Problem Tutorial: "Crab's Cannon"

This problem has short and beautiful solution. I have discovered a truly marvelous explanation of this, which this document is too small to contain.

That was a joke, so let's start the problem analysis. First, we will solve a simpler problem: we are given a set of integers, and we need to tell if there exists a string with its PPS equal to the given set. After solving this simpler problem, we use the observations from there to solve the harder version. I will try to provide a complete proof, so the analysis will be quite long. It's also possible that the solution can be proved in a shorter way, but I didn't think of it.

Solving a simpler problem

Convention

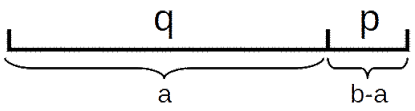
- s' denotes reversed s. So, if $s =$ "`crab`", then $s' =$ "`barc`"
- $a + b$ denotes concatenation of strings a and b
- Palindromic prefix of length x is called just "prefix x". It also means that x is in the PPS.

For simplification, we assume that 0 is present in the PPS. So, we add 0 to the given set. Also, we sort the set beforehands.

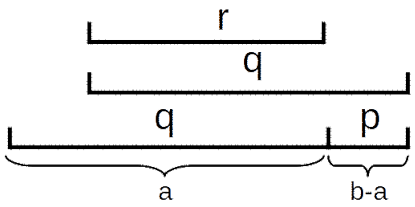
We need to define some criteria which are necessary and sufficient to determine if the set can be PPS of some string.

First of all, 1 must be present in the set (string of length 1 is always a palindrome), otherwise the answer is "NO".

Then consider two numbers from the set, a and b ($a < b$). What information we can obtain from this fact?

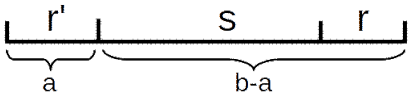


It's easy to see that, since $q + p$ is a palindrome, $q + p$ ends with q'. But, since q' is also a palindrome, $q' = q$:



Then consider the string r. It's the prefix of q and it the same time it's the suffix of q, thus r' is the prefix of $q' = q$. So, we get $r' = r$. r has length $a - (b - a) = 2a - b$. So, we get the palindrome prefix of length $2a - b$.

What if $2a - b < 0$? This is also a valid case, because the string grows more than twice:



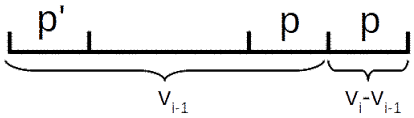
s can be any palindromic string.

So, the first condition (denote it C_1) is as follows: if a and b are in the set, then $2a - b$ is also in the set, or $2a - b < 0$.

Now we can try the following algorithm: for each a and b in the set ($a < b$), check whether $2a - b$ is negative or is present in the set.

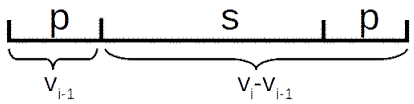
It can be shown that it's enough to check only the pairs a and b that are neighbors in the set. So, sort the set and for each i ($2 \le i \le n$), check if $2v_i - v_{i-1}$ is in the set or it's negative.

The idea above also gives us a way to construct such a string in $O(\ell)$ time. Iterate over the set. Suppose we had prefix v_{i-1} and now we consider prefix v_i:



Look at the picture above. The prefix v_i ends with p. It's a palindrome, thus the string starts with p'. But since the prefix v_{i-1} is also a palindrome, the string ends with p. So, we copy the suffix of length $v_i - v_{i-1}$ to the end of the string.

If $2v_i - v_{i-1}$ is negative, we don't have the suffix of length $v_i - v_{i-1}$:



So, we copy p to the end of the new string and fill s with unused characters. One can easily show that such situation happens no more than $\lceil \log_2 \ell \rceil + 1$ times, so we can use the alphabet of size no more than $\lceil \log_2 \ell \rceil + 1$.

But C_1 is not sufficient. Consider the set $\{1, 2, 4\}$. C_1 is still held, but the required string doesn't exist (constructing the string gives "`aaaa`", which has 3 in its PPS). What are we missing?

Using the constructive method above, we can just build the string and check if the given set and the PPS of the constructed string match. But it's $O(\ell + n)$, which doesn't pass if $\ell = 10^{18}$. So, it's a better idea to search for more conditions.

