when there is no such minimal excludant, i.e., when λ does not have any $\star$.

For conveniency, we abuse notation and we define $\lambda \cup \{\star\}$ as the sequence of parts obtained concatenating a $\star$ to λ. In the same way, we denote $\lambda \cup \{x\}$ as the concatenation of λ with number x, the latter operation being well defined only when $x = |\lambda| + 1$.

Let us now define the notion of refinability in the context of sequences of parts.

Definition 1 (Refining). Let us consider a sequence of parts

$$\lambda = (v_1, v_2, \ldots, v_\ell).$$

A part v_r is *refinable* when the equation $r = r_1 + \cdots + r_t$ holds for some $t > 1$, with $v_r = r$ and $v_{r_1}, \ldots, v_{r_t}$ all equal to $\star$. Accordingly, the equation $r = r_1 + \cdots + r_t$ is

---

[1] Notice that an integer $x \notin \lambda$ is not called a missing part when $x > |\lambda|$.

a *refinement* of v_r. A part that has no refinement is called *unrefinable*. A partition λ is *refinable* if some of its part admits a refinement, and it is *unrefinable* otherwise.

Since we are discussing algorithmic matters, it is a legitimate concern whether the length of a sequence of parts might be longer than the length of a simple list of parts (missing parts excluded). However it turns out that, if representing unrefinable partitions, their sizes are actually similar, as the following known lemma [ACCL22] and its corollary show.

Lemma 2 ([ACCL22]). *Let $\lambda = (v_1, v_2, \ldots, v_\ell)$ be a sequence of parts with $v_\ell = \ell$ representing an unrefinable partition. Then the number of missing parts in λ is at most $\lfloor \ell/2 \rfloor$.*

Corollary 3. *Let $\lambda = (v_1, v_2, \ldots, v_\ell)$, with $v_\ell = \ell$, be a sequence of parts representing an unrefinable partition and let k be the number of parts in λ. Then*

$$k \leq \ell \leq 2k.$$

Proof. Let k and m be respectively the number of parts and missing parts in λ. A sequence of parts of length ℓ can have at most ℓ parts, therefore $k \leq \ell$. For the second inequality we have that m is at most $\lfloor \ell/2 \rfloor$ by the previous lemma, and furthermore that $\ell = m + k$. Hence $\lceil \ell/2 \rceil \leq k$, which implies $\ell \leq 2k$. $\square$

At first glance it may seem that checking refinability for a partition λ should be computationally expensive, since, according to Definition 1, we potentially need to check all sums of two or more indexes that corresponds to $\star$ in λ. However, it is not hard to realize that if a λ is refinable, then there exists a part r with some refinement of the form $r = a + b$.

Proposition 4. *If a partition λ has some refinement, then its smallest refinable part r has a refinement of the form $r = a + b$.*

Proof. Let r be the smallest refinable part for which there exists some refinement $r = \nu_1 + \cdots + \nu_t$. If $t = 2$ there is nothing to prove. Otherwise, let us fix $a = \nu_1$ and $b = \nu_2 + \cdots + \nu_t$. If $b \in \lambda$, then $b = \nu_2 + \cdots + \nu_t$ would be a refinement itself, but $b < r$ and this would violate the minimality of r, hence b is not a part of λ. This shows that $r = a + b$ is indeed a refinement of λ. $\square$

From Proposition 4 it is easy to obtain a polynomial algorithm that checks refinability of a sequence of parts λ in time $O(\ell^3)$: for every part of λ, test whether it is the sum of two smaller missing parts. We will show in the next section how this algorithm can be improved.

3. A faster algorithm to check refinability

In this section we introduce Algorithm 1 to check refinability of sequences of parts. This algorithm is faster compared to the naïve one discussed above. Our improvement comes from the key observation that whenever a and b are missing elements in an unrefinable partition λ, then for any two integers $x, y > 0$ we have that $xa + yb$ cannot be in λ. This leads to the following idea: once we know $\mu = \mathrm{mex}(\lambda)$, if we find out that x is another missing part, then none of $x + \mu, x + 2\mu, x + 3\mu, \ldots$ can be parts of λ, unless λ is refinable. Hence, for each $0 \leq j < \mu$ we just need to keep trace of the first missing part (greater than μ) that is equal to $j \pmod{\mu}$ to completely characterize all parts that would violate unrefinability.

