Spring 2021 mBIT Standard Editorial

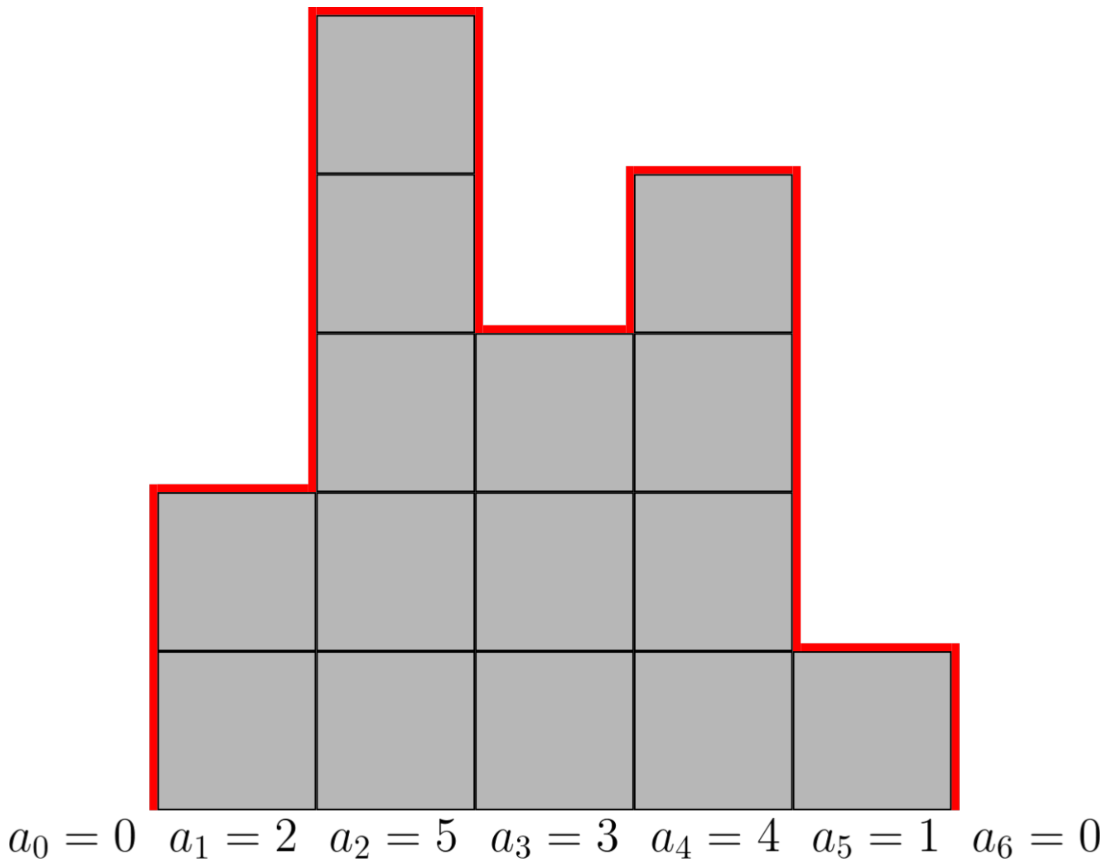
June 12, 2021

This editorial provides the intended solutions to each problem as well as accepted programs in each supported language. In some cases, the given programs may employ different algorithms than the one described in the editorial. For more complex problems, multiple solutions may be given, in which case there will be programs for each solution. Nevertheless, problems are likely to have solutions which are not covered here and we would be interested to hear about any such solutions the reader may devise.

Contents

§1 Mountain Climbing

Let one block length be one meter. The path's length is the sum of its horizontal movement, which is one meter for each column for a sum of N meters, and vertical movement, which varies between columns. Vertical movement occurs when Steve moves between different column heights; note that we must also account for climbing up the first column and down the last column, which we can do by creating "filler" columns $a_0 = 0$ and $a_{N+1} = 0$ on the edges, as shown.