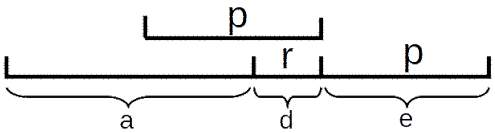
We can notice that while moving from prefix v_{i-1} to prefix v_i, we can skip some palindromic prefixes. It happens, for example, when we copy the string p in the constructive method, but this string is periodic. In this case, if we add only the period, we still get a valid palindromic prefix. So, we need to cut out such cases.

More formally, the following theorem is true:

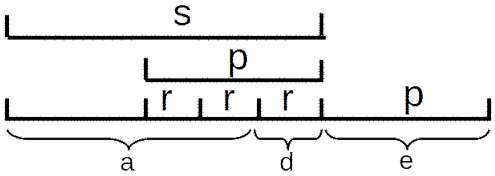
Theorem 1. If we have prefixes a, b, c ($a < b < c$) such that $c - b \ne b - a$, $2b - c \ge 0$ and $c - b$ is divisible by $b - a$, we have a prefix $b + (b - a) = 2b - a$ between b and c.

Remark. The second condition is required, because if $2b - c < 0$, some characters will be filled with the symbols never met before, so we won't get a periodic string.

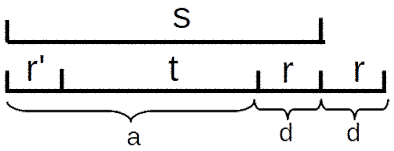
Proof. Denote $d = b - a$, $e = c - b$. Since e is divisible by d there exists such k that $e = kd$. Look at the picture below:



Since prefixes a and b are palindromic, $a - d$, $a - 2d$, $\ldots$, $a - kd$ are also palindromic prefixes. Using the same idea as in the contructive algorithm, it can be shown that p is equal to r repeated k times:



It's left to prove that $s + r$ is a palindrome, so there is a prefix $b + d = b + (b - a) = 2b - a$.



Apparently $s = r' + t + r$ is a palindrome, so t is also a palindrome. Also $r' + t$ is a palindrome, so $r' + t = t + r$. We have

$$s + r = r' + (t + r) + r = r' + (r' + t) + r = r' + (r' + t + r)' = r' + s' = (s + r)'.$$

So, $s + r$ is also a palindrome.

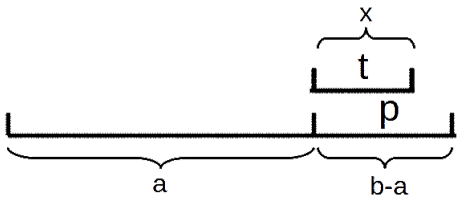
So, we proved that the condition in *Theorem 1* (denote it C_2) is necessary. C_2 can be used to check that we didn't skip a prefix in our set. If we have two prefixes v_i and v_{i-1}, we need to ensure that there's no prefix k such that $v_i - v_{i-1}$ is divisible by $v_{i-1} - k$, $2v_i - v_{i-1} \geq 0$ and $v_i - v_{i-1} \neq v_{i-1} - k$. It will be proved later that $k = v_{i-2}$ is enough to check.

It appears that these conditions are sufficient and there are no other prefixes we missed in the set. Let's prove it.

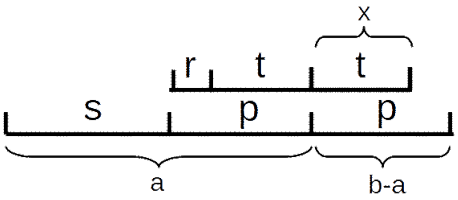
Proof. Proof by contradiction. Suppose we have $a = v_{i-1}$, $b = v_i$, the conditions above (C_1 and C_2) are met, and there's a palindromic prefix $b_1 = a + x$ ($0 < x < b - a$) between a and b. If there multiple such x, we use the minimal x possible.

First, if $2a - b < 0$, then using the constructive algorithm above, we build a string in which $b - 2a$ positions in the middle are filled with new characters, so there cannot be any palindromic prefixes between a and b.

Now consider the case when $2a - b \geq 0$:



Using C_1, we can see that $2a - b$ and $a - x$ are also palindromic prefixes:



As both $s + p + p$ and $s + r$ (prefixes $a - x$ and b respectively) are palindromes, $s + p + p$ ends with r. So, we get $r + t = t + r$, or similarly, the string is equal to its cyclic shift. It means that the string p is periodic. So, we get the contradiction with C_2.

