silly compile time tricks

I've been working on

`postcard-rpc`

problem:
de-duplicating lists

let's say I have a list:

```
[1, 5, 10, 5, 20, 1]
```

I want the unique items:

```
&[1, 5, 10, 20]
```

proc/decl macros: only work with tokens

would work with

```
[1, 5, 10, 5, 20, 1]
```

wouldn’t work with

`LIST_WITH_DUPES`

what about `const fn`?

We start with

`[u32; 6]` or `&[u32]`

We want to end up with `[u32; 4]` or `&[u32]`

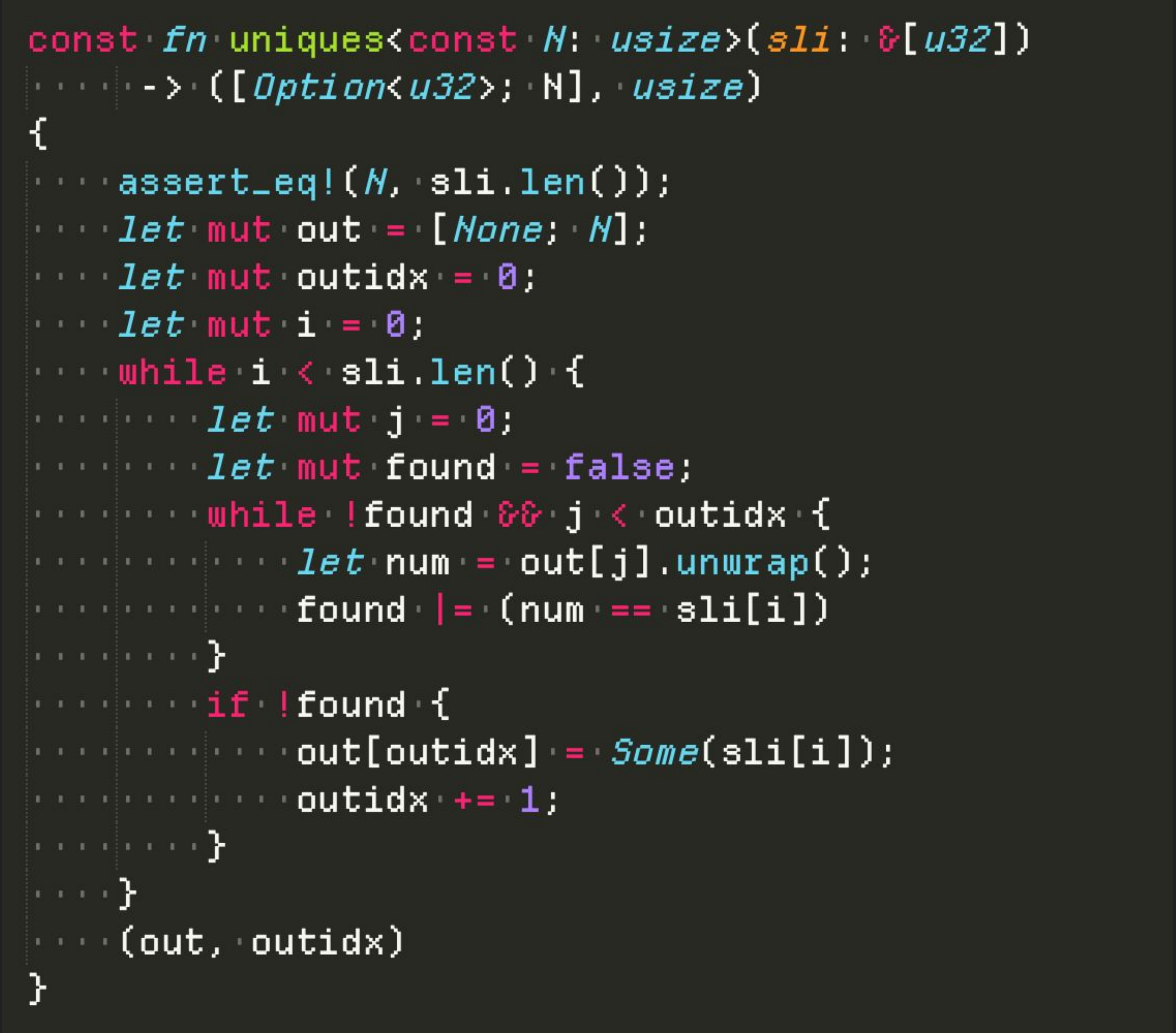
problem:

There's no `alloc` in `const`

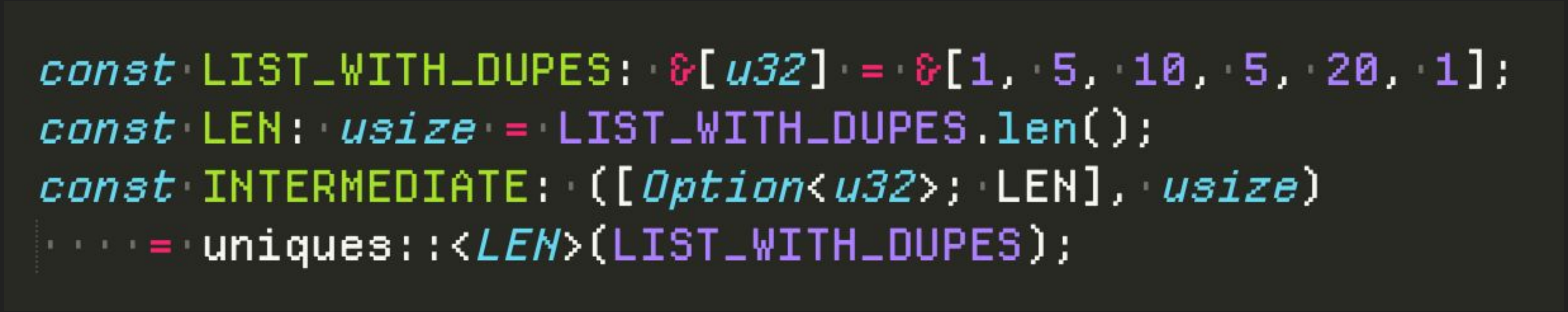
(yet)