Then, the path's total length is

$$N + \sum_{i=0}^{N} |a_{i+1} - a_i|.$$

Time complexity: $\mathcal{O}(N)$

Problem: Jeffrey Tong
Flavortext: Jeffrey Tong
Editorial: Jeffrey Tong
Code: C++, Java, Python

§2 Digit Sum

We're trying to find a sequence of M digits that sums to N, such that the first digit is not a 0. First, we check to make sure M is a positive number not larger than than $9N$ (if this is not the case, it is impossible because we must have at least one non-zero digit and the maximum sum is $9N$). Note that there is an edge case with $M = 0$ and $N = 1$, because we allow 0 as a valid answer. We can then go through the digits from left to right, and at each location we place the largest digit we can that doesn't push the running sum over M. For example, if $M = 22$ and $N = 5$, we would get 99400. This process ensures that our leading digit is not 0.

Time complexity: $\mathcal{O}(N)$

Problem: Jeffrey Tong
Flavortext: Gabriel Wu
Editorial: Gabriel Wu
Code: C++, Java, Python

§3 Reverse Race

There are many ways to go about implementing this procedure. Here is one way:

1. Break the input string into a list of words. This can be done by reading in the entire input line and splitting along the spaces (for example with the `.split()` method in Python), or by reading in the input one word at a time (for example with `std::cin` in C++).
2. For each word, reverse it, and capitalize it if it was initially capitalized. Reversing a string can be done naively with a for loop, or by using builtin methods like `std::reverse` in C++. Capitalization can be checked using builtin methods as well.
3. Reverse the entire list.

If you have questions on how to do this in your specific language, you can refer to our solution code linked below.

Time complexity: $\mathcal{O}(N)$ where N is the number of characters in the input.

Problem: Gabriel Wu
Flavortext: Gabriel Wu
Editorial: Maxwell Zhang
Code: C++, Java, Python

§4 Apple Orchard

Many people used casework by comparing the ratios of X/Y and C/D, but a simpler solution is to iterate over all possible trades in either direction. First, note that Carlos only needs to trade in one direction because two trades in opposite directions will cancel each other out. Further, one can prove that under the given bounds it is never optimal for Carlos to use more than 1000 trades (doing so would give him an excess of a resource). For any given number of trades (say, trading $3 \cdot C$ apples for $3 \cdot D$ bones), we can quickly calculate the number of each resource he must collect to complete the trades and reach N apples and M bones. In general, there are only $\mathcal{O}(N + M)$ possible numbers of trades in either direction.

Time complexity: $\mathcal{O}(N + M)$

Problem: Gabriel Wu
Flavortext: Gabriel Wu
Editorial: Gabriel Wu
Code: C++, Java, Python

§5 Pokémon Permutation

Note that we only care about the frequency of each character. This is because if we know the frequency of each character in the Pokémon's name, we can just have its name be the repetition of characters in sorted order. For example, the following frequency table:

Character	**Frequency**
a	3
b	5
e	2

becomes the name `aaabbbbbee` (after all, the problem never said the unknown name had to be a legit Pokémon name).

Now let's find the frequency of each character in the input string, and store it in a table $freq$. If the name was repeated $k \geq 2$ times, then we know the name contains $\frac{freq[a]}{k}$ copies of `a`, $\frac{freq[b]}{k}$ copies of `b`, and so on, so k must evenly divide the frequency of each character. Since k can't be larger than the length of the input string, we could just check all possible candidates of k. Alternatively, we could just take the GCD of the frequencies of all characters. If the GCD is 1, then there is no way we can find a $k \geq 2$ that evenly divides the frequency of each character, so the answer is `IMPOSSIBLE`. Otherwise, we let k equal the GCD. After finding k, we'll print each character $\frac{freq[character]}{k}$ times as our answer.

Time complexity: $\mathcal{O}(N)$, where N is the length of the input.

Problem: Jeffrey Tong
Flavortext: Gabriel Wu
Editorial: Maxwell Zhang
Code: C++, Java, Python

§6 Island Isolation

This editorial will use common terminology pertaining to trees in graph theory. If you are unfamiliar with trees, you can refer to online resources (1, 2, 3).

Poptropica can be described as a graph, where the islands are nodes and the connections are edges. Since we're guaranteed we can reach any island from any other (i.e. the graph is connected) and we have $N - 1$ connections, we are in fact working with a tree.

Root the tree arbitrarily. We have to remove bridges in some order so that the following condition remains satisfied:

After removing the bridge from u to v, the only outgoing bridge from v that may remain is towards u.

