FINDING A DIVISIBLE PAIR AND A GOOD WOODEN FENCE

Stelian Ciurea
Universitatea Lucian Blaga, Sibiu, Romania
stelian.ciurea@ulbsibiu.ro

Erik D. Demaine
MIT, Computer Science and Artificial Intelligence Laboratory
edemaine@mit.edu

Corina E. Pătraşcu
Harvard University, Department of Mathematics
patrascu@fas.harvard.edu

Mihai Pătraşcu
MIT, Computer Science and Artificial Intelligence Laboratory
mip@mit.edu

Abstract

We consider two algorithmic problems arising in the lives of Yogi Bear and Ranger Smith. The first problem is the natural algorithmic version of a classic mathematical result: any $(n+1)$-subset of $\{1, \ldots, 2n\}$ contains a pair of divisible numbers. How do we actually find such a pair? If the subset is given in the form of a bit vector, we give a RAM algorithm with an optimal running time of $O(n/\lg n)$. If the subset is accessible only through a membership oracle, we show a lower bound of $\frac{4}{3}n - O(1)$ and an almost matching upper bound of $\left(\frac{4}{3} + \frac{1}{24}\right)n + O(1)$ on the number of queries necessary in the worst case.

The second problem we study is a geometric optimization problem where the objective amusingly influences the constraints. Suppose you want to surround n trees at given coordinates by a wooden fence. However, you have no external wood supply, and must obtain wood by chopping down some of the trees. The goal is to cut down a minimum number of trees that can be built into a fence that surrounds the remaining trees. We obtain efficient polynomial-time algorithms for this problem.

We also describe an unusual data-structural view of the Nim game, leading to an intriguing open problem.

1. *Introduction*

Yogi Bear is on his constant search for picnic baskets in Jellystone National Park, while his sidekick Boo Boo Bear tries to distract Yogi by teaching him some mathematics.

"You see, Yogi," continued Boo Boo, "one number *divides* another number if you can multiply the first number to make the second number."

"Yogi is smarter than the average bear," responded Yogi, "but he never learned how to multiply."

"OK, forget multiplication. Think about division as splitting a number into a smaller number of groups of equal size. For example, we split the 12 months of the year into 4 seasons of 3 months each. So 4 divides 12, because you can split 12 into 4 groups of 3."

"Wait a minute, Boo Boo. I smell something. . . "

The bears came across a large camping ground near the lake. The campers were out for a quick swim in the lake before their lunch picnic. Yogi, being smarter than the average bear, seized the opportunity to steal several picnic baskets and take them back to his cave.

"There are so many different kinds of food," said Yogi, "that I don't know which to eat first."

"Then why don't you eat two kinds of food first?" suggested Boo Boo.

"But look at this basket, Boo Boo. There are 2 sandwiches, 3 apples, 5 grapes, and 6 strawberries. You've taught me enough about numbers to know that none of these numbers are the same. If I ate a grape with each apple, I'd have two grapes left over. That's not a balanced diet."

"But Yogi, here is where you can use division. Suppose there were 6 grapes. You can break 6 grapes into 3 groups of 2—3 divides 6—so you can eat 2 grapes for every apple and everything will balance."

"You are too abstract, Boo Boo. I see only 5 grapes."

"That's true, but even in this basket there are divisible numbers. Can you find them?"

Yogi stared at the food for a few minutes, while his stomach grumbled. Then his eyes lit up, and he put 3 strawberries on each of the 2 sandwiches, then ate both the combinations.

"Very good, Yogi!" said Boo Boo. "Now, what about this basket? There are 3 peaches, 4 pears, 5 nectarines, 7 oranges, and 9 peanuts."

Yogi's brow started to sweat as the problems got bigger. Finally he paired up 3 peanuts with every peach and ate his creations.

"This is getting hard, Boo Boo," complained Yogi. "Is there some easy way to find divisible numbers?"

Boo Boo got out some chalk and started writing on the cave wall. He noticed that the baskets encountered so far always have the property that the size of the largest group is at least two less than twice the number of groups. In other words, the group sizes form a subset of $\{1, 2, \ldots, 2n\}$ of size at least $n + 1$. Luckily, he observed, such sets always contain a divisible pair of numbers. But can he develop fast algorithms to find such pairs? In Section 3 we present several algorithms for this problem, depending on the model of how the numbers are specified.