step one:

```
[u32; 6]
->
([Option<u32>; 6], usize)
```

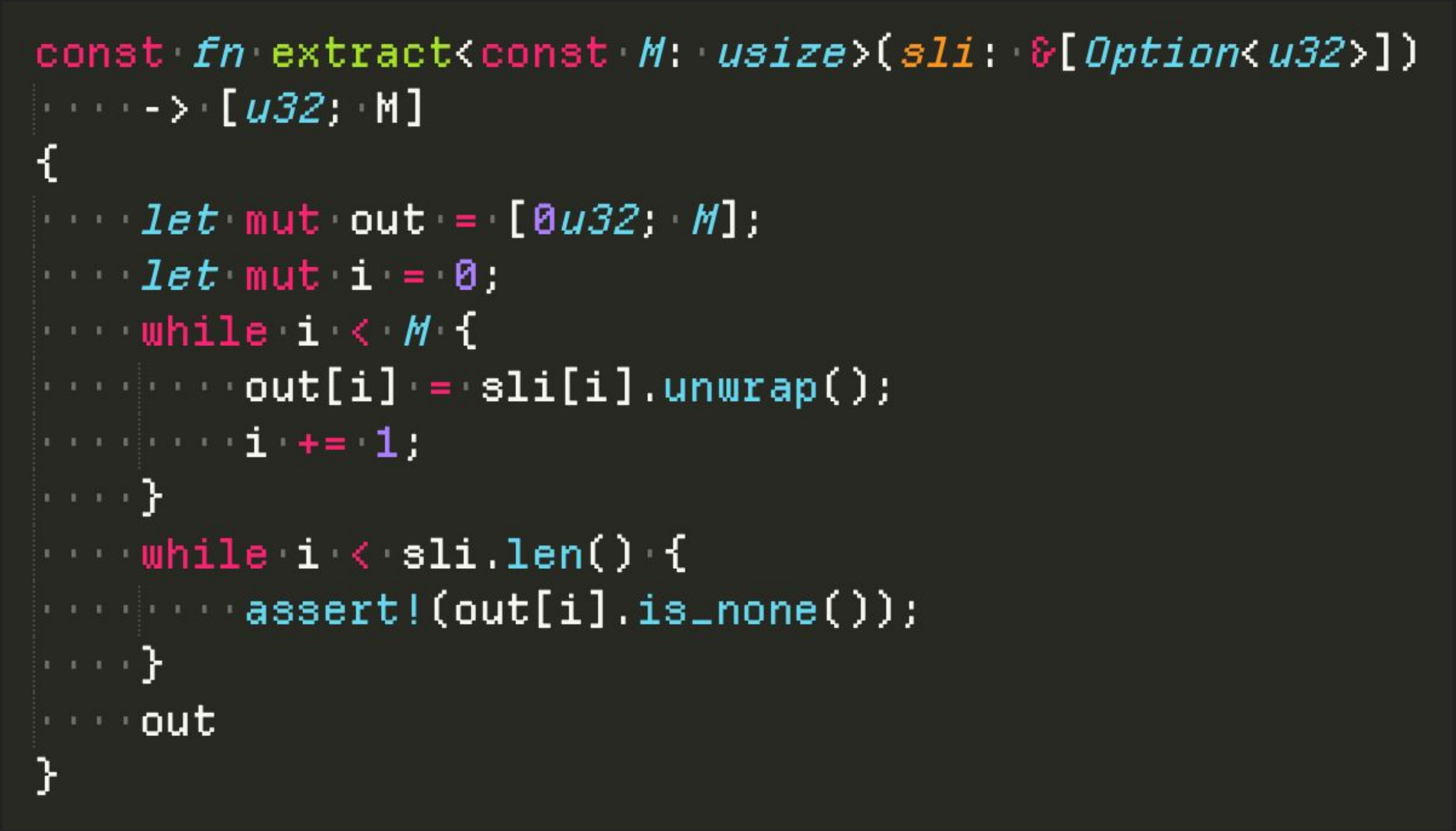


```
const fn uniques<const N: usize>(sli: &[u32])
     -> ([Option<u32>; N], usize)
{
    assert_eq!(N, sli.len());
    let mut out = [None; N];
    let mut outidx = 0;
    let mut i = 0;
    while i < sli.len() {
        let mut j = 0;
        let mut found = false;
        while !found && j < outidx {
            let num = out[j].unwrap();
            found |= (num == sli[i])
        }
        if !found {
            out[outidx] = Some(sli[i]);
            outidx += 1;
        }
    }
    (out, outidx)
}
```