Example 5. Assume we want to check whether a sequence λ is refinable and that λ has $(1, 2, 3, \star, 5, 6, \star, 8, 9, \star)$ as prefix. We know that $\mathrm{mex}(\lambda) = 4$, and since $7 \notin \lambda$, the numbers $\{11, 15, 19, \ldots\}$ cannot be parts of λ unless it is refinable. Similarly, integers as $\{14, 18, 22, \ldots\}$ are forbidden as well because $10 \notin \lambda$. The integer 17 is forbidden too, because it is $7 + 10$, and therefore also $\{21, 25, 29, \ldots\}$ are forbidden. Essentially, after reading the first 10 elements of λ, we already know that $7 + 4t$ cannot be in λ for all $t \geq 1$, unless the sequence is refinable, i.e. that every integer larger than 7 which belongs to the residue class of 3 modulo $\mu = 4$ would violate unrefinability. The same for $10 + 4t$ for $t \geq 1$ and for $17 + 4t$ for $t \geq 0$. The distinction between the case $t \geq 0$ and $t \geq 1$ will be clear in the next paragraph.

The algorithm that we are about to present scans the values of $v_1, v_2, \ldots, v_\ell$ from index $\mu + 1$ to ℓ, while maintaining the information about which numbers in each residue class modulo μ are *forbidden*, i.e., are either known missing parts or violate unrefinability if met later when scanning the sequence.

More precisely, this is accomplished by defining μ counters p_j for each residue class $0 \leq j < \mu$ modulo μ. We initially set $p_j := \infty$, meaning there is no forbidden number in the residue class j. Going from $\mu + 1$ to ℓ, the values of each p_j is updated every time a missing part is met in the sequence of parts. The invariant is when we reach position v_t in the sequence, the values

$$p_j \quad p_j + \mu \quad p_j + 2\mu \quad p_j + 3\mu \quad \ldots$$

are all forbidden in any unrefinable sequence of parts starting with the same prefix $v_1, \ldots, v_t$. This is indeed enough: we can update p_js so that when the scan reaches v_ℓ without meeting any forbidden value, then λ is unrefinable.

Algorithm 1 proceeds as follows. It starts by finding the minimal excludant μ, and if none exists, then the partition is obviously unrefinable. Then it checks all integers from index $\mu + 1$ to v_ℓ in order. For a given number r in this sequence there are two possibilities:

- $r \in \lambda$ and therefore we need to check if it has a refinement;

- $r \notin \lambda$ and then we update our knowledge of which numbers would contradict refinability, if met.

In the second case, such knowledge is represented by the numbers $p_0, \ldots, p_{\mu-1}$ which are updated at each iteration of the main loop of the algorithm. If p_j is finite, then it is equal to j modulo μ.

Algorithm 1: VERIFY (an algorithm to check refinability)

Input : $\lambda = (v_1, v_2, \ldots, v_\ell)$
Returns: REFINABLE or UNREFINABLE

1   $\mu \leftarrow \text{mex}(\lambda)$
2   **if** $\mu = 0$ **then return** UNREFINABLE
3   $\vec{p} = (p_0, \ldots, p_{\mu-1}) \leftarrow (\infty, \infty, \ldots, \infty)$
4   **for** r *in* $(\mu + 1), \ldots, \ell$ **do**
5      $j \leftarrow r \pmod{\mu}$
6   **if** $v_r = r$ *and* $v_r \geq p_j$ **then return** REFINABLE
7   **if** $v_r = \star$ **then** $\vec{p} \leftarrow$ UPDATE$(\vec{p}, r)$
8   **end**
9   **return** UNREFINABLE

Algorithm 2: UPDATE (improves p_js after a new missing part r is discovered)

Input : $\vec{p} = (p_0, \ldots, p_{\mu-1})$, r a newly discovered missing part
Returns: $\vec{p} = (p_0, \ldots, p_{\mu-1})$, updated

10   $j \leftarrow r \pmod{\mu}$
11   **if** $r > p_j$ **then**
12      $t \leftarrow r + p_j \pmod{\mu}$
13      $p_t \leftarrow \min(p_t, r + p_j)$
14   **else**
15      $p_j \leftarrow r$
16   **for** j' *in* $\{1, \ldots, \mu - 1\} \setminus \{j\}$ **do**
17          $t \leftarrow j + j' \pmod{\mu}$
18          $p_t \leftarrow \min(p_t, p_j + p_{j'})$
19   **end**
20   **end**
21   **return** $(p_0, \ldots, p_{\mu-1})$

