Introduction

Recently, I fell down the rabbit hole that is Ulm Number Generation[1]. Ulm Numbers are a sequence that starts $\begin{bmatrix}1 & 2\end{bmatrix}$, and continues by finding the **smallest** number that is:

- the sum of two *distinct* numbers already in the sequence, and
- has only one way of being constructed, and
- is greater than any number in the sequence so far.

So the sequence $\begin{bmatrix}1 & 2\end{bmatrix}$ is extended to $\begin{bmatrix}1 & 2 & 3\end{bmatrix}$, which is then extended to $\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}$. Now the candidate next number 5 can be made up by $1+4$, and $2+3$, so it is excluded, and the sequence is extended to $\begin{bmatrix}1 & 2 & 3 & 4 & 6\end{bmatrix}$, and so on.

It turns out that Ulm Numbers have very mysterious properties (more on this later). What originally piqued my interest was the challenge of how most efficiently to generate this sequence for up to 1000 or more entries.

Brute Force

Example

The obvious way is to find the next number is to compute the sums of all distinct number pairs, exclude the duplicates, exclude those sums smaller than the last in the sequence, and take the smallest of the remaining candidates

So in the example below:

		$u0$	$u1$	$u2$	$u3$	$u4$	$u5$	$u6$
		1	2	3	4	6	8	11
$u0$	1	.	3	4	5	7	9	**12**
$u1$	2	.	.	5	6	8	10	**13**
$u2$	3	.	.	.	7	9	11	**14**
$u3$	4	.	.	.	.	10	**12**	**15**
$u4$	6	.	.	.	.	.	**14**	**17**
$u5$	8	.	.	.	.	.	.	**19**
$u6$	11	.	.	.	.	.	.	.

the **bold** values are candidate sums (greater than the last value so far (11), but **13** is the smallest non-duplicated value.

The following Python code computes the next Ulm number, given a current sequence in variable **u**

[1] https://en.wikipedia.org/wiki/Ulam_number

```
for l in range(1):
    last = u[-1]
    # set up dictionary that defaults to 0 as key value
    sum_count = defaultdict(int)

    # range over all distict pairs of entries
    for i in range(len(u)):
        for j in range(i+1, len(u)):
            sum = u[i] + u[j]
            sum_count[sum] = sum_count[sum] + 1
        # end for
    # end for

    # find sum values that appear once, and are greater than last entry
    only_one = [k for k in sum_count.keys() if (sum_count[k] == 1) and (k>last)]

    u.append(min(only_one))
# end for
print(u)
```

Listing 1: Brute force approach

Drawbacks

The drawback with the approach above is that it really doesn't scale: the run time goes as n^2, where n is the length of the current Ulm sequence. About the only good aspect of the code above is that it is simple. Indeed, all the "code golf" entries for Ulm sequence generation (especially those based on PAL or its variants) seem to use a similar algorithm.

A better approach

A better approach starts with the observation that we don't need to compute sums that we know as *smaller* than the current last element of the sequence. In the diagram below, $\|$ marks the point past which it is profitable to start computing sums $u[i] + u[j]$, as we run along the row.

		$u0$	$u1$	$u2$	$u3$	$u4$	$u5$	$u6$
		1	2	3	4	6	8	11
$u0$	1	.	.	.	.	.	$\|$	**12**
$u1$	2	.	.	.	.	.	$\|$	**13**
$u2$	3	.	.	.	.	.	$\|$	**14**
$u3$	4	.	.	.	.	$\|$	**12**	?
$u4$	6	.	.	.	.	$\|$	**14**	?
$u5$	8	.	.	.	.	.	$\|$	**19**
$u6$	11	.	.	.	.	.	.	.

In order to find this "tide line", we can use the Python **bisect** package, and the fact that each row of our matrix is sorted in ascending order. For each row corresponding to an element $u[i]$, we can find the index to start summations in $\log(n)$ time. We have n rows (where n is the number of elements in the sequence so far), so building this tide line is order $n * \log(n)$.

We can further observe that for any given row, the sums are also sorted in ascending order, so it only suffices to compute the first (smallest) element of our summation row. In the example above, the ? denotes values we have postponed computing, because there is a smaller known candidate in that row.

Next, we build a Heap Queue (using the Python package **heap**) that contains all the current candidates for smallest sum. In our example, we build a Heap Queue using the list $[12, 13, 14, 12, 14, 19]$. One of the most wonderful properties of the heap is that it is possible to examine the two smallest members of the Heap to check if they are the same. We can `heappop` to get the current smallest element, and examine `heap[0]` to check the next smallest (which may be the same, of course).

So, to build the Heap Queue, Python offers a single call, running in $O(n)$. Each subsequent `headpop` call is $O(\log(n))$.

One small detail is that we build the Heap Queue with tuples giving the current sum, and row and column indices. If we find that the current smallest value is duplicated (and hence is not a valid candidate), we have to discard these duplicate values, and compute a new value to replace the discarded value, as we move right along the row where we discarded the value. So when we discard a value, we have to know which indices to use to compute the new value. So the Heap Queue build actually uses the list $[(12, 0, 6), (13, 1, 6), (14, 2, 6), (12, 3, 5), (14, 4, 5), (19, 5, 6)]$

When we initially examine the Heap Queue, we note that there are duplicated 12 values: the first in the row corresponding the $u[0]$, and the second in the row corresponding to $u[3]$. The first of these is at the end of the row so no computation is needed: the second requires computing the sum of $u[3] + u[6]$, and adding the appropriate tuple $(15, 3, 6)$ to the Heap Queue.

