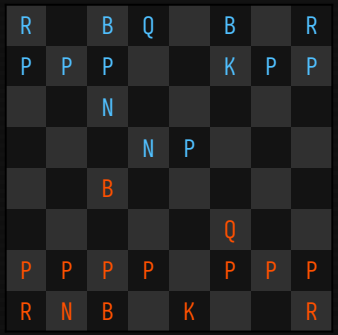
THIS SLIDE
INTENTIONALLY
LEFT BLANK.

WHAT IS A CHESS ENGINE?

A chess engine is a program that, given the state of a game of chess, attempts to both evaluäte that state, and recommend a sequence of moves.



→ ⟨ King to e6, +0.44 in White's favor ⟩

COMPONENTS OF A CHESS ENGINE

STATE REPRESENTATION	How is the chessboard represented in the program?
SEARCH	Constructing and pruning trees of future possibilities!
EVALUATION	How do we estimate the value of a chess position?

STATE REPRESENTATION

STATE REPRESENTATION

One must be able to

- represent the present state of the game
- query legal moves
- apply transitions caused by moves
- determine when & how a game has ended

This list is highly non-exhaustive.

STATE REPRESENTATION: TYPES

We use enums to represent colors, pieces, ranks, files...

```
#[derive(...)]
#[repr(u8)]
enum PieceType {
  Pawn   = 0,
  Knight = 1,
  Bishop = 2,
  Rook   = 3,
  Queen  = 4,
  King   = 5,
}
```

STATE REPRESENTATION: TYPES

We use enums to represent colors, pieces, ranks, files...

By doing this, we benefit manifold:

RECTITUDE Precisely specifying the valid values for a type eliminates swathes of bugs.

```
Rank::Third ≠ File::C ≠ PieceType::Knight
```

CONCISION Methods on enums are a joy to use.

```
Square::F2.relative_to(Color::Black) == Square::F7
```

ALACRITY The more constraints we have on our data, the more we can optimise.

STATE REPRESENTATION: INDEXING

Suppose we have a type for White *vs* Black like so:

```
enum Color {
  White = 0,
  Black = 1,
}
```

There are many places where we have something like

```
// occupancy : [u64; 2]
// side : Color
occupancy[side as usize] & ...
```

STATE REPRESENTATION: INDEXING

Typing out `usize` every time is annoying, and we'd like to avoid bounds checks.
Fortunately, we can implement the trait `std::ops::Index`!

```
impl<T> std::ops::Index<Color> for [T; 2] {
  type Output = T;

  fn index(&self, idx : Color) -> &Self::Output {
    unsafe { self.get_unchecked(idx as usize) }
  }
}
// ...and analogously for std::ops::IndexMut.
```

Typically, LLVM can be relied upon to elide bounds checks.
Avoid this pattern unless profiling reveals that elision has failed.

STATE REPRESENTATION: INDEXING

With this, we can now write

```
occupancy[side] & ...
```

without worrying about bounds checks or `usize` casts.

STATE REPRESENTATION: MOVES

What is a move? In chess, a source square, a destination square, and optionally a promotion type.

```
struct Move {
  src : Square,
  dst : Square,
  promotion : PieceType,
}
```

We can do better. There are only 64 squares, and 4 promotions, and $64 \times 64 \times 4 = 16384$ fits in two bytes, not three!

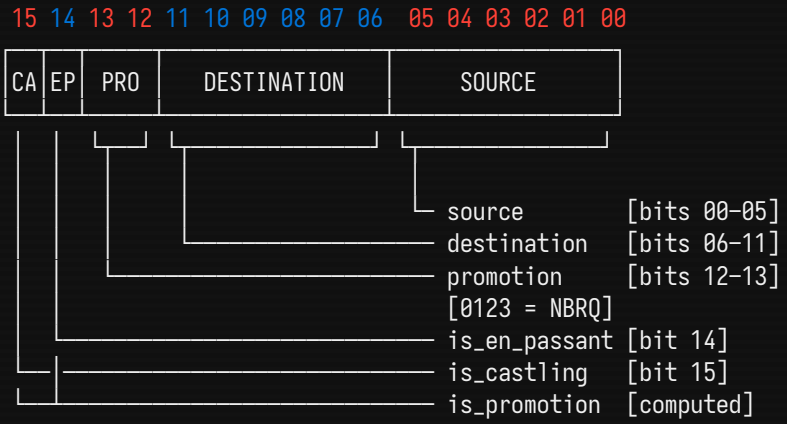
```
struct Move {
    data : u16,
}
```

STATE REPRESENTATION: MOVES

```
fn with_promo(from : Square, to : Square, promo : PieceType) -> Self {
  let promo = u16::from(promo.inner()).wrapping_sub(1) & PROMO_MASK;
  let data = u16::from(from)
    | (u16::from(to) << TO_SHIFT)
    | (promo << PROMO_SHIFT)
    | MoveFlags::Promo as u16;
  // SAFETY: data is always OR-ed with MoveFlags::promo,
  // and so is always non-zero.
  let data = unsafe { NonZeroU16::new_unchecked(data) };
  Self { data }
}
```

STATE REPRESENTATION: MOVES

What's in a move?



