

2026-08-26 • 14 minute read

The Depths of JavaScript: Minesweeper in 247 Bytes

247 characters.

```
b=e=>document.body.innerHTML=m(73,e||[~9,..."1172711",~9].map(j=>m[i-=j]?++e:i).map(j=>b(b[i=j]^1|e),b[i]=!m[0--,i]*~e));b(m=t=>t--?'<a onmouseup=b(b[i=${t}]^=event.which)>${t%9?"*#F"[x=b[t]??(0-=m[t]=Math.random())<1,1)]??~x:"<p>"}'+m(t):0,0=64)
```

These **247 characters of JavaScript code** is all it takes to run a minimal version of [Minesweeper](#) in your browser.

It features a randomly generated 8x8 board, lets you open cells with left click, toggle flags with right click, opens cells with zero neighboring mines recursively, and has basic win detection.

[Try it on JSFiddle!](#)

Why??

A couple years ago, I decided to [code golf](#) a working version of Minesweeper for fun. I wrote a 658 byte version in a single evening, and kept working on it throughout the next few weeks, eventually bringing it down to just 399 bytes.

That's when I assumed I hit a wall and stopped working on it.

Recently I was cleaning up my drive when I found this code golf, and thought it would be quite cool to write a blog post about it. But then I instantly found a way to make the code shorter. Then another. And another.

I started sharing my progress on a Discord server, where **DNEK** ended up joining me in my code golfing efforts, and after a couple of months of shared effort and so much trial and error, we brought this down to just 247 bytes.

How???

And now, the part that everyone is actually curious about: *How is this even possible???*

Please keep in mind that this is a highly technical blog post, and you will need some **JavaScript** knowledge to understand this—I am not going to explain literally everything. Feel free to ask any questions in the comments though!

To start off, let's format this monstrosity so that it's a little more readable:

```
b = (e) => document.body.innerHTML = m(
  73,
  e || [~9, ..."1172711", ~9]
    .map(j => m[i -= j] ? ++e : i)
    .map(
      j => b(b[i = j] ^ 1 | e),
      b[i] = !m[0--, i] * ~e
    )
);

b(
  m = (t) => t-- ? `<a
onmouseup=b(b[i=${t}]^=event.which)>${
  t % 9
  ? "*#F"[
    x = b[t] ??= (
      0 -= m[t] = Math.random() < .1,
      1
    )
  ] ?? ~x
  : "<p>"
}` + m(t) : 0,
  0 = 64
)
```

The code consists of two parts: the function **b** which handles **game logic**, and the function **m** which **generates the UI and the board**.

It also initializes **0=64** - this will be used to show the amount of unopened safe cells.

One of the first optimizations that you might see immediately is that **m** is actually defined inside of a function call:

`b(m=t=>..., 0=64)`. This works perfectly fine in JS, and saves one byte over using a delimiter:

- `b=...; m=...; b()`
- + `b=...; b(m=...)`

Board Initialization and Rendering

Let's see what happens when the game starts. It calls the `b` function, which then calls `m(73, ...)` and assigns its output to `document.body.innerHTML`.

`m` starts with `t=73` and generates the HTML recursively up until `t` is zero, adding 0 (the number of unopened safe cells) at the end, like so:

```
t--  
? '<a...`' + m(t)  
: 0
```

Every row is put in a separate `<p>` element, with each cell wrapped in an `<a>`. It uses `onmouseup` to detect left and right clicks.

As you might have noticed, `b` actually passes two arguments to `m`. But `m` only accepts one argument! The other argument is ignored and is there just to save two bytes:

- `b=e=>{someLogic; m(73)}`
- + `b=e=>m(73, someLogic)`

HTML shenanigans

But where are the closing tags?! The original code does not have any `</p>` or `</a>`, or even a single `/` for that matter! That is a

very neat trick that took me a long time to discover.

Browsers don't allow `<a>` to contain itself, nor do they allow `<p>` to contain itself. When a new tag starts, the HTML parser automatically closes the previous one.

```
<p><a>1<a>2<p><a>3<a>4
<!-- turns into -->
<p><a>1</a><a>2</a></p>
<p><a>3</a><a>4</a></p>
```

It's important to note that HTML ignores line breaks by default — you have to use `<br>`, or some block element like `<p>` or `<div>`, or a white-space preserving element like `<pre>`, or some custom CSS.