After Yogi had a balanced diet of the foods he could pair together, he and Boo Boo went back to the campsite to return the rest of the food. To Yogi's great surprise, they found a group of angry campers talking to Ranger Smith. When Ranger saw them, he got angry too.

"Yogi, you have to stop stealing picnic baskets!"

"But Ranger," Yogi pleaded, "I brought back what I didn't eat."

"It doesn't matter, Yogi. This stealing must end. I'm putting up a fence around the whole forest to keep bears *out*."

Yogi was devastated. He looked at Boo Boo worryingly. What were they to do? Suddenly Boo Boo had an idea.

"Ranger," Boo Boo asked, "where are you going to get the wood for your fence?" Ranger paused for a minute.

"I guess I'll get the wood by cutting down a few trees from the forest."

"Ah ha," said Boo Boo. "But you don't want to cut down too many trees in our beautiful Jellystone."

"Of course not," answered Ranger. "If I cut down trees along the perimeter, I'll have a smaller region to fence off."

"Ah ha," said Boo Boo. "But the trees in the middle of the forest are larger and offer more wood. So wouldn't it be more beneficial to cut them?"

"I certainly want to cut the minimum number of trees," said Ranger. He scratched his chin in thought. "There seems to be a trade-off between cutting fat trees and cutting trees that reduce the perimeter…"

Ranger is left with a challenging optimization problem, which we solve in Section 2 by a polynomial-time algorithm.

2. *How to Build a Wooden Fence*

The problem Ranger Smith faces is the following. Consider n trees at given coordinates in the plane. Find a minimum set of trees to cut down such that the remaining trees can be surrounded with a fence made out of the resulting wood. An important parameter besides n is the number of trees that need to be cut in the optimal solution; call this k. It is reasonable to assume that k is in general much smaller than n. For example, $k = \Theta(\sqrt{n})$ if all trees contribute a unit of wood and the trees are roughly uniformly spaced at roughly unit distances. Therefore, to help Ranger Smith finish his plans in time, we strive to obtain bounds in terms of k, rather than n.

We assume that each tree i has a given parameter W_i specifying the amount of wood that can be obtained by cutting down this tree. This quantity should be properly scaled, so that the condition of the problem is that the perimeter of the convex hull surrounding the remaining trees must be at most the total wood amount of the trees that have been cut down. The algorithm we has a running time of $O(n^2k + nk^5)$; if $k = \Theta(\sqrt{n})$, this translates into an $O(n^{3.5})$ running time. In the special case when all trees contribute the same amount of wood, we can achieve a running time of $O(n^2 + n \cdot k^4)$.

One might also consider the case when trees have different weights (for instance, based on the botanical importance of the tree) and we want to cut down a set of minimum weight. However, this problem is NP-hard by an easy reduction from SUBSET-SUM [CLRS01]. Assume we want to find a subset of $\{a_1, \ldots, a_n\}$ with minimum sum greater than 1. Place three trees as the vertices of a triangle of perimeter 1; each of the three trees has a very large weight. For every a_i, place a tree somewhere inside the triangle, with both the weight and the amount of wood equal to a_i. Then, a subset of the a_i trees must be cut, with the sum of the wood at least 1, but minimizing the sum of the weights (which is equal to the amount of wood). Fortunately, the democratic nature of Jellystone Park, where every tree is born equal, makes the weighted problem irrelevant.

Our algorithm first preprocesses the trees into a data structure that supports the following type of queries. Given three trees i, j, k, the data structure reports the number and total wood amount of the trees contained in each of the four angles determined by the two lines i, j and i, k. (For simplicity, we assume that trees are in general position.) We show how to support this query in $O(1)$ time using $O(n^2)$ preprocessing time and space. For every tree i, we sort all other trees angularly around i. We store the index of every tree $j \neq i$ in this order, as well as where the reflection of j through i would fit in the order. We also store the sum of the wood amounts W_x over all trees x that come before j in the order. Using this data we can calculate the answers to a query in constant time by subtracting two indices and two partial sums. This data structure can be constructed in $O(n^2)$ time by dualizing points to lines and computing the line arrangement.

In the remainder of this section, we give an algorithm with running time $O(n \cdot K_{\max}^5)$, which either finds an optimal solution that involves cutting down at most $K_{\max}$ trees, or reports a failure. By calling this algorithm repeatedly until success, doubling $K_{\max}$ each time, we obtain an algorithm for general k whose running time is $O(n \cdot k^5)$ from a geometric series. For simplicity, we write k instead of $K_{\max}$ in the rest of the section.