Which bridges are we allowed to remove first? Let's notice that if we remove bridges pointing towards leaves, then each leaf will only have one bridge towards its parent, so the condition is satisfied. So let's start by removing those. After all of those bridges are removed, notice that we have the same situation with the leaves' parents! If we remove the bridges from the leaves' parents' parents to the leaves' parents, the leaves' parents will each only have one bridge point up towards their parents, since all other bridges pointing down were previously removed. And now, we have the same situation with the leaves' parents' parents! In general, this strategy allows us to remove all bridges pointing downwards, in order from deepest (furthest from root) to shallowest (closest to root).

Great, we're halfway there! After removing all downward bridges, we are left with only upward bridges. But now we can apply the same logic, just in reverse: the root has no outgoing bridges, so we can safely remove all bridges pointing up into the root. Once that is done, we have the same situation with the root's children. And in general, we can remove all bridges pointing upwards, in order from shallowest to deepest.

All of this logic can be implemented via depth first search. You can refer to our solution code for implementation details.

Time complexity: $\mathcal{O}(N)$

Problem: Timothy Qian
Flavortext: Gabriel Wu
Editorial: Maxwell Zhang
Code: C++, Java, Python

§7 Map Matching

In short, this problem wants us to check if one polygon can be mapped onto another via translation and dilation. First, let's make sure the two polygons line up (p_1 corresponds to q_1, p_2 corresponds to q_2, and so on). To do this, we can rotate both lists so that the bottom leftmost point is first in both lists, which guarantees all the points will correspond at the same indices.

Next, note that a polygon P can be dilated to become congruent to polygon Q if each side of P is a constant factor longer than the corresponding side in Q, and if the corresponding angles are congruent with identical orientation. Because working with angles risks introducing floating point error, we can instead utilize vectors: if we describe a polygon as a clockwise path of vectors, then P can be dilated to Q if the magnitude of each vector of P is a constant factor larger than the corresponding vector magnitude in Q, and the unit vectors point in the same direction.

Finally, when checking for the constant factor, we note that this factor may not be an integer. If we break vectors into x and y components and check both separately, the constant factor is always rational, so we can represent it using fractions to maintain perfect precision.

Time complexity: $\mathcal{O}(N)$

Problem: Maxwell Zhang
Flavortext: Gabriel Wu
Editorial: Maxwell Zhang
Code: C++, Java, Python

§8 Street Layout

In this problem, we are looking for a way to place N houses on a number line, such that the maximum distance from a house to a fire station is as small as possible. First, note that if we have a maximum distance m, we can check if there exists an arrangement of houses such that their maximum distance from a station is at most m. To do this, we first sort the fire stations by position, and greedily place houses as soon as a position is available. This can be done in $\mathcal{O}(N)$, as station i and $i+1$ can hold a total of $\min(2m, a_{i+1} - a_i - 1)$ houses between them. After iterating through all of the adjacent stations and accounting for the houses that can be placed on the left/rightmost edges, we can check if we have any houses left to place. If so, then the solution must be greater than m, and otherwise, the solution must be less than or equal to m.

Thus, the final step of the problem is to binary search on the answer (the value m) to find the optimal distance. Overall, this solution runs in $\mathcal{O}(N \log N)$.

Time complexity: $\mathcal{O}(N \log N)$

Problem: Jeffrey Tong
Flavortext: Gabriel Wu
Editorial: Aaron Mei
Code: C++, Java, Python

§9 Grid Shuffling

First, ignore the colors (so imagine that all squares are identical). After a few moves are done, all of the squares will be bunched up into a corner. Once the squares are in a corner, **after 4 rotations, the squares will be in the same shape as before**. There are many ways to think about this, one of which is to imagine the shuffles as reflections of the shape formed by the squares. Left/right shuffles reflect across the vertical line at the center of the grid, and up/down shuffles reflect across the horizontal center. So every 4 moves, there are two vertical reflections (which cancel out) and two horizontal reflections (which cancel out as well), leaving the same shape as before.

Now we bring the colors back into play. While the shape is the same after 4 moves, the colors may necessarily not be. However, every square at each location will be mapped to some other position in the new shape. Since the transformation is the same every time, this mapping will be the same every four rotations, so the rotations essentially turn the squares into a permutation of themselves, many times.