This makes `<a>` and `<p>` the perfect set of elements for us—without a set link, `<a>` acts as a regular inline element with no extra functionality—and `<p>` acts as line breaks for rows.

Player Input

Another thing that we have to consider is all the different mouse events. My first versions used `onClick=b(...)` for opening cells and `oncontextmenu=return!f(...)` for toggling flags.

`f(...)` was responsible for toggling a flag, and always returned a truthy value, which then got inverted and returned. All inline event listeners in HTML are secretly functions, and as you might know, returning `false` from an event listener is the equivalent of doing `e.preventDefault()`.

This served as a good way of preventing the context menu from showing up on right-click.

(by the way, `return!` is perfectly valid JS syntax, and actually saves three bytes!)

```
- oncontextmenu="return !f(...)"  
+ oncontextmenu=return!f(...)
```

But in code golf, we don't care about how playable something is, do we?

The current version uses a single event listener, `onmouseup`, which also calls the same function and uses `event.which` to tell which button was pressed (`1` for left-click, `2` for middle click, and `3` for right-click)

Unfortunately, there's a small trade-off that I had to make: the context menu no longer gets prevented, so you actually have to right click twice on a cell to toggle a flag. We deemed this to be a worthy trade-off, however.

Doing Everything At Once

You might think that before displaying the board, we actually need to initialize it. I mean, we have to know what do we want to show, right? Where do we store game state anyway?? There's not a single array in the code!

When coding in **JavaScript**, you have to remember that everything is an object. You can assign any value to any object with any key, including array indexes.

A Minesweeper game typically needs to hold two game states: the visible board with flags and digits, and the actual board dictating where the mines are. We just so happen to have exactly two objects that we can use — the `b` and `m` functions!

Because we iterate over `t=73..0`, we can simply assign `b[t]` and `m[t]` in the recursive loop. This creates a one-dimensional "array" for the visible board and the mines.

For this, I used a clever trick with the `??=` operator, and the fact that assignments can be used as expressions.

```
x = b[t] ??= (
    0 -= m[t] = Math.random() < .1,
    1
)
```

When `b[t]` is defined (not null or undefined), this simply sets `x` to the current value of `b[t]` and does nothing. But when `b[t]` is *not* defined, it goes into the assignment, where we:

1. Assign the outcome of `Math.random() < 0.1` to `m[t]` — this gives a ~10% chance for any given cell to be a mine.
2. Subtract that boolean from `0`, abusing type coercion: subtracting `true` from a number is equivalent to decrementing it by one, and subtracting `false` results in no change.
3. Finally, this is wrapped in parentheses — `(..., 1)` — which means the expression always evaluates to just `1`.

This results in all cells in the visible board `b` being initialized with `1`, and a randomly generated mine field `m` with a 10% chance for a mine. After initializing, `0` is left with the total amount of safe cells that the player has to uncover.

Displaying The Board

This initialization is wrapped in a condition: `t % 9 ? ... : "<p>".`

This means that all columns from `1` to `8` get shown and are initialized with mines, and leaves us with one extra invisible empty column on the right. This extra column is shown as `<p>`, essentially acting as a separator between rows.

The visible board is stored using integers, which are then mapped through a lookup string:

```
"*#F" [
    x = ...
```

] ?? ~x

This results in ***** for 0 (mine), **#** for 1 (default unopened state), and **F** for 2 (flag).

We can store opened cells as negative integers: **-1** for zero neighboring mines, **-2** for one neighbor and so on. Using negative indexes in strings or arrays always results in **undefined**, which triggers the ?? operator, where we can negate **x** using **~x** (bitwise NOT).

It's equivalent to doing **-x - 1**, which is perfect for us here! It turns **-1** into **0**, **-2** into **1** and so on. The resulting value is then used to show the current cell wrapped in **<a>**.

Some of you might've noticed that I actually have a stray empty **<a>** before every **<p>**: this is an intended optimization! Inline elements with no content are essentially non-existent for the player—they have a size of 0x0, so you can't click on them or see them at all.

```
- (i%9 ? `<a>${...}` : "<p>") + m(t)
+ `<a>${i%9 ? ... : "<p>"}` + m(t)
```

Due to operator precedence, **a?x:y+b** will get interpreted as **a?x:(y+b)**, which is clearly not what we want here. Having the ternary operator outside of **\${...}** would mean that we have to wrap the whole condition in parentheses, which wastes two bytes.