The next time we examine the Heap Queue, we see a 13 value, which is not duplicated. By the properties of the Heap Queue, we know this is the smallest non-duplicated value, and hence a successful candidate. We append 13 to the sequence.

Implementation

Build the Heap Queue

The code below build the initial Heap Queue.

```
        # for each row, we find the index start such that (u(i)+u(start)>last
        # start is the first column such that u(i)+u(start)>last
        n = len(u)
        h = []  # will be priority queue of candidate sums

        # last is the greatest ulam number so far - we must find the smallest sum
of pairs
        # of distinct ulam numbers greater than last
        last = u[-1]

        for i in range(len(u) - 1):
            # starting with u(i), find first value u(k) such that u(k)+u(i)> last
            # ie u(k)>last-u(i)
            start = bisect.bisect_right(u, last - u[i])

            # only examine upper half of matrix
            start = max(start, i + 1)
            candidate_count = candidate_count + (n - 1 - start + 1)

            sum = u[start] + u[i]

            # push the value row, col of this candidate
            h.append((sum, i, start))

        # end for
        # create priority queue.  The smallest candidate sum will always be at
h[0]
        # but may not be unique
        hq.heapify(h)
```

Listing 2: Building the initial Heap Queue

Process the Heap Queue

The code below will process the Heap Queue until a smallest non-duplicated candidate value is found (considering $u[n-1]+u[n]$, shows that there is always at least one unique value to be found for the sequence ending in $u[n]$)

```
        # we have a priority queue of the smallest candidate sums, one from each
row
        # we look at smallest of the row candidate sums: if it is unique
        # then it is our smallest overall sum
        # if it is not unique, then we pop of the duplicated values, and
        # compute the next smallest sum for those rows where we popped off a
duplicated value

        while len(h) > 0:
            pop = hq.heappop(h)
            first = pop[0]
            row = pop[1]
            col = pop[2]

            # to do maintains a memory of the row, column indices of deleted
items
            # that we must replace (if we are not at the end of  the row the
duplicated item was in)
            to_do = []
            if first == h[0][0]:
                # Have nonunique element
                to_do.append((row, col))
                # clear the queue of duplicated values
                while first == h[0][0]:
                    pop = hq.heappop(h)
                    row = pop[1]
                    col = pop[2]
                    to_do.append((row, col))
                # end while

                for r, c in to_do:
                    # advance column along row if not at end
                    # add to priority queue of candidate values
                    if c < (n - 1):  # still more cols left in this row to
consider?
                        sum = u[r] + u[c + 1]
                        hq.heappush(h, (sum, r, c + 1))
                    else:
                        pass
                    # endif
                # end for

            else:
                # Have smallest unique element
                u.append(first)
                break
            # end if
        # end while
        else:
            print("No unique elements found")
        # end else
```

Listing 3: Processing Heap Queue

Results

I grew the Ulam sequence, storing the elapsed time to add each extra 500 elements. Very rough experiments indicate we have about n to $3n$ Heap Queue operations per new element added, giving an approximate complexity of $n \log(n)$.

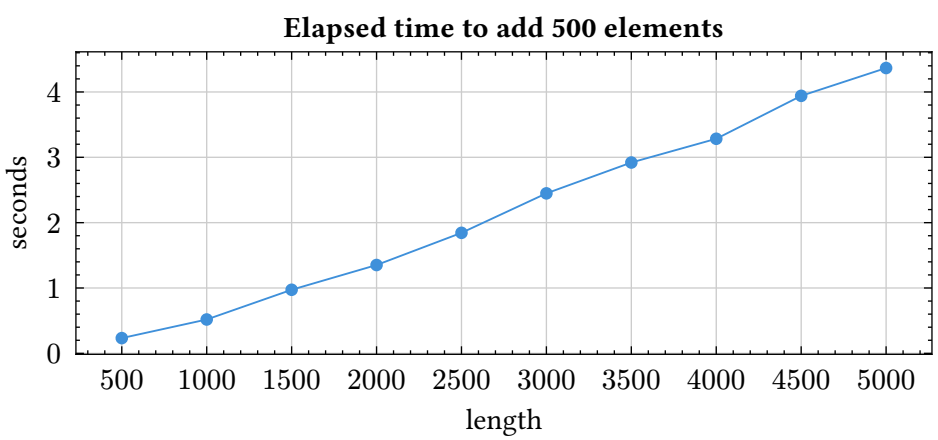

Figure 1: Processing time vs Sequence Length

It appears (from my rather limited testing) that the elapsed processing is roughly linear (and certainly not quadratic).

Mysterious property

If I take my 10,000 element Ulm sequence, and run the following code snippet,

```
for c in u[4:]:
    alpha = 2.5714474995
    y = math.cos(alpha * c)
    if y >= 0.0:
        print(f" for {c}, cos(alpha*c) = {y}")
    # end if
# end for
```

Listing 4: Compute $\cos(\alpha * u[i])$

we get only two values out of 10,000 with positive result!

```
for 47, cos(alpha*c) = 0.09314945116148635
for 69, cos(alpha*c) = 0.07005004608158771
```

The result holds true up to $n = 10^9$