So, C_1 and C_2 are necessary and sufficient. Now we can construct the algorithm in $O(n \cdot \log n)$ time (or even in $O(n)$ time if you use a hash set) to check both conditions. Unlike the analysis, the code is very neat and short :)

```
bool isValidPalindomicPrefixSet(vector<int64_t> v) {
  if (v[0] != 1) return false;
  v.insert(begin(v), 0);
  set<int64_t> s(begin(v), end(v));
  for (size_t i = 2; i < v.size(); ++i) {
    if (v[i] - v[i-1] == v[i-1] - v[i-2]) continue;
```

```
    if (2 * v[i-1] - v[i] < 0) continue;
    if (!s.count(2 * v[i-1] - v[i])) return false;
    if ((v[i] - v[i-1]) % (v[i-1] - v[i-2]) == 0) return false;
  }
  return true;
}
```

Now the only thing we need to prove is that it's enough to take only neighbors while checking C_1 and C_2.

To simplify the reasoning, consider the PPS in terms of *segments*. A *segment* here is the distance between two neighboring prefixes. For example, if the string has palindromic prefixes of length $\{0, 1, 3, 7, 11\}$ (do not forget that we also consider prefix of length zero), then we can construct the segments of length $\{1, 2, 4, 4\}$. A *group of segments* is a set consisting of one or more consecutive segments. The total length of the group of segments is the total length of all the segments in the group. Each segment and group of segments have their starting and ending prefixes, i. e. the numbers each prefix starts and ends with.

Later we will use the length of a segment or a group of segments. Length of the segment s is denoted $|s|$. Length of the group G is denoted $|G|$.

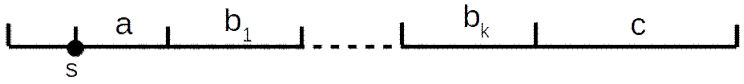
It's easy to notice that the lengths of the segments are non-decreasing, otherwise C_1 isn't held.

We can also explain C_1 and C_2 in terms of segments. C_1 means that if we have a group of segments G starting at prefix p, then we also have a group of segments H ($|H| = |G|$) ending at p, or $p < |G|$. And C_2 means that for each segment s starting at p and each group G ending at p, one of the following conditions is held: $|s| = |G|$, or $|s|$ is not divisible by $|G|$, or $p < |s|$.

We also need to prove the following statement:

Theorem 2. Suppose we have the segment b ending in p and the segment c starting in p, and $|b| < |c|$ applies. Then, $|b| + |c| > p$.

Proof. Consider the neighboring pairs of segments b and c such that $|b| < |c|$ from left to right and check if $|b| + |c| > p$ for all of them. For first such pair we obviously have $|b| = 1$, so $|c| > p$ must hold (otherwise we get a contradiction with C_2 because $|c|$ is always divisible by $|b| = 1$). Otherwise, the situation is as follows:

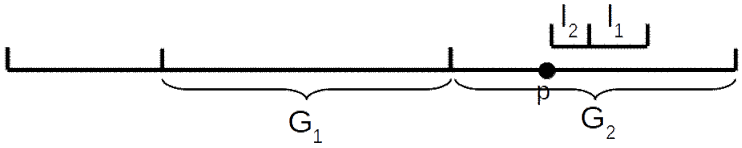


Here, $a, b_1, \ldots, b_k$ and c are segments, and s is some prefix. Also we have $|b_1| = \cdots = |b_k|$ and $b_k = b$. Check if the theorem is true for segment c. By considering all the smaller prefixes, it's already proved that $|a| + |b| > s + |a|$. Now we need to prove that $|b| + |c| > s + |a| + k \cdot |b|$, so the entire theorem will be proved. It's easy to see that $|c| \geq k \cdot |b| + |a|$ (otherwise either C_1 isn't held or $|c|$ is divisible by $|b|$ and C_2 isn't held). So, we have $|b| + |c| \geq |b| + k \cdot |b| + |a| > k \cdot |b| + s + |a|$.

Using the results above, we can prove another theorem:

Theorem 3. If we have two neighboring groups of segments G_1 and G_2 (G_2 is to the right of G_1) such that $|G_1| = |G_2|$, then G_2 consists of segments of equal length.

Proof. Proof by contradiction. Suppose we have such neighboring groups G_1 and G_2 such that G_2 contains at least two segments (l_2 and l_1) of different lengths. It's easy to notice that we can pick l_2 and l_1 in such a way that they are neighbors:



It's safe to assume that $|l_1| > |l_2|$. Also, both l_1 and l_2 are in G_2, and $|G_1| = |G_2|$, so $p \geq |l_2| + |l_1|$. Contradiction with *Theorem 2*.