Algorithm 1 uses a subroutine called UPDATE (cf. Algorithm 2). Once we discover a new missing part r, two different circumstances can occur, and they are addressed accordingly by UPDATE. Precisely, either r is not the smallest missing part in its residue class, and in this case we just need to see how r interacts with smaller missing parts in the same residue class (**if** branch); or r is the smallest missing part in its residue class, and then we need to check how this influences all other missing parts (**else** branch). We give examples for both cases, precisely Example 6 for the **else** branch and Example 7 for the **if** branch.

Example 6. Let $\lambda = (1, 2, 3, 4, 5, \star, 7, 8, \star, \star, 11, 12, 13, \star, \ldots)$ and consider all calls to UPDATE when Algorithm 1 reaches 14 in λ. We have $\mu = 6$. The first call sets $p_3 = 9$. The second call sets $p_4 = 10$, and since $19 = 9 + 10$, we need 19 to be forbidden as well. This happens in the **for** loop that sets $p_1 = 19$. At this stage we have $(p_0, p_1, p_2, p_3, p_4, p_5) = (\infty, 19, \infty, 9, 10, \infty)$. The third call happens when the scan reaches 14. Here the algorithm sets $p_2 = 14$, and afterward the **for** loop in line 15 computes the forbidden values

$$19 + 14 = 33, \quad 9 + 14 = 23, \quad 10 + 14 = 24.$$

The information that 33 is forbidden is included in p_3 (previously set to 9), while the information $p_5 = 23$ and $p_0 = 24$ is newly determined. When 14 is reached and processed, the information on forbidden numbers is represented by

$$(p_0, p_1, p_2, p_3, p_4, p_5) = (24, 19, 14, 9, 10, 23).$$

The partition λ may continue either with 15 or with $\star$. Notice that $15 = 3 \pmod 6$ and $15 > p_3 = 9$, therefore $15 \in \lambda$ would prove refinability (indeed $15 = 6 + 9$). Therefore λ can only continue with $\star$.

Example 7. Let $\lambda = (1, 2, 3, \star, 5, 6, \star, 8, 9, \star, \star, 12, 13, \star, \ldots)$ and consider the call to UPDATE when Algorithm 1 reaches 14 in λ. We have $\mu = 4$. By the time the algorithm scans position 14 we know that the sequence misses parts 10 and 14, therefore 24 must be forbidden as well. Indeed in this call we have $r = 14$ and $p_2 = 10$, and line 13 runs and sets p_0 to 24 as desired.

In the rest of the section we discuss the correctness and complexity of Algorithm 1. First we prove the correctness of the algorithm on unrefinable and refinable sequences of parts separately, then we discuss its complexity.

Lemma 8. *Algorithm 1 outputs* UNREFINABLE *on every unrefinable λ.*

Proof. Consider an unrefinable sequences of parts λ. We start by proving that when the algorithm assigns a value w to some p_j, it means that $w \notin \lambda$. We prove this by induction on the iterations of the main loop at line 4. The base of the induction trivially holds because before the loop all p_j are set to ∞.

For the inductive step we discuss all the ways these assignments occur in the UPDATE function described in Algorithm 2. If we set p_j to w at line 15, then $w \notin \lambda$ because UPDATE would have been called when $v_w = \star$. If we update p_t to value w either at line 13 or at line 18, we already know that $v_r = \star$ and, by induction, that p_j and $p_{j'}$ are not in λ. Since λ is unrefinable, the new value of p_t (namely w) cannot be in λ either.

We just proved that, at any moment in the algorithm, every finite valued p_j is not in λ. We improve this by showing that the same holds for $p_j + t\mu$ for $t \geq 0$, by induction on t. The case $t = 0$ is what we have proved so far. Assuming $p_j + t\mu \notin \lambda$, then by unrefinability the same holds for $p_j + (t + 1)\mu$.

To conclude, observe that the only possible way for Algorithm 1 to be incorrect is to return at line 6. This happens when there is some $v_r = r$ which is greater than both μ and p_j, and that it is equal to j modulo μ. Hence $r = p_j + t\mu$ for some $t > 0$. But we just showed that these values are not in λ, therefore the algorithm cannot return at line 6. $\qquad\square$

To prove the correctness of Algorithm 1 on refinable partition we use the following two propositions.