Permutations can be decomposed into cycles. For a permutation $p_1, \ldots, p_n$, a cycle of length k means that if you start at any location x on the cycle, and jump from x to p_x a total of k times, you'll end up back at x. So representing this cycle as a circular array, to find the location of the square at x after m moves, we add m to the index of x in the cycle and take it modulo k (this might be seen more clearly in the attached code).

The plan is to do "a few" moves at the beginning so that the squares are in a corner, find the permutation, then decompose it into cycles and find the new locations after the rest of the moves are done (remember to divide by 4, since the permutation represents doing 4 moves). We may have a few moves left to do, which can be simulated manually (as it's at most 4).

Some implementation notes: 2 rotations is enough for "a few". To implement the shuffles, you can implement up, left, down, and right separately, or you can rotate the grid 90 degrees (counterclockwise) each time so you only have to implement downward gravity. Cycle finding can be done manually, but for an implementation-speed tradeoff, you can use binary lifting (ignore the LCA part) to make the implementation much simpler. The official solution in C++ implements these ideas.

Time complexity: $\mathcal{O}(N^2)$ or $\mathcal{O}(N^2 \log K)$, depending on implementation.

Problem: Gabriel Wu
Flavortext: Gabriel Wu
Editorial: Colin Galen
Code: C++, Java, Python

§10 Goomba Grouping

First, let's determine when the answer is `-1`. After trying some cases on paper, we will find that Bowser's algorithm is always optimal when $N \leq 4$.

$N = 1$: The only possible partition is one Goomba on its own, for a difference equal to its own weight.

$N = 2$: For two Goombas of positive weight A and B, it's always optimal to place them in separate groups, since $|A - B| = \max(A - B, B - A) < A + B$, and Bowser's algorithm does just that.

$N = 3$: For three goombas of positive weights $A \leq B \leq C$, Bowser will place C in one group and A and B in the other. This is always optimal. Placing three goombas in one group gives the worst difference. Let's say we isolate B instead (proof is same for isolating A). Our difference becomes $A + C - B$. Compare that to isolating C for a difference of $\max(A + B - C, C - A - B)$, and we note that $A + C - B \geq A + B - C$ and $A + C - B \geq C - A - B$, so isolating C is always more optimal.

$N = 4$: For four goombas of positive weights $A \leq B \leq C \leq D$, Bowser will partition into $\{A, D\}$ and $\{B, C\}$ if $B + C > D$, and $\{A, B, C\}$ and $\{D\}$ otherwise. You can do more math to show that this is always optimal.

Now for $N \geq 5$ and any value of K, we can always construct a counter case to Bowser's algorithm. First, let's understand the flaw with Bowser's reasoning. Bowser assumes that it is always optimal to keep the two groups as even as possible at each intermediate step, so his logic breaks when it is optimal to place two large weights in one group, and many small weights that add up to the same amount in the other group. Knowing this, let's begin our construction with two large goombas each of weight 10^{18}, which we want to be in the same group in the optimal partition.

Since the optimal difference needs to be K, let's make the weights in the other group sum up to $S = 2 \cdot 10^{18} - K$. If N is odd, then we can fill the other group with goombas of weight $\left\lfloor \frac{S}{N-2} \right\rfloor$ or $\left\lfloor \frac{S}{N-2} \right\rfloor + 1$, such that they add up to S. Bowser's algorithm will place the two massive 10^{18} goombas into different groups, and because there are an odd number of the smaller goombas, his final difference will be of order of magnitude $\sim \left\lfloor \frac{S}{N-2} \right\rfloor$. Since $K \leq 10^9$ and $N \leq 20$, $\left\lfloor \frac{S}{N-2} \right\rfloor \gg K$, so this is ok.

What about even N? This can be remedied by adding a goomba of weight 1, which will hardly change the difference between the two groups from Bowser's algorithm because 1 is so small in comparison. So the even N case reduces to the odd N case. In fact, this observation also leads to an alternative solution, which is to simply pad the set with $N - 5$ ones, and then hard-code a solution for the $N = 5$ case.

Time complexity: $\mathcal{O}(N)$

Problem: Maxwell Zhang
Flavortext: Gabriel Wu
Editorial: Maxwell Zhang
Code: C++, Java, Python

§11 Rabbit Subtraction

For the moment, let's assume that N is of the form 2^m. Let's define the function $dp[i][j]$, with $0 \leq i \leq n-1$ and $0 \leq j \leq m$, to be the value of the first rabbit in the line if we rotated so that index i is the start of the sequence, and j spells have been used. Note that this value only depends on $a_i, a_{i+1}, \ldots, a_{i+2^j-1}$.