For the initial and final portions of our algorithm, we require a tree on the convex hull of the uncut trees in an optimal solution. The simplest way to find such a tree is to try each of the lowest $k + 1$ trees in terms of y coordinate. These trees are precisely the trees that can be the lowest tree of the convex hull, by cutting all trees with smaller y coordinate. For the rest of the algorithm, L denotes the tree currently chosen as the lowest tree of the convex hull.

Our algorithm is based on a dynamic program, guessing the convex hull of the uncut trees one vertex at a time in counterclockwise order. Each subproblem considers guessing a subchain of the convex hull that starts at the lowest tree L and ends with some edge. By the final subproblem, we consider subchains that return to L, forming an entire convex polygon. All trees outside this convex polygon must be cut. If trees contribute an equal amount of wood, there is an optimal solution that cuts no trees from inside the convex hull: we can always cut trees on the hull instead, and only reduce the perimeter of the new hull, maintaining the same amount of available wood.

If trees contribute a different amount of wood, an optimal solution may involve cutting trees from inside the convex hull. In such a case, if a tree contributing wood in the amount W_j is cut, in an optimal solution all trees contributing amounts $W_i > W_j$ are also cut. Indeed, we can always replace cutting a tree giving less wood with cutting a tree giving more wood; the number of trees that are cut remains the same, the amount of wood available increases, and the perimeter of the convex hull doesn't increase (because the initial tree was inside the hull). Thus, an optimal solution involves cutting the c_0 trees giving the most wood, for some $0 \leq c_0 \leq k$, and then all trees outside some optimal convex hull. To allow for this possibility, we will iterate over all c_0, cut the first c_0 trees in the order of the wood provided, and then run the entire algorithm as described. This includes regenerating the data structure in the absence of the c_0 trees. Tie breaking for trees providing the same amount of wood can be handled using common perturbation techniques.

We now return to the dynamic program guessing the convex hull. This program evolves in three dimensions $[i, j, c]$. The first two dimensions i and j specify the last segment i, j of the guessed subchain. The third dimension c is the number of trees that we are forced to cut based on the subchain constructed so far. These trees are precisely those to the right of at least one segment of the subchain. Each cell from the dynamic program holds the maximum amount of wood that can be left after the construction of a subchain satisfying the restriction $[i, j, c]$. The wood left after a construction is the total wood obtained from the c cut trees, minus the perimeter of the subchain.

Using a greedy exchange argument, we show that subproblems are characterized completely by the three coordinates. Assume we have two possible constructions for a subchain of the hull starting at tree L, ending in the segment i, j, and leaving c trees to the right of some segment. Then, given a convex hull that contains one of the subchain constructions, we can replace it with the other subchain construction. The result will still be a convex hull: the subchain $L, \ldots, i, j$ is still convex, and so is the subchain $j, \ldots, L$. In addition, the tree following j must be to the left of the edge i, j, so the two convex subchains fit together to form a convex polygon. Also, the number of trees cut down by both solutions is the same: any tree that is cut down is either below L (which is fixed), to the right of the subchain $L, \ldots, i, j$ (there are exactly c such segments in both constructions), or to the right only of some segments from $j, \ldots, L$ (and this subchain has not changed). Given that we can interchange two constructions with the same description $[i, j, c]$, we are interested only in the one leaving the most possible wood after constructing the subchain $L, \ldots, i, j$. Certainly, if we have a solution involving a suboptimal subchain $L, \ldots, i, j$, we can always exchange it for the corresponding subchain of an optimal solution, and have more spare wood at the end.

We initialize the dynamic program by filling in the cells $[L, i, c]$, for every i, where c is the number of trees below L or to the right of the edge L, i. All other cells $[L, i, c']$ are initialized with the value $-\infty$. To solve the overall problem, we consider all cells of the form $[i, L, c]$, which corresponds to a completed convex hull $L, \ldots, i, L$. Among these cells, we choose the cell with smallest c such that the wood left after the subchain's construction is non-negative. At some intermediate points in the construction, it is valid for the wood left to be negative, but this deficit must eventually be covered by trees that are cut later.