Proposition 9. *Consider the iteration r of the main loop of Algorithm 1, where $r \notin \lambda$ and $r = j \pmod{\mu}$. After that iteration, $p_j \leq r$.*

Proof. UPDATE is called, and when it reaches line 11 either the test $r > p_j$ passes, or p_j is set to r. Hence at the end of iteration r we have that $p_j \leq r$. Successive iterations can only decrease the value of p_j. $\qquad\square$

Proposition 10. *Assume that Algorithm 1 reaches iteration r, and let j be the residue class of r modulo μ. The assignment at line 15 of UPDATE is executed if and only if r is the smallest number strictly greater than μ in residue class j with $r \notin \lambda$.*

Proof. If there is a smaller $r' \notin \lambda$ in the same residue class j, then by proposition 9 we have $p_j \leq r' < r$. In that case, line 15 is not reached.

In the other direction, let r be the smallest number in the residue class j for which $r \notin \lambda$. If $r \leq p_j$ at the time the main loop reaches iteration r, then line 15 is executed. Otherwise, p_j must have been assigned to the current value at lines 13 or 18 in some iteration r' earlier than r. In both cases the assigned value is strictly larger than r'. Hence we have $r' < p_j < r$ and therefore $p_j \in \lambda$ by hypothesis. Algorithm 1 returns REFINABLE at iteration p_j or earlier, and therefore never reaches iteration r as assumed. $\qquad\square$

Now we can prove the correctness in the refinable case.

Lemma 11. *Algorithm 1 outputs REFINABLE on every refinable λ.*

Proof. By Proposition 4 we know that the smallest refinable part r is refinable as $a + b$ with $a, b \notin \lambda$. Let us denote $j_a = a \pmod{\mu}$, $j_b = b \pmod{\mu}$, and $j_r = r \pmod{\mu}$. Clearly $j_r = j_a + j_b \pmod{\mu}$.

If the algorithm does not reach iteration r, it must be because it returned REFINABLE earlier and so there is nothing to prove. Otherwise let us show that it must return RE-FINABLE at iteration r.

The case of $j_a = 0$ is simple: we have that $j_r = j_b$ and $p_{j_r} \leq b$ by Proposition 9. Therefore we get $r > b \geq p_{j_r}$ and the algorithm returns at line 6. The case of $j_b = 0$ is symmetric.

For the remaining case of $j_a \neq 0$ and $j_b \neq 0$ we split into two further subcases: when $j_a = j_b$ and when $j_a \neq j_b$.

When $j_a \neq 0$, $j_b \neq 0$ and $j_a = j_b$, we may assume without loss of generality that $a < b$. By the time the algorithm reaches iteration b, we have that $p_{j_a} \leq a$ because of Proposition 9. The test at line 11 at call UPDATE$(\vec{p}, b)$ can be rewritten as $b > p_{j_a}$, hence the value p_{j_r} is assigned to a number smaller or equal than $r = a + b$ in the residue class of j_r, in line 13. At the time the main loop reaches the iteration r, the algorithm reaches line 6 and returns REFINABLE.

When $j_a \neq 0$ and $j_b \neq 0$ and $j_a \neq j_b$, we need to consider the smallest missing elements a' and b' that are equal to j_a and j_b, respectively, modulo μ. We assume without loss of generality that $a' < b'$. When the algorithm reaches iteration b' we have that $p_{j_a} \leq a'$ because of Proposition 9, and that assignment $p_{j_b} \leftarrow b'$ in line 15 is executed because of Proposition 10. In the for loop right after line 15, we know that $j_a \in \{1, \dots, \mu - 1\} \setminus j_b$, therefore we get that p_{j_r} is set to some value smaller or equal to $a' + b'$ and in particular to a value smaller equal than r. In the successive iteration the value never increases, and at iteration r we know that line 6 gets executed. $\square$

Lemma 12 (Running time). *Algorithm 1, executed on the sequence of parts $\lambda = (v_1, v_2, \dots, v_\ell)$ with $\mu = \mathrm{mex}(\lambda)$, runs in time $O(\ell + \mu^2)$.*

Proof. The initialization of the p_js and the computation of $\mu = \mathrm{mex}(\lambda)$ takes $O(\ell)$ steps. The main loop in Algorithm 1 is executed at most ℓ times. The inner loop in Algorithm 2 is executed in at most μ of them. The total running time is therefore $O(\ell + \mu^2)$. $\square$

Putting together Lemmas 8, 11 and 12 we obtain the main theorem of this section.