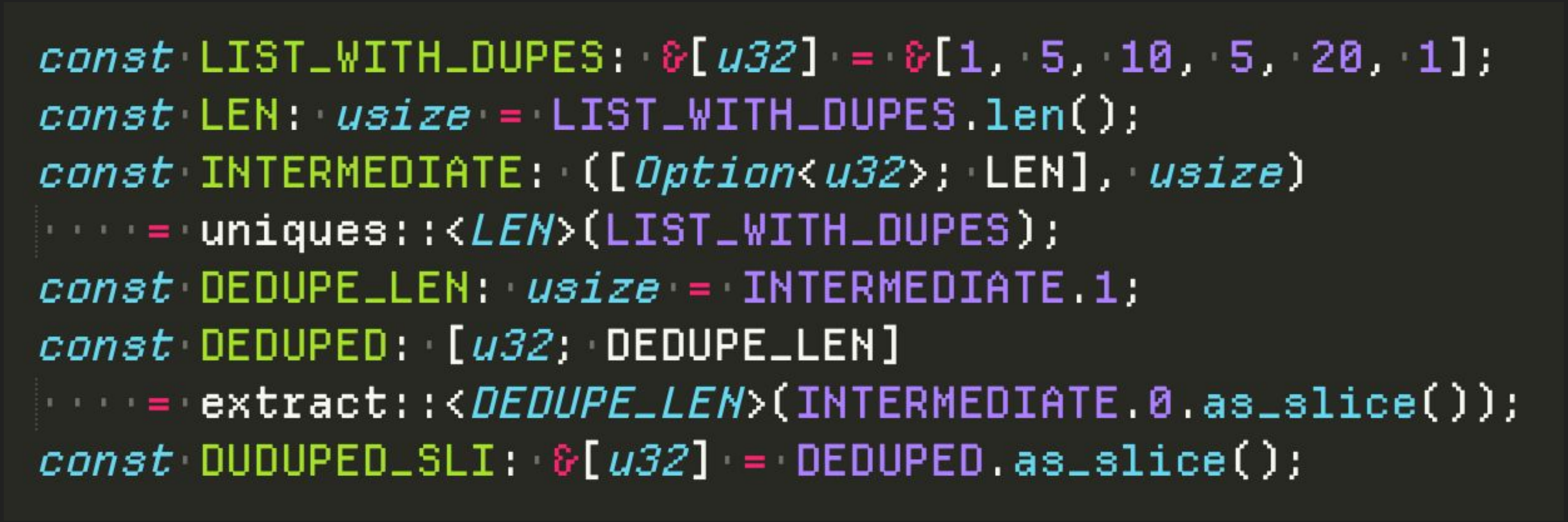


```
const LIST_WITH_DUPES: &[u32] = &[1, 5, 10, 5, 20, 1];
const LEN: usize = LIST_WITH_DUPES.len();
const INTERMEDIATE: ([Option<u32>; LEN], usize)
    = uniques::<LEN>(LIST_WITH_DUPES);
```