It is conceivable that partial constructions of the convex hull might actually intersect themselves: local convexity of each vertex does not imply global convexity in the case of nonzero winding. To avoid this problem, we only consider cells $[i, j, c]$ for which L is to the left of the segment i, j. For every possible pair $\{i, j\}$, exactly one of i, j or j, i will be valid according to this condition. This condition also suggests a good order in which the dynamic program should be filled out. The convex hull can be broken into two parts, one part starting from L and going monotonically upwards, and the other part going monotonically downwards and returning to L. To compute all interesting ways to build the first part, we consider all valid cells $[i, j, c]$, where j is above i, in increasing order of j's y coordinate. Any tree that could appear before i must have a y coordinate less than i, and hence less than j, so all data necessary to compute $[i, j, c]$ is available. After this upward scan, we perform a downward scan to find all possible downward continuations of previously computed portions going upward. In this scan, we consider all valid cells $[i, j, c]$, where i is above j, sorted in decreasing order of j's y coordinate. This ensures once more that all data to compute a cell is available at the needed time.

The last detail regarding our algorithm is how exactly the value of cell $[i, j, c]$ is calculated. To do that, we consider all possible trees p that could immediately come before i in the convex hull. The angle p, i, j must turn to the left to ensure convexity. In addition to the trees cut down by the subchain $L, \ldots, p, i$, we also cut down trees that are both left of the edge p, i and right of the edge i, j. These are the only new trees that need to be cut for the longer subchain $L, \ldots, p, i, j$. To determine the number and weight of these trees, we use the data structure described in the beginning. Thus, filling out the cell $[i, j, c]$ involves iterating over all possible trees p, and performing $O(1)$ computations for each.

It remains to analyze the running time of our algorithm. We have an $O(k)$ factor from the search for c_0 — the number of trees giving the most amount of wood that are cut initially. Another $O(k)$ factor comes from the initial search for L. A first estimate for the size of the dynamic table is $O(n^2k)$ because $i, j \in \{1, \ldots, n\}$ and $c \leq k$. (We need not consider partial solution cutting down more than k trees, though they may obviously exist.) However, this bound can be reduced to $O(n \cdot k^2)$ by arguing that there are at most $O(k)$ feasible choices of i for every j. Specifically, for every tree j, we claim that there are at most $O(k)$ trees i such that the segment i, j has at most k trees on at least one side. Clearly, if there are more than k trees on both sides, we cannot include this segment on the convex hull, because we must cut all trees on one of the two sides. Furthermore, we claim that these $O(k)$ trees can be precomputed while building our data structure in the beginning, at no additional cost. Thus, the dynamic program can consider only the choices for i in j's precomputed list of feasible neighbors.

To see the $O(k)$ bound, consider some trees j and p. Let $\overline{p}$ be the reflection of p through j. Consider sorting all trees except j angularly around j. Every tree i that is more than k positions in the sorted order away from both p and $\overline{p}$ cannot be a feasible neighbor for j, because then the segment i, j has at least k trees on p's side and at least k trees on $\overline{p}$'s side. Thus there are at most $4k$ feasible neighbors for j.

The last detail of the analysis is how long it takes to compute the value in each cell. This computation involves iterating over all possible predecessors p of i on the convex hull. Again, there are at most $O(k)$ such trees p because the edge i, p must be on the convex hull. Thus, the total running time is $O(n \cdot k^5)$ for the dynamic program, plus $O(n^2k)$ for construction the data-structure (this needs to be done for every c_0).

FACT 1 *An optimal wooden fence can be found in time $O(n^2k + n \cdot k^5)$. If all trees contribute an equal amount of wood, the running time is $O(n^2 + n \cdot k^4)$.*

3. *Finding a Divisible Pair*

The task Yogi and Boo Boo face is to find a pair of divisible numbers in an $(n + 1)$-subset of $\{1, \ldots, 2n\}$. The existence of such a pair is a well-known fact belonging to the mathematical folklore. We include below the proof which is usually given, because it forms the basis of a (suboptimal) algorithm. The quantities arising from our analysis will often be of the form $\frac{c_1}{c_2}n + O(1)$. The additive constants are usually small – most of them are at most 2. We do not bother to specify these constants exactly, since they generally depend on $n \bmod c_2$. For the proofs, we generally assume that n is a multiple of c_2. In all cases, it is straightforward, but somewhat tedious, to show that the bound for other values of n differs by only an additive constant.

A natural formalization of this problem is in an oracle model: the $(n+1)$-subset is not known to the algorithm, and is only accessible through membership queries asked to the oracle. The main results of this section will be a lower bound of $\frac{4}{3}n - O(1)$ on the number of queries necessary in the worst case, and an almost matching upper bound of $\left(\frac{4}{3} + \frac{1}{24}\right) n + O(1)$. We believe the upper bound can be improved to match the lower bound, though we were unable to do that.