Theorem 13. *Algorithm 1, executed on the sequence of parts $\lambda = (v_1, v_2, \dots, v_\ell)$, returns UNREFINABLE if and only if the partition λ is unrefinable. Its complexity is $O(\ell + \mu^2)$, where $\mu = \mathrm{mex}(\lambda)$.*

We conclude this section with some remarks on Algorithm 1, which will be important in the next section regarding the enumerating algorithm. First of all, notice that if its main loop reaches an iteration r where $p_j \leq r$ for all $0 \leq j < \mu$, then the only possible way to extend λ is adding an arbitrary number of $\star$. Any additional part would lead Algorithm 1 to return REFINABLE. Moreover:

Definition 14. Let $\lambda = (v_1, v_2, \dots, v_\ell)$ be an unrefinable sequence of parts with $\mathrm{mex}(\lambda) = \mu$ and let $\vec{p}$ be the values computed by Algorithm 1 on λ. We say that λ is *saturated* when

$$|\{p_j \leq \ell \ : \ 0 \leq j < \mu\}| = \mu.$$

Proposition 15. *Consider an unrefinable partition $N = \lambda_1 + \lambda_2 + \dots + \lambda_k$ with minimum excludant μ. There are at most μ unrefinable sequence of parts whose sum is N, which are not saturated, and which have parts $\{\lambda_1, \lambda_2, \dots, \lambda_k\}$.*

Proof. Assume without loss of generality that λ_k is the largest part. There is a unique sequence of parts λ with parts $\{\lambda_1, \lambda_2, \ldots, \lambda_k\}$ and length λ_k. Any other partition is obtained adding stars to the sequence, but after adding μ stars the resulting sequence of parts must be saturated. □

4. How to enumerate unrefinable partitions

The verification via Algorithm 1 of a sequence of parts $\lambda = (v_1, v_2, \ldots, v_\ell)$ with $\mu = \text{mex}(\lambda)$ starts by scanning the interval $\mu + 1, \ldots, \ell$. Up to the point when some index r is under scrutiny, the algorithm uses no information about the elements of λ of successive indexes. More concretely, the values $\vec{p}$ computed at iteration r are completely determined by the same old values computed at iteration $r - 1$ and by the fact that r is either in λ or not. Therefore we can design the enumeration process as the visit of the tree of all possible sequences of parts, so that the verification algorithm is run on the sequence corresponding to any branch of tree (see Figure 1). A branch is pruned as soon as the the corresponding sequence has no possible extensions that are unrefinable and of sum at most N. When the sum of a sequence corresponding to a surviving branch equals the goal value N, the sequence is returned as output.

It is convenient to enumerate separately all unrefinable partitions of N that have the same minimal excludant. Given N, we set n as the largest positive integer such that

$$\sum_{i=1}^{n} i \le N$$

and then we partition the search space of sequences of parts according to prefixes:

$$
\begin{aligned}
\lambda^\dagger &= (1, 2, 3, 4, \ldots, n-2, n-1, n), \\
\lambda^n &= (1, 2, 3, 4, \ldots, n-2, n-1, \star), \\
\lambda^{n-1} &= (1, 2, 3, 4, \ldots, n-2, \star), \\
&\vdots \\
\lambda^4 &= (1, 2, 3, \star), \\
\lambda^3 &= (1, 2, \star), \\
\lambda^2 &= (1, \star), \\
\lambda^1 &= (\star).
\end{aligned}
\tag{1}
$$

If N is triangular, i.e. if $N = n(n + 1)/2$, then the sequence $\lambda^\dagger$ itself is the unique unrefinable partition of N with no minimal excludant, and it must be in the output of the enumeration. If N is not triangular, i.e. if $n(n + 1)/2 < N$, there is no unrefinable partition with prefix $\lambda^\dagger$: any additional part would make the sum exceed N.