that's not QUITE what we want

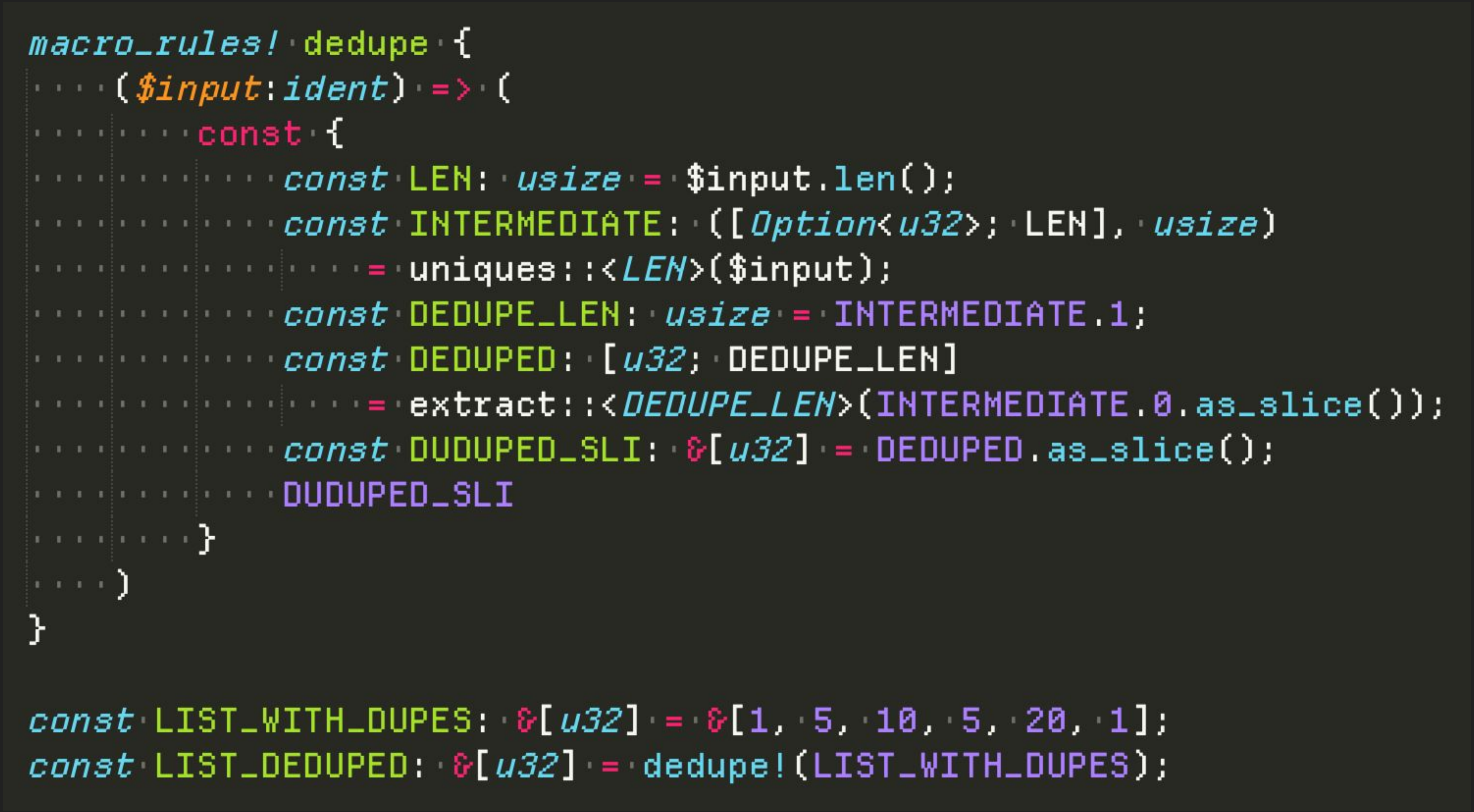


```
const fn extract<const M: usize>(sli: &[Option<u32>])
    -> [u32; M]
{
    let mut out = [0u32; M];
    let mut i = 0;
    while i < M {
        out[i] = sli[i].unwrap();
        i += 1;
    }
    while i < sli.len() {
        assert!(out[i].is_none());
    }
    out
}
```