As computer scientists, we cannot help digressing to another interesting model of computation: the word RAM [FW93], with memory cells containing $\Theta(\lg n)$ bits. In this case, a natural assumption is that the input is directly accessible to the algorithm, in the form of a bit vector $A[1\,..\,2n]$, holding the membership information. In this context, the optimal complexity turns out to be $\Theta(n/\lg n)$, i.e. linear in the input size. The algorithm uses only AC^0 operations, and avoids the multiplication operation, which is not supported efficiently in Yogi's neuronal CPU.

3.1. *Warming up*

In this section, we give a proof of the existence of a divisible pair, and turn this proof into an asymptotically optimal algorithm for the RAM. The next section will discuss some consequences of this result in the oracle model.

FACT 2 *For any $S \subset \{1, \ldots, 2n\}, |S| = n + 1$, there exist $a, b \in S$ such that a divides b.*

Proof: For every odd number $q \in \{1, 3, 5, \ldots, 2n-1\}$, let $B_q = \{q \cdot 2^i \mid 0 \leq i \leq \log_2 \lfloor n/q \rfloor\}$ – that is, a bin with all power-of-two multiples of q. Since every number is in the bin generated by its largest odd divisor, we have $B_q \cap B_r = \emptyset$ and $\bigcup B_q = \{1, \ldots, 2n\}$. There are exactly n such bins, so at least one bin must contain two elements $a, b \in S$. By construction of the bins, a and b are a divisible pair. $\square$

This proof is quite strong: it not only tells us that a divisible pair exists, but that one exists in which one number is a power-of-two multiple of the other! This has a few important and simple consequences, like very efficient parallel algorithms or a data structure with constant update time for the dynamic problem. For our purposes, it suffices to note that it immediately gives a linear time algorithm. The algorithm simply tests membership of all $i = 1\,..\,n$ in increasing order, and for each $i \in S$, tests membership of all multiples of the form $2^j i$. The only interesting detail in analyzing the running time is noting that every number is examined at most once as a power-of-two multiple during the entire execution. Assume for contradiction that t is examined twice. Then two values x and y ($x < y$) exist such that $x, y \in S$ and $t = x \cdot 2^a = y \cdot 2^b$.

But then $y = x \cdot 2^{a-b}$, and when $i = x$, $j = a - b$, the procedure must have stopped and returned the pair (x, y).

Now we improve the running time of this algorithm to $O(n/\lg n)$ using the word-level parallelism of the RAM. The difficulty with the previous algorithm is that it does not exhibit locality in its bit accesses, which is necessary to make efficient use of the ability to read $O(\lg n)$ bits in parallel. Fortunately, a simple reformulation of the search strategy solves the problem. For each p in the set, we simply examine $2p$. If it is in the set, we are done; otherwise, we artificially add $2p$ to the set, so that its multiples (which are also multiples of p) will be considered later. Of course, when we find a pair $(p, 2p)$, we cannot be sure that p was originally in the input set. However, a number of the form $p/2^i$ must have been in the original set, and such a number can be found efficiently.

To improve this algorithm, we assume the set is given as a bit vector $B[1\,..\,n]$ in packed form, with at least $\lg n$ bits per word. It is well known that we can read or write any block of up to $\lg n$ bits using $O(1)$ operations. This requires accessing at most two adjacent words of memory (because the bits may be split between two consecutive words), and extracting the relevant information using shifting and masking. Our strategy will be to execute $(\lg n)/2$ iterations of our main loop in parallel. We begin by reading the $(\lg n)/2$ bits corresponding to a range of values in B. We then space out these bits, by inserting a zero between every two bits. This is needed because consecutive bits become spaced when we examine $B[2 \cdot i]$. We can support this operation by lookup into a table of size $2^{(\lg n)/2} = O(\sqrt{n})$ words. Having this suitable representation, we can test whether $B[2 \cdot i]$ is set in parallel, for all i's in the range, using a bitwise and. Similarly, artificially adding $2 \cdot i$ to the set can be simulated with a bitwise or. Thus, $O(\lg n)$ iterations of the main loop can be simulated with $O(1)$ operations, unless a divisible pair is found. When this happens, we examine all $O(\lg n)$ bits individually and determine which one triggered a stop. The final portion of the execution needs not be optimized, since it only takes $O(\lg n)$ time.