Any other unrefinable partition of N must have minimal excludant $1 \le \mu \le n$, and for a given value of μ there is a one-to-one correspondence between these partitions and the sequences of parts λ that
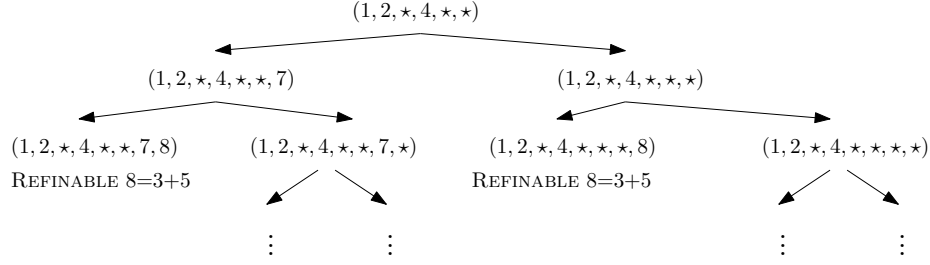
- are unrefinable,

Figure 1: The branching from the sequence $(1, 2, \star, 4, \star, \star)$. For any two sequences in the tree, the running of Algorithm 1 proceeds identically up to the point that the corresponding branches diverge.

- have $\mathrm{sum}(\lambda) = N$,

- have prefix $\lambda^\mu = (1, 2, 3, 4, \ldots, \mu - 1, \star)$,

- have $v_\ell = \ell$ (i.e. not ending with $\star$).

In order to enumerate them, we describe the recursive algorithm ENUMERATE (cf. Algorihtm 3).

Algorithm 3: ENUMERATE

Input : N

$\qquad$ $\lambda = (1, 2, 3, \ldots, \mu - 1, \star, v_{\mu+1}, v_{\mu+2}, \ldots)$, unrefinable

$\qquad$ $\vec{p} = (p_0, \ldots, p_{\mu-1})$

Output : All unrefinable partitions of N with prefix λ, not ending with $\star$.

---
1 $r \leftarrow |\lambda| + 1$

2 $j \leftarrow r \pmod{\mu}$

$\quad$ // Cases when we extend with r, if possible

3 **if** $r < p_j$ *and* $\mathrm{sum}(\lambda) + r = N$ **then output** $\lambda \cup \{r\}$

4 **if** $r < p_j$ *and* $\mathrm{sum}(\lambda) + r < N$ **then** ENUMERATE($N, \lambda \cup \{r\}, \vec{p}$)

$\quad$ // Case when we extend with $\star$

5 $\vec{p} \leftarrow$ UPDATE($\vec{p}, r$)

6 **if** $\lambda \cup \{\star\}$ *is not saturated* **then** ENUMERATE($N, \lambda \cup \{\star\}, \vec{p}$)

ENUMERATE starts with a sequence of parts λ with $\mathrm{mex}(\lambda) = \mu$ and extends it in all possible ways in a binary tree-like fashion (cf. Figure 1). When visiting the node of the tree corresponding to sequence λ, the algorithm decides whether to branch on $\lambda \cup \{v\}$, and successively whether to branch on $\lambda \cup \{\star\}$. Therefore, the tree is visited in lexicographic order. A branch is pruned either when a partition of N is reached, when an extension goes over the goal value N, when it introduces a refinable part, or when the sequence of parts is saturated according to Definition 14, and therefore no non-trivial extension would ever be unrefinable.

Walking along the tree, we update the values $\vec{p}$ using the same UPDATE function that we used in Algorithm 1. The idea is that the computation done by the recursive process on the sequence corresponding to some path is the same as the one done by Algorithm 1 on the same sequence. Formally we consider

- $P1$ the set of pairs $(\lambda, \vec{p})$ such that λ is unrefinable and not saturated, $\mathrm{mex}(\lambda) = \mu$, $\mathrm{sum}(\lambda) < N$, and such that running Algorithm 1 on λ computes the values $\vec{p}$;

- $P2$ the set of pairs $(\lambda, \vec{p})$ such that the execution of ENUMERATE$(N, \lambda^\mu, (\infty, \ldots, \infty))$ produces a recursive call ENUMERATE$(N, \lambda, \vec{p})$.

Lemma 16. *The two sets $P1$ and $P2$ are equal.*

Proof. We prove this statement by induction on the length of the sequence. For the base case, the sequence of parts λ^μ, paired with all p_js set to ∞, is both in $P1$ and $P2$ because $\mathrm{sum}(\lambda^\mu) < N$.