We can initialize $dp[i][0]$ to be $a[i]$ and find $dp[i][j]$ from the bottom-up as follows:

$$dp[i][j] = dp[i][j-1] - dp[(i+2^{j-1})\%N][j-1]$$

Our final answer would be $\max\limits_{0 \leq i < n} dp[i][m]$.

Now let's solve the problem for N not a power of 2 using the same dp table. Let $score(i, x)$ be Willy's final score if the original array only included rabbits $a_i, \ldots, a_{i+x-1}$ (all indices taken mod N).

To get $score(i, x)$, where x isn't a power of 2, there must be $s := \lceil log_2 x \rceil$ stages of merging. At the second-to-last stage with only 2 rabbits remaining, the first 2^{s-1} elements would all be merged into one group, with a score of $score(i, 2^{s-1})$. The remaining $N - 2^{s-1}$ elements must also have culminated into one group at this point. Therefore,

$$\begin{aligned} score(i, x) &= score(i, 2^{s-1}) - score(i + 2^{s-1}, x - 2^{s-1}) \\ &= dp[i][s-1] - score(i + 2^{s-1}, x - 2^{s-1}) \end{aligned}$$

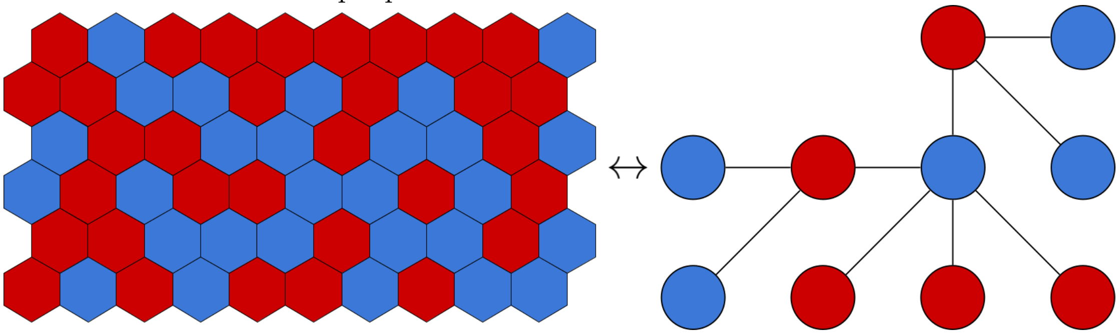
By induction, $score(i, x)$ is an alternating sum of scores of chunks with sizes of decreasing bits of x. For example, $score(0, 7) = dp[0][2] - dp[5][1] + dp[7][0]$ and $score(0, 5) = dp[0][2] - dp[5][0])$. Thus, we can compute any $score(i, x)$ in logarithmic time, as long as our dp values are precomputed. Our final answer is the maximum over all valid i of $score(i, N)$.

Time complexity: $\mathcal{O}(N \log N)$

Problem: Gabriel Wu
Flavortext: Gabriel Wu
Editorial: Claire Zhang
Code: C++, Java, Python

§12 Squid Art

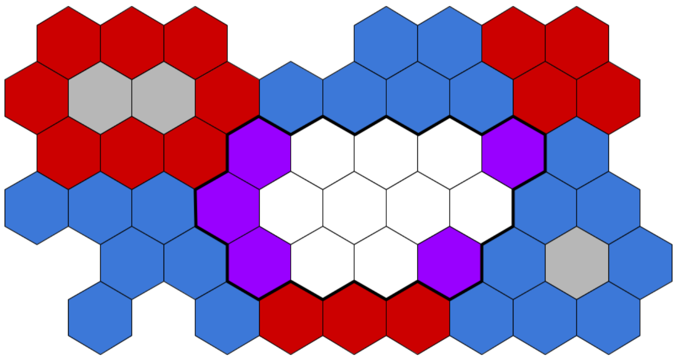
If we view each component of uniform color as a single vertex and draw an undirected edge between each pair of adjacent components, we can produce an undirected graph G equivalent to the original grid. By definition, G is bipartite. Notice that applying this transformation to the sample produces a tree:



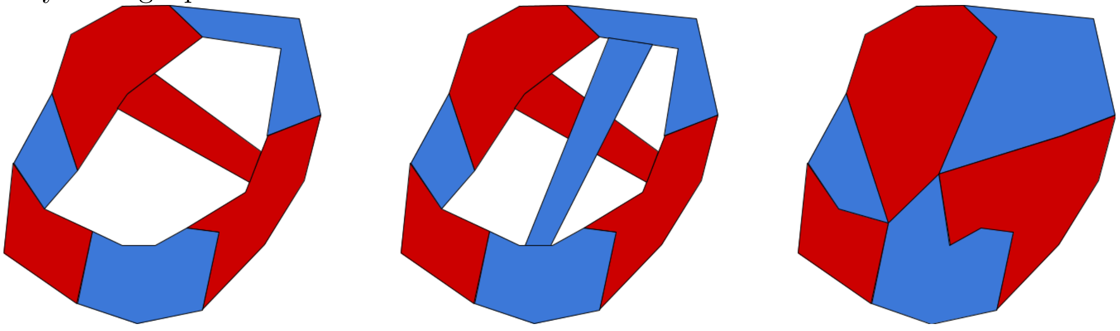
Theorem 12.1

G is a tree.

Proof. Assume for the sake of contradiction that G has a cycle C; since C must have an even length, it has at least two red and two blue vertices. The loop L in the grid that transforms to C thus has at least two red and two blue components. L cannot enclose a central "hole", as otherwise some cells (shown in purple) in the hole would neighbor components of both colors; such cells must actually be part of a component in L, not in the hole. Individual components can have holes, but these do not affect the proof.



We now analyze the center of L. WLOG, consider any two red components in L: they cannot connect through the center, and they also cannot be cut off by a contiguous wall of blue cells (or two blue components would connect through the center.) Thus, the only way the center of L can close up is that some pairs of same-color components touch, but only at single points:



In a hexagonal grid, however, no two cells touch at exactly one point, so the third situation is actually impossible, a contradiction.

Thus, G is a connected acyclic graph. □

Theorem 12.2

The minimum number of operations to turn the whole grid into a single color is the radius of G.

Proof. Consider the equivalent of filling in a component in the original graph on G: we switch the color of a vertex, which is equivalent to merging the vertex and its neighbors into a single vertex of the new color.

Let r be the radius of G. This pattern makes it easier to see that filling in the component corresponding to a centroid of G decrements r, and repeating the process r times reduces G to a single vertex, so r operations are sufficient.

To show that r operations are necessary, consider a diameter $D \subseteq G$. One operation can reduce at most 3 vertices (the chosen vertex and one neighbor to either side) of D to a single vertex, decreasing the diameter by at most 2. Since it is well-known that radius $= \lceil \frac{\text{diameter}}{2} \rceil$ for a tree, the radius is therefore decreased by at most 1. □

Computing the diameter and radius of a tree is a well-known problem solvable in linear time. There is a bit of casework dealing with parity to be done at the end because the problem requires all of the tiles to be blue (not just the same color).

Time complexity: $\mathcal{O}(NM)$

Problem: Gabriel Wu
Flavortext: Timothy Qian
Editorial: Jeffrey Tong
Code: C++, Java, Python

§A Among Us References

Just for fun, we hid three references to the infamous game *Among Us* in our problem sets. Each reference was in a problem shared by the Standard and Advanced divisions.

- In *Pokémon Permutation*, the first letter of each sentence in the first paragraph spells `AMONGUS`.
- In *Goomba Grouping*, the sample output for the second test case contains eight numbers. If you convert these numbers to letters (1=`A`, 2=`B`, . . . , 26=`Z`), it spells `IMPOSTOR`.
- The second pretest in *Squid Art* depicts a character from the game:

```
19 30
111111111110000000000011111111
111111110000000000000000011111
111111110001111111000000001111
111111100011100000000000000111
111111100011000001111111000001
110000000111000001111111111001
100000000111000000000000000001
100011000111100000000000000001
100011000111110000000000000011
100011000111111111111111100011
100111000111111111111111100011
100111000111111111111111100011
100011000111111111111111100011
100001000111111111111111100011
110000000111110000000001000111
111111100111110001000011000111
111111100111110001000111000111
111111100000000001000000001111
111111100000000011111111111111
```