A final word about the RAM model: this algorithm is asymptotically optimal, since any algorithm must read a constant fraction of the input. To see that, consider the input $\{x, n+1, \ldots, 2n\}$, where x is chosen randomly from $\{1, \ldots, n\}$. Since x is part of any divisible pair, an algorithm must probe the cell containing $A[x]$ to learn the value of x. Since x is chosen uniformly at random, the correct cell is randomly distributed among $n/\Theta(\lg n)$ cells. So any deterministic algorithm must make $\Omega(n/\lg n)$ probes on average; by duality the the same bound holds for randomized algorithms.

FACT 3 *The complexity of finding a divisible pair on the RAM is* $\Theta(n/\lg n)$.

3.2. *A First Take at the Oracle Model*

We now lay the ground for the following sections by considering the oracle model. Here, we are not satisfied with an asymptotic analysis, which easily reveals a $\Theta(n)$ complexity, but aim for a more precise understanding of the number of queries needed in the worst case. A more careful analysis of our first algorithm reveals that it makes $3n/2 + O(1)$ queries in the worst case, if we cache the oracle answers and never ask the same question twice. This is because odd numbers greater than n are never considered. Indeed, these numbers have no multiple below $2n$, so they cannot be part of a pair where the two numbers differ by a power of two factor. Since the algorithm always finds such a pair, it has to stop before reaching an odd number greater than n.

In fact, the $3n/2$ query complexity is not particular to this implementation of the ideas from the proof of fact 2. The hint given by this proof is deceiving: any algorithm that always finds a pair of the form $(s, 2^i s)$ must perform at least $3n/2 - O(1)$ queries in the worst case. To see that, consider again the bins B_s from the proof of fact 2. The algorithm must identify two elements in the input subset that are in the same bin B_s. Now consider an adversary with the following strategy: the first element probed from each bin is declared to be *in* the set, and any subsequent elements from the same bin are declared to be *out*. If the adversary has already declared $n-1$ elements to be out, it is forced to answer any remaining queries in the affirmative.

To prove the effectiveness of this adversary, we begin by noting that it never generates a pair $(s, 2^i s)$ in the set, except when forced to do so because it has already declared $n-1$ numbers to be out of the set. So the algorithm must make $n-1$ queries which receive a negative answer to fix the divisible pair. In addition, note that a negative answer is never generated the first time a bin is touched. To accommodate $n-1$ negative answers, all but one of the bins generated by $q \in \{1, 3, \ldots, n-1\}$ must be touched. So, the algorithm must also make $n/2 - 1$ queries receiving positive answers before the pair is fixed.

3.3. *A Good Lower Bound*

In this section, we prove a lower bound of $4n/3 - O(1)$ on the number of queries necessary in the worst case. The adversary from the previous section fails in the general case, because it has no control over the divisibility relations except for numbers differing by a power-of-two factor, and could easily introduce a divisible pair at any step. The adversary from this section applies a similar bucketing technique, but only for pairs $(x, 2x)$ with $x \in \{2n/3 + 1, \ldots, n\}$. Since numbers between $2n/3 + 1$ and n have just one multiple, the adversary can more easily control the effects of its decisions.

The strategy of the adversary is simple. The first query to every pair $(x, 2x)$, with $x \in \{2n/3+1, \ldots, n\}$, receives a positive answer. As long as the adversary has a choice, i.e. it has not already declared $n-1$ numbers to be out of the set, the second query made to a pair receives a negative answer. Numbers greater than n which are not part of such a pair are always included in the set, so an algorithm which knows the adversary need not ask about these. Number below $2n/3$ are *in principle* not included in the set. However, the very last one probed might be included in the set, if the adversary has already declared $n-1$ values to be out of the set.

As before, the lower bound comes from analyzing the number of queries needed to fix the divisible pair. Once the pair is fixed, an optimal algorithm need not actually query the two numbers in the pair. First note that our adversary never generates a divisible pair unless it has already declared $n-1$ numbers to be out of the set. Indeed, before this inevitable moment, no number smaller than $2n/3$ is included in the set (so numbers above n which are always included have no divisors in the set), and only one number from each special pair is declared to be in.

Therefore, a divisible pair is not fixed, unless $n-1$ queries have already received a negative answer. However, at most $2n/3$ of these can come from positions $1, \ldots, 2n/3$. Therefore, at least $n/3-1$ must come from one of the special pairs. Now remember that the first query touching a special pair always receives a positive answer. So in addition to the $n-1$ negative answers, the adversary must also have given $n/3-1$ positive answers.