Opening Cells

The function **b** accepts a single argument, **e**. Any truthy value short-circuits its main cell opening logic via logical OR (**||**):

```
b = (e) => {
  document.body.innerHTML = m(
    73,
```

```
        e || ...
    );
}
```

This lets us control `b`'s behavior in a very compact way: a falsy value will open the cell and then update the board, and a truthy value will only update the board.

Since all unopened cells are initialized with `1`, XORing a cell's value with `1` (e.g. `b[i]^1`) tells us if the cell is currently unopened (since `1^1 = 0`). It also serves as an out-of-bounds check because accessing an invalid index returns `undefined`, and `undefined^1` evaluates to `1`—if we pass that to `b`, it will short-circuit.

Let's look at the `onmouseup` event from `m`:

```
`<a onmouseup=b(b[i=${t}]^event.which)>...`
```

This assigns the cell's index to `i` (a global variable accessed later by `b`), mutates `b[i]` by XORing it with `event.which`, and calls `b` using the resulting cell value as `e`.

`event.which` is either `1` (left click), `2` (middle click), or `3` (right click).

With this,

- Left clicking changes an unopened cell to `1 ^ 1 = 0`. Because `0` is falsy, `b` runs its main logic to open the cell.
- Right clicking changes an unopened cell to `1 ^ 3 = 2`. Flags are represented by `2` in the lookup string, so the cell now shows a flag. Because `2` is truthy, `b` short-circuits and does not open the cell.
- Right clicking a flag changes it to `2 ^ 3 = 1`, resetting it back to an unopened cell. `b` short-circuits again because `1` is truthy.

This also prevents you from opening an already opened cell or a flag, albeit with the side effect of modifying `b[i]`, which looks as if it increments and decrements the cell. The trade-off is worth it though, as the bug is purely visual and you can fix it by clicking again.

When `b` recurses, it calls itself with `b[j] ^ 1 | e`. This expression evaluates to `0` only if the target neighbor is unopened (`b[j] ^ 1 = 0`) and the current cell has zero neighboring mines (`e = 0`).

This prevents infinite recursion, prevents out of bound indexes, and handles flag placement, all thanks to a single variable and XOR. And what makes this an even crazier solution is that because `b`'s cell opening logic is guaranteed to only run when `e=0`, we also have a free variable which we can use as an accumulator to count how many neighboring mines a cell has!

Tip: When working with integers or booleans, and when you don't care about short-circuiting, you can use bitwise operations (`|` and `&` instead of `||` and `&&`), which are one character shorter!

```
- b[j]^1 || e
+ b[j]^1 | e
```

Thinking In One Dimension

We have two separate 2D states, each stored as a 1D array. Calculating neighboring mines and opening empty cells recursively will require us to think in one dimension.

In a typical Minesweeper implementation, you would have a double `for`-loop for this:

```
for (let dy = y - 1; dy ≤ y + 1; dy++) {
  for (let dx = x - 1; dx ≤ x + 1; dx++) {
    // check if (dx, dy) is inbounds,
```

```
        // increment neighbor counter if it's a
mine,
        // etc
    }
}
```

But you could also hardcode the offsets, making an array of eight differences:

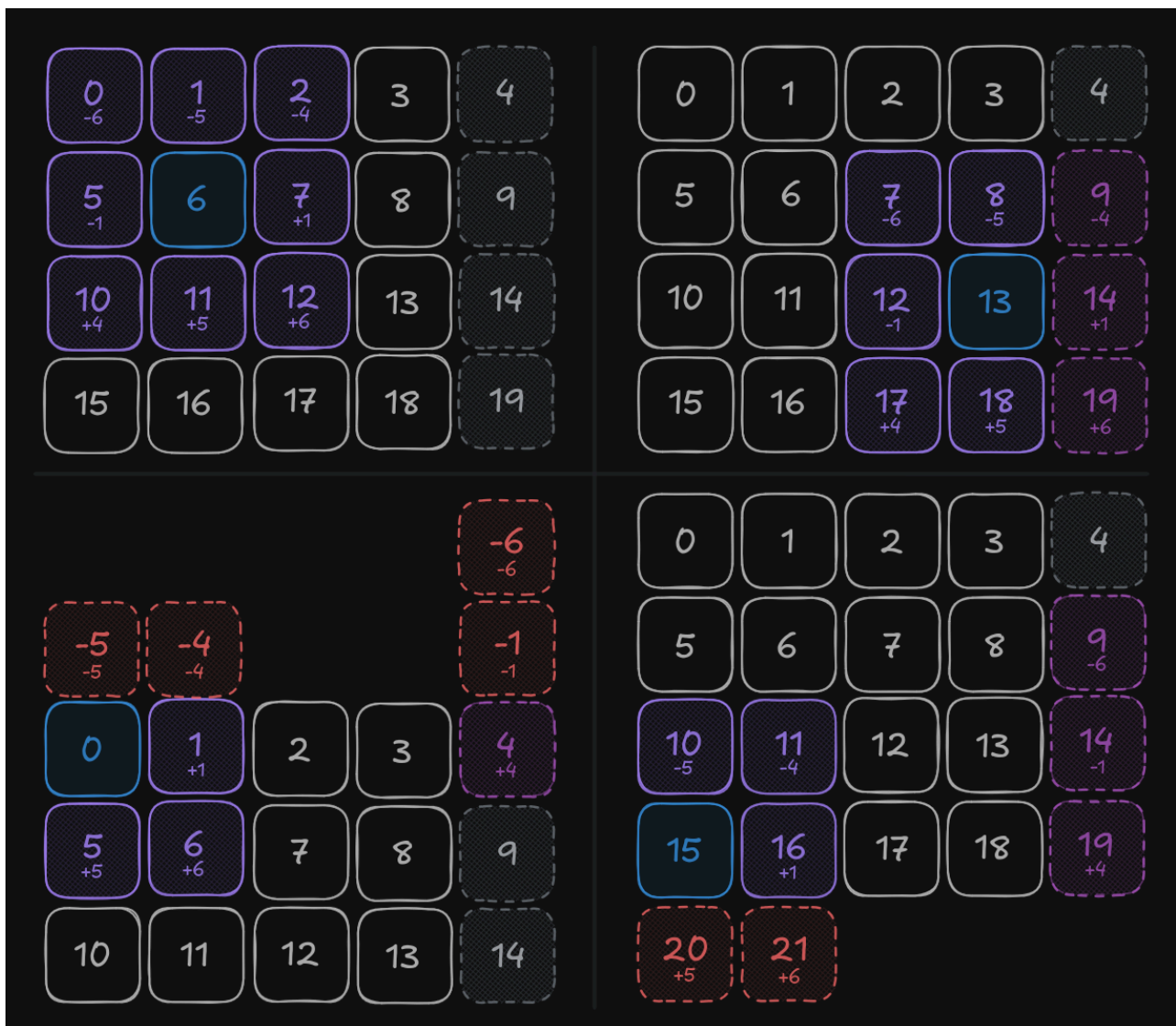
```
[
  [-1, -1],
  [0, -1],
  [1, -1],
  [1, 0],
  // ...
]
```

and then iterate over that...

This is exactly what I did in this code golf. Hardcoding offsets might seem silly for a 2D implementation, but its quite trivial in 1D, provided the board size is fixed.

JavaScript is perfectly fine with you using out-of-bounds indexes on an array—that'll simply return `undefined`. And what helps us even more is the fact that we already have a hidden column on the right; that's enough to take care of all problems with out-of-bound indexes!

An example of how this would work on a 4x4 board:



If we check a cell on the sides or corners, applying these offsets will either make invalid neighbors land on the hidden column (visualized as magenta), or completely out-of-bounds of the array (visualized as red).

When checking if a cell at a given offset is a mine, we just need to check if its truthy. The hidden column does not have any mines, and `undefined` is falsy.

This results in a very compact loop already, but we can make it even better using **deltas**.

Instead of storing full offsets and wasting bytes on several negative signs, we can just keep modifying `i`:

```
[-10, 1, 1, 7, 2, 7, 1, 1, -10].map(
  j => m[i -= j]
```

```
    ? e++  
    : i  
  )
```

This will start from `i+10`, keep subtracting small 1-digit deltas from `i`, and then return `i` back to its original state.