For the induction step, consider the pair $(\lambda, \vec{p})$ for which we know that λ is unrefinable, is not saturated, that $\mathrm{mex}(\lambda) = \mu$ and $\mathrm{sum}(\lambda) < N$, and that a recursive call ENUMERATE$(N, \lambda, \vec{p})$ occurs.

For the extension $\lambda \cup \{r\}$ the values of $\vec{p}$ do not change in both algorithms, therefore if λ is not saturated, neither is $\lambda \cup \{r\}$. The pair $(\lambda \cup \{r\}, \vec{p})$ is in $P1$ if and only if $r < p_j$ for $r = j \pmod{\mu}$ and $\mathrm{sum}(\lambda) + r < N$. But these are exactly the same condition for the recursive call ENUMERATE$(N, \lambda \cup \{r\}, \vec{p})$.

Considering the extension $\lambda \cup \{\star\}$, this is of course as unrefinable as λ and the sum does not change either. Let $\vec{q} \leftarrow$ UPDATE$(\vec{p}, r)$. The pair $(\lambda \cup \{\star\}, \vec{q})$ is in $P1$ if and only if $\lambda \cup \{\star\}$ it is not saturated , and that is the exact same condition for the recursive call ENUMERATE$(N, \lambda \cup \{\star\}, \vec{q})$ to happen. $\square$

We are ready to show that, provided the appropriate input, ENUMERATE correctly produces all the unrefinable partitions of N with a given minimal excludant μ.

Lemma 17. *The recursive algorithm ENUMERATE$(N, \lambda^\mu, \vec{p})$, where $\vec{p} = (p_0, \ldots, p_{\mu-1})$ are all set to ∞, outputs the unrefinable sequences of parts whose sum is N with minimum excludant μ, and without $\star$ in the last position.*

Proof. By definition, the output of the enumeration only includes sequences of parts of N, not ending with $\star$. We need to prove that the output includes all unrefinable ones and no refinable ones.

Any unrefinable sequence of parts of N with minimal excludant μ, not ending with $\star$, can be written as $\lambda \cup \{r\}$ where $\mathrm{mex}(\lambda) = \mu$ and $\mathrm{sum}(\lambda) = N - r < N$. By Lemma 16, there is a recursive call ENUMERATE$(N, \lambda, \vec{p})$ where $\vec{p}$ are the values computed by Algorithm 1 on λ. By the correctness of Algorithm 1 it must be $r < p_j$ for $j = r \pmod{\mu}$ since $\lambda \cup \{r\}$ is unrefinable. Hence the call ENUMERATE$(N, \lambda, \vec{p})$ outputs $\lambda \cup \{r\}$.

Now we want to show that no refinable sequence of parts of N is in the output. Consider the shortest prefix $\lambda \cup \{r\}$ of any such sequence where λ is unrefinable and $\lambda \cup \{r\}$ is refinable. It still holds that $\text{mex}(\lambda) = \mu$ and that $\text{sum}(\lambda) < N$, therefore, by Lemma 16, there is a recursive call $\text{ENUMERATE}(N, \lambda, \vec{p})$ where $\vec{p}$ are the values computed by Algorithm 1 on λ. By the correctness of Algorithm 1, it must be $r \geq p_j$ for $j = r \pmod{\mu}$ since $\lambda \cup \{r\}$ is refinable. Hence ENUMERATE skips $\lambda \cup \{r\}$ and all its extensions. $\square$

We are ready to describe the algorithm that enumerates all unrefinable partitions of N.

Algorithm 4: UNREFINABLEPARTITIONS (enumerate all unrefinable partitions of N)

Input : N
Output : All unrefinable partitions of N.

1   $n \leftarrow$ largest n such that $\sum_{i=1}^{n} \leq N$
2   **if** $\sum_{i=1}^{n} = N$ **then output** $(1, 2, 3, \ldots, n)$
3   **for** μ *in* $\{n, n-1, \ldots, 2, 1\}$ **do**
4      $\vec{p} = (p_0, \ldots, p_{\mu-1}) \leftarrow (\infty, \infty, \ldots, \infty)$
5      $\lambda^{\mu} \leftarrow (1, 2, 3, \ldots, \mu-1, \star)$
6      $\text{ENUMERATE}(N, \lambda^{\mu}, \vec{p})$
7   **end**

Theorem 18. *Algorithm 4 outputs all unrefinable partitions of N in time $O(N) \cdot U(N)$, where $U(N)$ is the number of unrefinable partitions of integers $< N$.*

Proof. The algorithm correctly outputs $\lambda^{\dagger}$ if and only if N is a triangular number. All other unrefinable partitions have, as discussed above, a minimal excludant $1 \leq \mu \leq n$, where n is defined as in the algorithm. Any such partition is uniquely represented by an unrefinable sequence of parts not ending with a $\star$ and having as prefix the appropriate λ^{μ}. By Lemma 17 these sequences are produced by the calls to ENUMERATE in the main cycle. Hence the algorithm correctly enumerates the unrefinable partitions of N.