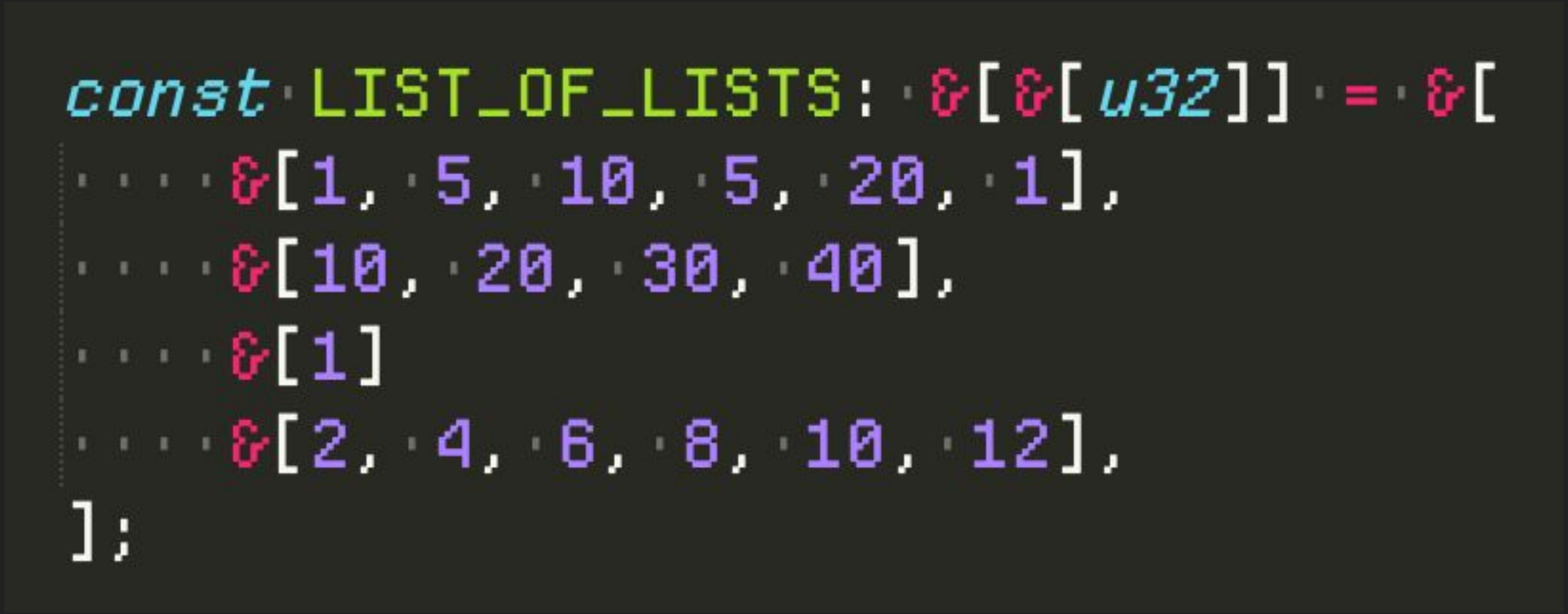
```
const LIST_WITH_DUPES: &[u32] = &[1, 5, 10, 5, 20, 1];
const LEN: usize = LIST_WITH_DUPES.len();
const INTERMEDIATE: ([Option<u32>; LEN], usize)
    = uniques::<LEN>(LIST_WITH_DUPES);
const DEDUPE_LEN: usize = INTERMEDIATE.1;
const DEDUPED: [u32; DEDUPE_LEN]
    = extract::<DEDUPE_LEN>(INTERMEDIATE.0.as_slice());
const DUDUPED_SLI: &[u32] = DEDUPED.as_slice();
```

that's a little verbose you say...