For every mine it finds, it increments `e` (the free accumulator that `b` has, as explained earlier). If no mines were found, this produces an array of indexes that we can use to recursively open all neighbors. If at least one mine was found, then you don't want to recurse anyway, so keeping a valid list of indexes is not needed.

This *does* count the original cell twice, but that doesn't matter here—if the cell you clicked on is a mine, you'll see `*`; and if it isn't, counting it doesn't change anything.

Finally, we can use some tricks to make the array very compact:

```
[~9, ..."1172711", ~9].map(  
  j => m[i -= j]  
    ? ++e  
    : i  
)
```

The spread operator is the shortest way to split a string into individual characters in JS. In our case, it produces an array of digits as strings.

Subtracting from `i` instead of adding is necessary because of the way type coercion works:

- Adding a string to a number will coerce the number into a string: `3+"1"` is `"31"`.
- Subtracting a string from a number will coerce the *string* into a number: `3-"1"` is `2`.

We also use `~9` (bitwise NOT) instead of `-10` to save two bytes.

Updating the cell and recursing

After producing the array, we call `.map` on it to update the visible board state for the opened cell, and recurse. This uses the same trick with extra arguments as I mentioned earlier:

```
.map(  
  j => b(...),  
  // this is evaluated before the .map call  
  b[i] = !m[0--, i] * ~e  
)
```

We first decrement `0` and check if the current cell is NOT a mine with `!m[0--, i]`. The index expression evaluates to `i`, but silently decrements `0` as a side effect.

Then we multiply that by the bitwise NOT of the mine counter:

```
!m[0--, i] * ~e.
```

Booleans coerce to integers, so `false * x = 0`, and `true * x = x`.

We save this to `b[i]`. This produces `0` if the current cell is a mine, `-1` if the cell has no neighboring mines, `-2` if it has one neighbor, and so on.

After that, `.map` calls this function on every index in the array:

```
j => b(b[i = j] ^ 1 | e)
```

This updates the global variable `i` to `j`, and, as explained earlier, constructs the argument to `b` such that it only runs its main logic when the target cell is unopened (`b[j]^1 == 0`) and there are no neighboring mines (`e == 0`).

After all this is done, `m` finally gets called, and its result is assigned to the body's `innerHTML`, showing the player an updated board with the newly opened cell(s).

Now what?

That's it. That's all it takes to run Minesweeper.

It's definitely not the *smallest* possible Minesweeper out there. You can make this version so much smaller by just removing most of the features and using `prompt` for displaying the board and getting player input.

An idea I had is to abuse the fact that `prompt` freezes execution until the user submits the prompt—this means we can use a neat trick with `Date` to get random numbers during initialization (albeit requiring you to press enter like a hundred times to start the game):

```
- Math.random()<.1  
+ new Date%7<1
```

But I started with this specific feature set, and the game went from 658 bytes down to just 247, with **zero main features removed**.

This version is probably very close to being the smallest possible, but unfortunately it's impossible to know that due to the Kolmogorov Complexity being uncomputable. There might be other *entirely* different approaches that achieve the same result in fewer bytes, but we'll never know until someone comes up with them.

Final Thoughts

There was a lot of trial and error that went into this. Me and DNEK have tried so many different ideas and approaches—sometimes making very significant changes to the code, sometimes going back to older ideas, sometimes getting excited about a potential

optimization only to find out it doesn't work. Every single byte has been carefully put together and thought of.

I wouldn't be surprised if someone manages to make this smaller right after I publish this blog post, but I am personally very satisfied with the result.

This might be pointless in the grand scheme of things, but that's the beauty of recreational programming. I've viewed code as a form of art ever since I was a kid, and I'm glad to still find joy in making silly things simply for the sake of making them, especially in this day and age.

Please consider [supporting me](#) if you like what I do! You can also [reach out to me](#) if you'd like to work together, I am currently looking for remote opportunities!

3,120 words • 100 paragraphs

0 reactions



[1 comment](#) – powered by [giscus](#)

Oldest

Newest

Write

Preview

Aa

Sign in to comment



Sign in with GitHub



m-i-z-u-k-i [2 days ago](#)

okay

↑ 3



0 replies