Finally, we show that taking neighbors while checking C_1 and C_2 is sufficient.

First, prove it for C_1 by contradiction. Then there's a group of segments G starting at p, which violates C_1 and consists of more than one segment. If there are two segments of different lengths, then $|G| > p$ by *Theorem 2*, so C_1 is held. Now suppose G consists of k segments of the same length l. In this case we need to check for the prefix $p - kl$. But we can achieve this only using one segment of length l. So, we don't miss anything.

To show that checking only neighbors is valid for C_2, we use *Theorem 3*. Consider adding a segment s with starting point p. If $|s| > p$, then C_2 is held. Otherwise there is a group G ending in p such that $k \cdot |G| = |s|$ and $k > 1$ (if $k = 1$ or $|s|$ is not divisible by $|G|$, C_2 is held either). But as $k \cdot |G| = |s| \le p$, we can show using C_1 that there is a group H right before G such that $|G| = |H|$. Now *Theorem 3* shows that G consists of segments of the same length l. But in this case checking only the segment of length l ending in p is OK.

Good, we finally dealt with the simpler version. So, let's move on and solve the harder one.

Solving the harder version

In the harder version, we need to add minimum amount of numbers in the set in such a way that it becomes a PPS or some string. When we know the solution for the simpler version, it's easy to construct a naive solution for the harder one:

- If 1 is not in set, then add it.
- Check if C_1 holds, i. e. for each neighboring prefixes a and b ($a < b$) either $2a - b < 0$ must hold or $2a - b$ must be present in the set. If not, add $2a - b$ to the set.
- Check if C_2 holds, i. e. for segment s starting at prefix p that has previous segment t, if $|s|$ is divisible by $|t|$ and $|s| \le p$, then the segment s must be split with a new prefix into smaller segments that have length $|t|$.

Unfortunately, such solution works in $\Theta(\ell)$ time in the worst case (i. e. not better than in linear time), so we need something better.

First, forget about the prefixes itself. Now we consider only segments. And, what's more, we compress the information about the segments, keeping pairs of the form (l, c) instead of original segments, where l is the length of one segment, and c is the amount of segments of such length coming sequentially. For example, the set of prefixes $\{0, 1, 3, 7, 11, 15\}$ will be transformed to the sequence of segments $\{1, 2, 4, 4, 4\}$ and then transformed to the sequence of pairs $\{(1, 1), (2, 1), (4, 3)\}$.

Call the segment s starting at prefix p *large* if $|s| > p$. (By the way, p is the sum of previous segments' lengths.)

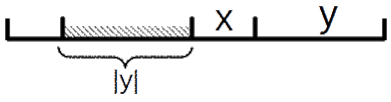
Now, let's take a different look at C_1 and C_2. C_1 says that a segment s must be either large, or $|s|$ must be equal to the sum of lengths of sequential segments coming right before s. Similarly, C_2 says that a segment s with previous segment t must be either large, or $|s| = |t|$ holds, or $|s|$ is not divisible by $|t|$.

Recall that the segments' lengths are non-decreasing, otherwise a C_1 doesn't hold. Another useful observation comes from *Theorem 3*: if the segment s starting from prefix p is longer than the previous one, t, then $s \ge \frac{p}{2}$. So, adding a new segment with different length increases the total length at least by half, which means that the number of pairs in compressed representation is $O(\log \ell)$, which is much smaller than the original sequence of segments.

Before going further, we prove the following theorem:

Theorem 4. Consider a segment s, which is not large. This segment is split into two smaller segments, x and y ($|x| + |y| = |s|$). If the split these segments to the segments of equal length $g = \gcd(|x|, |y|)$, the answer to the problem won't change.

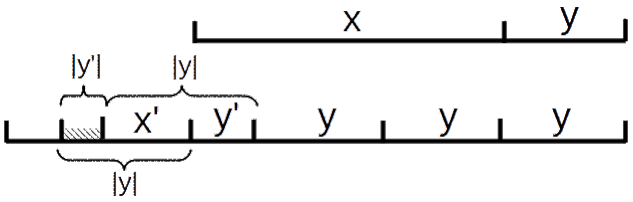
Proof. If s is not large, then there exists a group of segments G such that $|G| = |s|$. For the sake of clarity, we merge the group G into one segment and use the last $|y|$ ($|y| \le |G|$) positions of it as a scratch space:



Now we need to prove the possibility of such replacement if we have $|y|$ positions of scratch space before x. We do it by induction on $|x|$ and $|y|$.