STATE REPRESENTATION: NULL MOVES

We often need to represent a “null move”.

Most chess engines encode the nullmove as specific sort of Move, e.g. a1 → a1.*

We have a better option.

```
struct Move {
    data : NonZeroU16,
}

const { assert_eq!(size_of::<Move>(), size_of::<Option<Move>>()); }
```

*In our encoding, a1 → a1 is the all-zeros bit-pattern.

STATE REPRESENTATION: BITBOARDS

Bitboards are unsigned 64-bit integers that represent *sets of squares* on the chessboard.

$$\{A1, C1, D1\} \cong \ldots00000001101$$

Conveniently, 64-bits is the size of the machine word on modern hardware.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
08	09	10	11	12	13	14	15
00	01	02	03	04	05	06	07

STATE REPRESENTATION: BITBOARDS

Typically, one stores a chess position using 8 bitboards.

```
struct Position {
  colors : [Bitboard; 2], // occupancy of each color
  pieces : [Bitboard; 6], // occupancy of each piece-type
}
```

Then, a query like “how many unblocked pawns are there” is merely:

```
let pawns = colors[Color::White] & pieces[PieceType::Pawn];
let movable_pawns = pawns & !colors[Color::Black].south_one();
return pawns.count_ones();
```

EFFICIENTLY ITERATING A BITBOARD

Given a bitboard (a square-set), how would you efficiently loop over the squares present in the set?

THE BMI INSTRUCTION SET

TZCNT	counts the number of trailing zeros in a value.
BLSR	resets the least significant set bit of a value.
POPCOUNT	counts the number of set bits in a value.

```
   tzcnt(0b10110100) == 2
    blsr(0b10110100) == 0b10110000
popcount(0b10110100) == 4
```

TZCNT and BLSR can be paired to form efficient loops over set bits.

THE BMI INSTRUCTION SET

TZCNT and BLSR can be paired to form efficient loops over set bits.

```
impl Iterator for SquareIter {
  fn next(&mut self) -> Option<Square> {
    if self.value == 0 {
      None
    } else {
      let lsb = self.value.trailing_zeros() as u8; // TZCNT
      self.value &= self.value - 1;                // BLSR
      Square::new(lsb)
    }
  }
}
```

THE BMI INSTRUCTION SET

`TZCNT` counts the number of trailing zeros in a number.

`BLSR` resets the least significant set bit of a number.

Abstraction in hand, we can loop over square-sets with maximum efficiency!

```
let our_knights = board.pieces[Knight] & board.colors[side];
for src in our_knights {
    let moves = knight_attacks(src);
    for tgt in moves & !blockers {
        moves.push(Move::new(src, tgt));
    }
}
```

EVALUÄTION

EVALUÄTION

Modern chess engines use Efficiently Updatable Neural Networks (ИИUЭ).

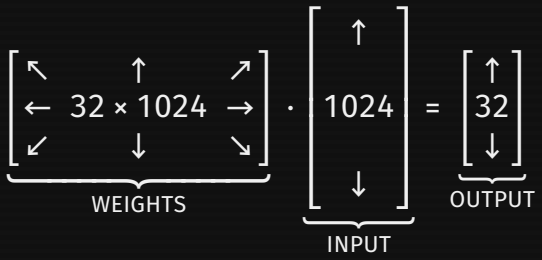
The input features of the network are entirely binary.

∴ Whenever a piece moves, we simply add or subtract columns of the embedding matrix to generate a new hidden state from an old one.

$$\begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}}_{\text{BOARD}} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \rightarrow \begin{bmatrix} \cdots & \cdots & e_{02} & \cdots & \cdots & \cdots \\ \cdots & \cdots & e_{12} & \cdots & \cdots & \cdots \\ \cdots & \cdots & e_{22} & \cdots & \cdots & \cdots \\ \cdots & \cdots & e_{32} & \cdots & \cdots & \cdots \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}}_{\text{BOARD}} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} + \begin{bmatrix} e_{02} \\ e_{12} \\ e_{22} \\ e_{32} \end{bmatrix}$$

INFERENCE OPTIMIZATION

The most expensive part of the engine is a 1024 → 32 fully-connected layer.