FACT 4 *Finding a divisible pair requires at least* $\frac{4}{3}n - O(1)$ *queries in the worst case.*

3.4. *A Good Upper Bound*

As shown in section 3.2, we can only hope to beat the $3n/2$ barrier by abandoning the strategy of searching only for power-of-two multiples. However, it might seem that a strategy that probes all multiples of an element can do even worse than $3n/2$, since it foregoes even the basic guarantee of not touching odd numbers greater than n. Surprisingly, a naive algorithm which probes all i's in increasing order, and for every i in the set probes all its multiples, has relatively good performance. We can prove that this algorithm does at most $\left(\frac{4}{3}+\frac{1}{12}\right)n + O(1)$ queries, and we can exhibit a family of inputs on which this bound is tight. All the ideas necessary to prove this will also appear in the proofs of this section.

In this section, we analyze a simple improvement to this strategy. The algorithm considers all numbers i in increasing order. Usually, i is probed, and if it is in the set, its multiples are also probed. However, if $i > 3n/2$ and its single multiple $2i$ has previously been probed and received a negative answer, there is no point in querying i.

We will show that the performance of this improved algorithm is $\left(\frac{4}{3}+\frac{1}{24}\right)n+O(1)$, which almost matches our lower bound.

For the purpose of the analysis, we break the execution of this algorithm into two stages. The first stage lasts as long as $i \leq 2n/3$. We begin with the case when the algorithm finishes during the first stage, i.e. with $i \leq 2n/3$. In this case, we show the algorithm does at most $4n/3+O(1)$ queries. The algorithm can ask at most $n-1$ queries which receive a negative answer, so we must prove that at most $n/3+O(1)$ queries can receive a positive answer. At most one query for a number greater than $2n/3$ can receive a positive answer, because such a number is only probed when one of its divisors is in the set, so we stop after the first positive answer. On the other hand, at most $n/3+1$ queries from the range $\{1,\ldots,2n/3\}$ can receive a positive answer. Indeed, by fact 2 a $(n/3+1)$-subset of $\{1,\ldots,2n/3\}$ contains a divisible pair, so if we have received $n/3+1$ positive answers, we have also found a divisible pair.

Now assume that the algorithm finishes only in the second stage. In this case, all numbers between 1 and $2n/3$ are probed. Let T be the set of such numbers which received a positive answer, and let N be the set of probed numbers which received a negative answer during the first stage. Given this, exactly $n-1-|N|$ numbers are outside the set and not yet discovered. For every $i>2n/3$, we only probe i if $2i$ has not been probed already. Thus, either i and $2i$ are both in the set and the algorithm stops, or we discover one new number that is outside the set (and possibly also one that is in the set). Thus, the number of queries done in the second stage is at most $2+2(n-1-|N|)=2n-2|N|$. The total number of queries must then be at most $|T|+|N|+2n-2|N|=2n-(|N|-|T|)$. We will show below that $|N| \geq \left(\frac{2}{3}-\frac{1}{24}\right)n+|T|$, which immediately implies our desired bound.

To prove our bound on $|N|$, first note that N contains exactly $2n/3-|T|$ numbers from $\{1,\ldots,2n/3\}$. Let $M=N \cap \{2n/3+1,\ldots,2n\}$. Our bound is equivalent to $|M| \geq 2|T|-\frac{n}{24}$. First note that M contains all multiples greater than $2n/3$ of numbers from T, and in particular all power-of-two multiples. There are 2 power-of-two multiples for every number in $T \cap \{1,\ldots,n/2\}$ and one for the rest of T. However, for every number in T between $n/2+1$ and $2n/3$, M must also contain its triple. Thus, we have identified two multiples belonging to M for every number from T. No two numbers from T can have a power-of-two multiple in common, since they would be divisible, and the algorithm would stop during the first stage. Thus, the only double counting can come from triples, namely when $3x=2^iy$ with some $y \in T, x \in T \cap \{n/2+1,\ldots,2n/3\}$. Clearly, $3x \in \{3n/2+3,\ldots,2n\}$, so $2^{i-1}y \in \{3n/4+1,\ldots,n\}$. In addition, 2^iy must be a multiple of three, so $2^{i-1}y$ must also be a multiple of three. Finally, $2^{i-1}y$ must be even, because $y \leq 2n/3$, so $i \geq 2$. Thus, $2^{i-1}y$ must be a multiple of 6 in the range $\{3n/4+1,\ldots,n\}$. Since there are only $n/24$ such possibilities, and each one defines at most one double-counted multiple, we obtain $|M| \geq 2|T| - n/24$, which completes our proof.