To discuss the runtime, observe that each of the calls to ENUMERATE in the main cycle makes a number of recursive call equal to the set $P1$ discussed in Lemma 16. Collecting together all the call of all values of μ, this is by definition the number of unrefinable and unsaturated sequences of parts of a number $< N$. For any unrefinable partitions of a number $\leq N$, by Proposition 15 there are at most $O(\sqrt{N})$ such sequences of parts. Hence the process produces at most $O(\sqrt{N})$ recursive calls for each unrefinable partitions of a number $< N$. The cost of each recursive call is dominated by the cost of the call to UPDATE, hence $O(\sqrt{N})$, plus the possible cost to output an actual unrefinable partition of N, when encountered, which is again at most $O(\sqrt{N})$.

Therefore the total cost of the $O(N)$ times the number[2] of all unrefinable partitions of

---

[2]At the time of writing, $U(N)$ is unknown.

integers less than N. □

An implementation in C++ of the presented algorithms, made available online at `https://github.com/MassimoLauria/a179009`, has been used to compute the number of unrefinable partitions of N, with N up to 1500. The full list can be found online. Some of the data are also available here in Table 1 (cf. also [OEI, `https://oeis.org/A179009`]).

N	unref. partitions of N	N	unref. partitions of N
10	1	400	57725
20	7	500	275151
30	5	1000	84527031
40	9	1100	220124218
50	15	1200	559471992
100	104	1300	1383113838
200	1616	1400	3357904448
300	11801	1500	7734760269

Table 1: The number of unrefinable partition for some integers up to 1500. A full list up to 1500 is available at `https://massimolauria.net/perm/unrefinable_01500.txt`.

Acknowledgments

References

[ACC22] R. Aragona, L. Campioni, and R. Civino. The number of maximal unrefinable partitions. Available at `https://arxiv.org/abs/2206.04261`, 2022.

[ACCL22] R. Aragona, L. Campioni, R. Civino, and M. Lauria. On the maximal part in unrefinable partitions of triangular numbers. *Aequationes Math.*, 2022. doi:10.1007/s00010-022-00890-6.

[ACGS21] R. Aragona, R. Civino, N. Gavioli, and C. M. Scoppola. Rigid commutators and a normalizer chain. *Monatsh. Math.*, 196(3):431–455, 2021.

[ACGS22] R. Aragona, R. Civino, N. Gavioli, and C. M. Scoppola. Unrefinable partitions into distinct parts in a normalizer chain. *Discrete Math. Lett.*, 8:72–77, 2022.

[AN19] G. E. Andrews and D. Newman. Partitions and the minimal excludant. *Ann. Comb.*, 23(2):249–254, 2019.

[And76] G. E. Andrews. *The theory of partitions*. Encyclopedia of Mathematics and its Applications, Vol. 2. Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1976.

[BM20] C. Ballantine and M. Merca. The minimal excludant and colored partitions. *Sém. Lothar. Combin.*, 84B:Art. 23, 12, 2020.

[FP15] A. S. Fraenkel and U. Peled. Harnessing the unwieldy MEX function. In *Games of no chance 4*, volume 63 of *Math. Sci. Res. Inst. Publ.*, pages 77–94. Cambridge Univ. Press, New York, 2015.

[Gur12] V. Gurvich. Further generalizations of the Wythoff game and the minimum excludant. *Discrete Appl. Math.*, 160(7-8):941–947, 2012.

[HSS22] B. Hopkins, J. A. Sellers, and D. Stanton. Dyson's crank and the mex of integer partitions. *J. Combin. Theory Ser. A*, 185:Paper No. 105523, 10, 2022.

[OEI] The On-Line Encyclopedia of Integer Sequences. Published electronically at https://oeis.org. Accessed: 2021-11-01.