QUANTIZATION	We compress the weights and inputs to `i8`.
SPARSITY	We train the input activations to be mostly zero.
SIMD	We find and index the non-zero activations in parallel.

NETWORK SHARING

Many copies of an engine may live and play simultaneuously inside the same machine. Reading so many copies of the network causes cache pressure, so we map the same copy of the weights into all processes.

```
fn weights() -> &'static Network {
  static MAP : OnceLock<Mmap> = OnceLock::new();
  MAP.get_or_init(|| {
    if !network_path.exists() {
      decompress_network_into(&network_path)
    }
    unsafe { Mmap::map(network_path) }
  }) // transmutation and inter-process coördination elided.
}
```

SEARCH

SAFELY REMOVING BOUNDS-CHECKS

TASK Sort a list of moves by quality, best-to-worst.

```
moves.sort_by_key(|m| {
  (exchange_winning(m), history(m)) // (bool, i32)
});
```

REFINEMENT A We usually only need the first 1-2 elements of the sorted list.

REFINEMENT B The sorting key is dominated by exchange_winning.

REFINEMENT C exchange_winning is very costly.

SAFELY REMOVING BOUNDS-CHECKS

```
// each time around the loop, select the maximal element
while let Some((idx, max)) = moves.iter()
    .enumerate()
    .max_by_key(|(_, m)| history(m)) {
  // `max` is only the max-history move, prior.
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  moves.swap(0, idx); // :(
  moves = &mut moves[1..];
}
```

Frustratingly, `moves.swap(0, idx)` emits a bounds-check for `idx`.

SAFELY REMOVING BOUNDS-CHECKS

Ideally, we'd just write through our `max` reference – we could just `mem::swap` it with element zero.

```
while let Some(max) = moves.iter_mut().max_by_key(history) {
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  if done() { break; }
  std::mem::swap(&mut moves[0], max);
  //             ^^^^^^^^^^^^^ fails borrowchk
  moves = &mut moves[1..];
}
```

This doesn't work – `max` is uniquely borrowing `moves`, so `&mut moves[0]` fails borrowchk.

CELL

A mutable memory location.

```
#[repr(transparent)]
pub struct Cell<T : ?Sized> {
  value : UnsafeCell<T>,
}

impl<T> Cell<T> {
  pub fn new(value : T) -> Cell<T>;
  pub fn get(&self) -> T where T : Copy;
  pub fn set(&self, value : T);
  //         ^^^^^ a shared borrow!
}
```

THE SLICE-CELL TRICK

```
//  &mut [T] → &Cell<[T]> → &[Cell<T>]
let mut moves = Cell::from_mut(&mut moves).as_slice_of_cells();

while let Some(max) = moves.iter().max_by_key(|m| history(m.get())) {
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  if done() { break; }
  Cell::swap(&moves[0], max);
  moves = &moves[1..];
}
```

Because we can make multiple *shared* borrows of `moves`, all is well in the world, and this routine emits no bound-checks.

THE TRACK! MACRO

A lot goes on inside a chess engine*!*

```
// what's this value on average?
if track!(cheap_heuristic(...)) > 42 {
  // how often does this fire?
  if track!(expensive_heuristic()) {
    return;
  }
}
```

THE TRACK! MACRO

With macros, one can trace and aggregate statistics in an *ad-hoc* manner.

```
macro_rules! track {
  ($name:expr; $v:expr) => {{
    #[distributed_slice(TRACKED_VALUES)]
    static ENTRY : TrackedValue = TrackedValue::new(const { $name });
    let value = $v;
    ENTRY.record(value as i64);
    value
  }};
  ($v:expr) => {{ track!(stringify!($v); $v) }};
}
```

DIGRESSION: DISTRIBUTED_SLICE

linkme provides the #[distributed_slice] macro, which is *incredible*.

```
#[distributed_slice]
static CONSTANTS : [i32];

#[distributed_slice(CONSTANTS)] // foo.rs
static A : i32 = 45;
#[distributed_slice(CONSTANTS)] // bar.rs
static B : i32 = 20;
#[distributed_slice(CONSTANTS)] // baz.rs
static C : i32 = 12;

// main.rs
println!("{:?}", CONSTANTS); // → [20, 45, 12]
```