FACT 5 *In the worst case, $\left(\frac{4}{3} + \frac{1}{24}\right) n + O(1)$ queries are sufficient to find a divisible pair.*

4. *Conclusions*

The first problem we considered is an interesting and unexpected variation of the classic convex hull problem. What is maybe surprising at first glance is that a polynomial-time solution exists. A similar approach proves effective for several natural problems, such as fitting a maximum number of points in a polygon of given area or perimeter. We believe such problems have potential applications to statistical geometry.

The second problem is an unusual algorithmic challenge associated with a number-theoretic problem. This contrasts with most work from algorithmic number theory by being a game and an online problem. This allows us to use algorithmic and complexity-theoretic approaches, and obtain both upper and lower bounds. This should be compared with the current state of facts in algorithmic number theory, where known techniques are too weak to give any interesting lower bound. However, our work is also quite far from vanilla algorithmic and complexity theory. Normally, work in these areas deals with complicated problems over simple structures (popular examples are matrices and graphs). Our problem is interesting because it applies similar ideas to a rather intricate structure, which cannot be characterized as easily, nor understood as fully: the natural numbers, including the divisibility relations that can arise.

In conjunction, the problems we considered have considerable practical impact to the lives of Yogi Bear and Ranger Smith. As such, we expect that our results will also have an impact on the lives of millions of children of all ages, who are fans of these characters.

5. *Postscriptum: A Third Problem*

After the last two rather different problems, the reader should not be surprised that we end our paper with a third problem of an entirely different nature. Nim is probably the best known example of a combinatorial game, whose strategy is simple to understand, yet interesting and nontrivial. The game involves n piles of objects, having sizes $a_1, \ldots, a_n$. A move consists of removing an arbitrary number of objects from a single pile. The player who removes the last remaining objects wins. It is known that the current player has a winning strategy if and only if the exclusive or of a_1 through a_n is nonzero; we write this as $\bigoplus a_i \neq 0$. If that is true, the player should make a move which makes $\bigoplus a_i = 0$. At least one such move exists. If m is the index of the most significant set bit in $\bigoplus a_i$, then at least one a_k must have the m-th bit set. Then, by leaving $a_k \oplus (\bigoplus a_i)$ objects in pile k, we achieve the desired effect. Also note that this is a valid move, since the most significant change is clearing the m-th bit, so a_k decreases.

Our question is how fast a winning strategy can be implemented. This goes beyond the traditional polynomial vs. hard distinction, and asks for the optimal time in which a player can respond to any move made by the opponent throughout the game, while still making sure he is winning. This very appealing, yet unconventional question has an obvious data-structural flavor. Since the last time the player pondered on a move, the configuration was only changed by one move made by the opponent. Thus, by maintaining a suitable data structure representing the configuration, it is reasonable to hope that a good move can be found efficiently (say, in $O(\lg n)$ time). Also note that if multiple winning moves exist, as is usually the case, the player may choose any of them in order to minimize his present and future response times.

For Nim, a classic idea proves effective. We will maintain a balanced binary tree with n leaves, where leaf i holds the number of objects in pile i. Each internal node holds the xor of its two children. When the opponent makes a move, we simply change the value in the corresponding leaf, and recalculate the values on the path to the root. If the xor of all a_i's (stored at the root) is nonzero, we have a winning strategy. To find a suitable move, we first determine the index m of the most significant set bit in $\bigoplus a_i$; this can be done in constant time [FW93]. Exactly one child of the root must have the m-th bit set. The search continues down the tree until we find a leaf with the m-th bit set, which is the pile we want to change.

Thus, a winning strategy for Nim can be implemented with an $O(\lg n)$ response time for generating every move. While obtaining sublogarithmic bounds seems a realistic possibility, we do not know how to achieve that, nor can we prove any nontrivial lower bounds in a general model of computation, such as the cell-probe model. We consider this a very interesting problem for future research.

Acknowledgements

Radu Ştefan invented the wooden fence problem and asked whether a polynomial-time solution existed (personal communication with the authors).

References

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. of Computer and System Sciences*, 47(3):424–436, 1993.