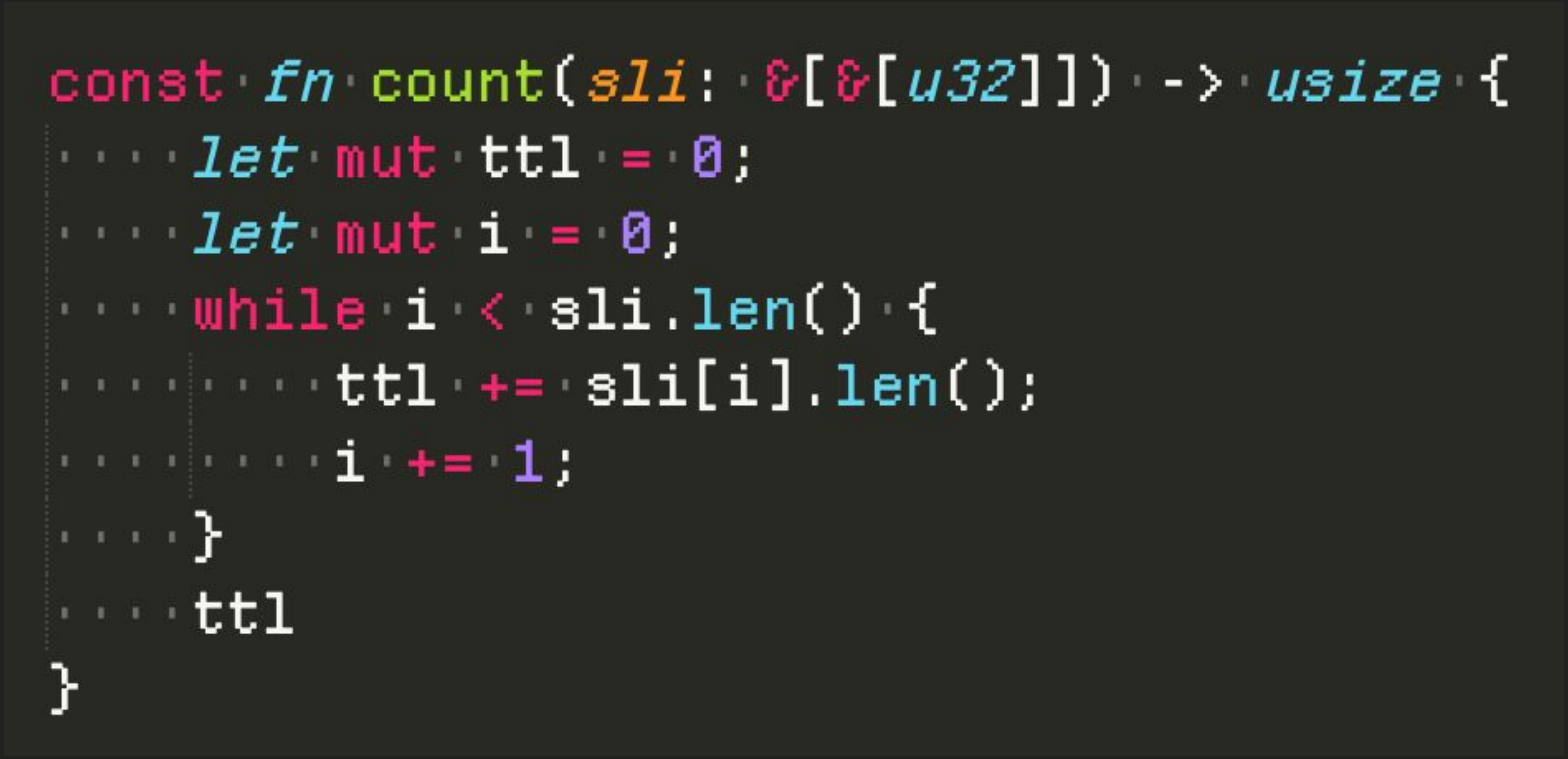
```rust
macro_rules! dedupe {
    ($input:ident) => (
        const {
            const LEN: usize = $input.len();
            const INTERMEDIATE: ([Option<u32>; LEN], usize)
                = uniques::<LEN>($input);
            const DEDUPE_LEN: usize = INTERMEDIATE.1;
            const DEDUPED: [u32; DEDUPE_LEN]
                = extract::<DEDUPE_LEN>(INTERMEDIATE.0.as_slice());
            const DUDUPED_SLI: &[u32] = DEDUPED.as_slice();
            DUDUPED_SLI
        }
    )
}

const LIST_WITH_DUPES: &[u32] = &[1, 5, 10, 5, 20, 1];
const LIST_DEDUPED: &[u32] = dedupe!(LIST_WITH_DUPES);
```

can we take this further?