If $|y|$ is divisible by $|x|$, then we use C_2 to split $|y|$ to the segments of size $|x| = g$. If $|x|$ is divisible by $|y|$, then then we apply C_1 to segments of length y sequentially, splitting x by segments of size $|y| = g$. In both cases, the theorem is proved.

Now consider the case where y is not divisible by x. Then, we apply C_1 to segments of length y sequentially, cutting the segments of size $|y|$ from x. The length of the remaining segment y' is equal to x modulo y. Then, we apply C_1 once more, getting the segment x' (see the figure below):



We used $|x'|$ positions of scratch space, so $y - |x'| = y - (y - |y'|) = |y'|$ positions remain. So we perform an induction step and split x' and y' into segments of size $g' = \gcd(|x'|, |y'|)$. By Euclid's theorem, $g' = g$. Then, as $|y|$ is divisible by $|g'|$, we apply C_2 to all the upcoming segments of length $|y|$ and split them into the segments of length $|g'|$. The theorem is proved.

What basically happened in the proof is that we used Euclid's algorithm to split the segments. Neat.

Now, we come up with an optimal algorithm. We maintain the sequence of pairs (l, c) for all the added prefixes. We add the prefixes from the input one by one, in sorted order. Do not forget to add 1 to the input set beforehands if it's not already present.

When we add a new prefix, we push a new segment s and try to apply C_1 and C_2 until we get a valid PPS. So, we have $O(\log \ell)$ pairs after each addition.

To perform addition fast, we do it in the following way:

1. If s is a large segment, we can just add it.
2. Suppose t is a segment previous to s. If $|s|$ is divisible by $|t|$, then we split s into the segments of size $|t|$, add them and finish the addition.
3. Let $|t| > |s|$. Then, we try to apply C_1 to s once. If no new segments are added (i. e. $|s|$ is equals to the size of some neighboring group G), then we just add it and return. Otherwise, some segment is split. We termorarily pop all the segments after the split, applying them later. Now consider the segment being split:
   - If this segment is large, we just try to add the two subsegments recursively.
   - Otherwise, we use the result from *Theorem 4* and split those two segments x and y into smaller segments of size $g = \gcd(|x|, |y|)$.

   When returning the popped segments back, we use the same algorithm recursively. But it won't take long, as the algorithm either just adds a segment, or splits it to equal subsegments on step *2)*, so it's $O(1)$ for each readded segment.
4. Otherwise, $|t| < |s|$. We apply C_1 many times as in the proof of *Theorem 4*, getting many segments of length s and one segment t' of length t modulo s. Now, we have two cases:
   - If t was a large segment, we add the segment t' and one of the segments of length $|s|$ recursively. Then, we add the rest of the segments of length $|s|$ in $O(1)$ time, as they are either added untouched, or all of them pass through step *2)* of the algorithm and are split on the subsegments of the equal size.
   - Otherwise, we use the result from *Theorem 4* and split all the segments into the subsegments of size $g = \gcd(|s|, |t|)$.

It's easy to see that this algorithm works. All we have to do now is to estimate its time complexity. Consider the cases with two recursive calls. They only happen if we split a large segment. If at least one of the subsegments is large, then the corresponding recursive calls stop at step *1)*, so such call is basically $O(1)$. The recursive call which is not stopped proceeds with a smaller number of pairs, so these cases have $O(\log \ell)$ time complexity.

Otherwise, we split a large segment into two non-large segments. Such event destroys a large segment completely. It's not hard to observe that there are at most $O(\log \ell)$ large segments created during all the additions, since each large segment doubles the total length of the segments. Even if we obtain a large segment by splitting two large segments, the original segment had quadrupled the total segment length when it was added. So, there will be no more than $O(\log \ell)$ such splits, and $O(\log \ell)$ extra recursive branches, respectively.

The considerations above give time complexity $O(\log^2 \ell + n \log \ell)$. Though, I don't know how to generate a test for which $O(\log^2 \ell)$ part runs much slower than $O(n \log \ell)$ part.

So, finally, after all those pages of theorems and explanations, we obtain a fast and beautiful soltuion for this problem :)