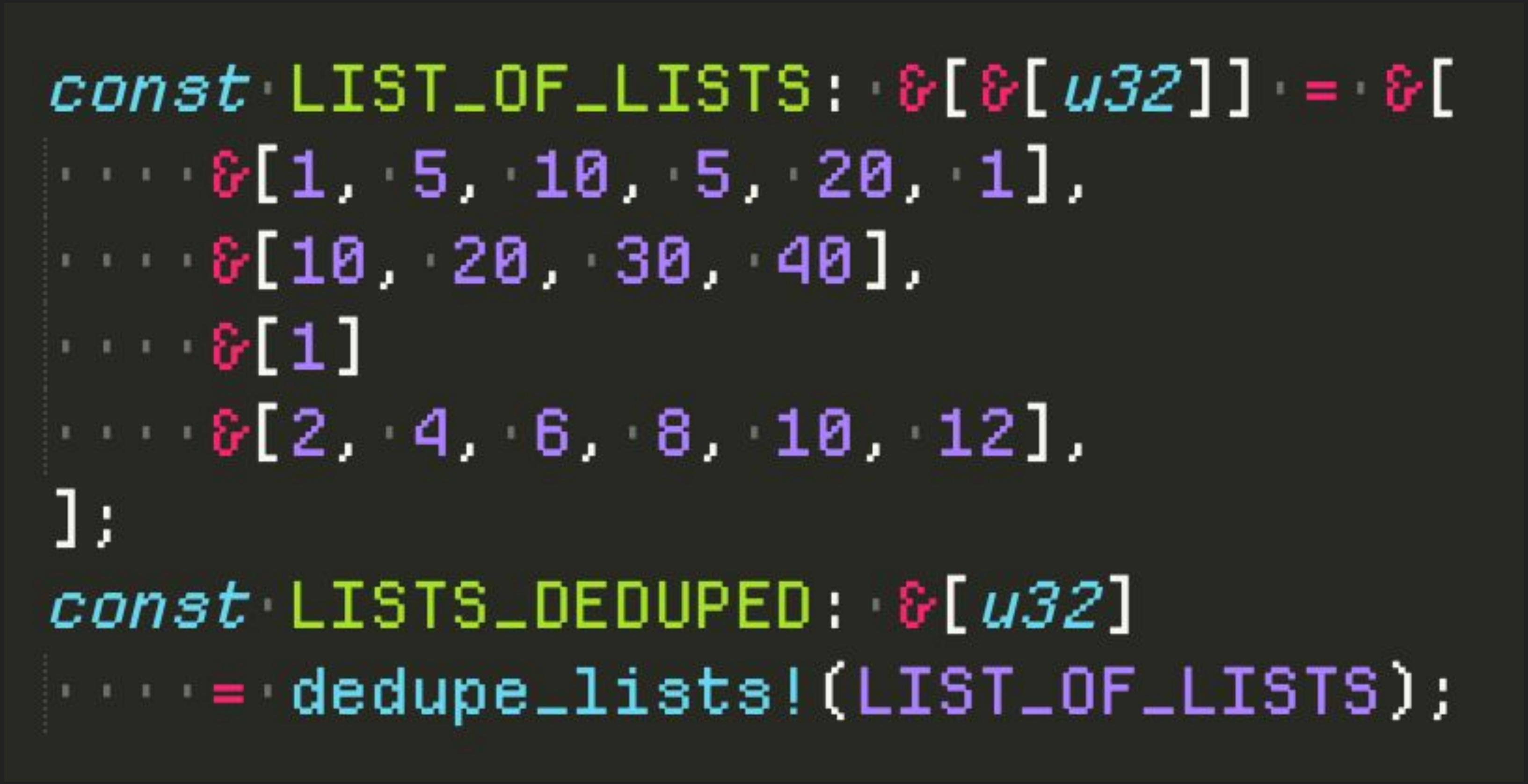
```rust
const LIST_OF_LISTS: &[&[u32]] = &[
    &[1, 5, 10, 5, 20, 1],
    &[10, 20, 30, 40],
    &[1]
    &[2, 4, 6, 8, 10, 12],
];
```

step one:
get the TOTAL size



```
const fn count(sli: &[&[u32]]) -> usize {
    let mut ttl = 0;
    let mut i = 0;
    while i < sli.len() {
        ttl += sli[i].len();
        i += 1;
    }
    ttl
}
```

step two: fill in `[Option<u32>; TOTAL]` and count uniques then we’re back to the same tricks



```rust
const LIST_OF_LISTS: &[&[u32]] = &[
    &[1, 5, 10, 5, 20, 1],
    &[10, 20, 30, 40],
    &[1]
    &[2, 4, 6, 8, 10, 12],
];
const LISTS_DEDUPED: &[u32]
    = dedupe_lists!(LIST_OF_LISTS);
```