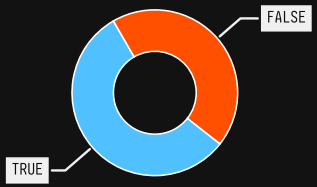
THE TRACK! MACRO

The track! macro behaves just like the identity function!

```
if track!(cheap_heuristic(...)) > 42 {
  if track!(expensive_heuristic()) {
    return;
  }
}
```

RFP

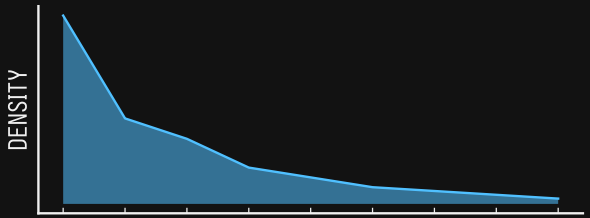
N = 18 964 497 μ = 0.6



DEPTH

N = 18 837 802 μ = 2.8 |μ| = 2.8

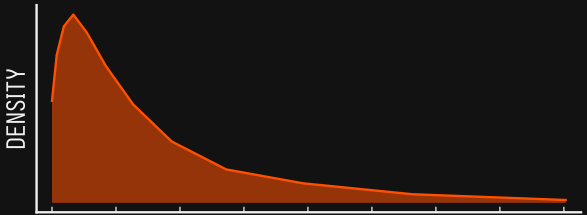
σ = 2.3 LO = 1 HI = 22



(EVAL - BETA) / 3

N = 10 635 066 μ = 201.2 |μ| = 201.2

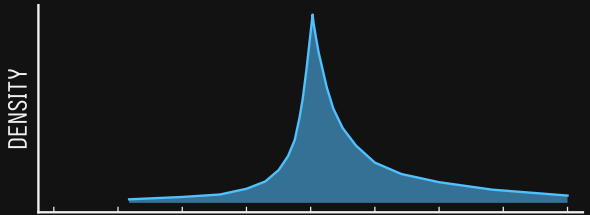
σ = 199.5 LO = 0 HI = 5658



EVAL

N = 18 194 503 μ = 437.1 |μ| = 709.3

σ = 1548.5 LO = -5916 HI = 32398



THIS SLIDE
INTENTIONALLY
LEFT BLANK.

TYPE-BASED ALIASING

or the lack thereof

In Rust, memory is a sequence of bytes, not of objects.

The following is perfectly legal in Rust:

```
let x : u32 = 0x1234_5678;
let y       = unsafe { &*(&raw const x).cast::<[u16; 2]>() };
assert_eq!(*y, [0x5678, 0x1234]);
```

This isn't the case in C or C++, where the *strict aliasing rule* applies.

```
const uint32_t x = 0x1234'5678;
const auto     y = (array<uint16_t, 2>*) &x;
// this cast is okay ↑
//       ↓ but this dereference is UB!
assert((*y)[0] == 0x5678 && (*y)[1] == 0x1234);
```

INCLUDE-BYTES

The weights of an ИИUƎ are a necessary part of the engine, and it makes little sense to distribute one *sans*-network.

```
/// The embedded neural network parameters.
static ИИUƎ : &[u8] = include_bytes!("../../weights.nnue.zst");
```

We can also save on recalculating the huge slider attack-tables.

```
// SAFETY: All bitpatterns of SquareSet are valid.
pub static DIAG_ATTACKS: [[SquareSet; 512]; 64] =
    unsafe { transmute(*include_bytes!("embeds/diag_attacks.bin")) };
pub static ORTH_ATTACKS: [[SquareSet; 4096]; 64] =
    unsafe { transmute(*include_bytes!("embeds/orth_attacks.bin")) };
```

SPARSE INDEX EXTRACTION

GOAL Write indexes of all non-zero blocks* into `nnz`.

$$\text{INPUT} = [\underbrace{0,0,0,0}_{0},\overbrace{0,11,17,0}^{1},\underbrace{0,0,0,0}_{2},\overbrace{0,0,0,5}^{3},\underbrace{13,37,0,0}_{4},\cdots]$$

In INPUT, blocks 0 and 2 are all-zeroes, while blocks 1, 3, and 4 contain at least one non-zero value.

As such, we fill `nonzero_list` with the values $[1,3,4,\cdots]$

*We use 4-element block-sparsity to make extraction of nonzero_list much faster, and to make the list itself shorter.

SPARSE INDEX EXTRACTION

GOAL Write indexes of all non-zero blocks into nonzero_list.

```
let mut nnz_count = 0;
for (chunk_idx, chunk) in input.chunks_exact(LANES).enumerate() {
  // get a mask of the non-zero blocks of 4.
  let mut nnz_mask = simd::nonzero_mask(simd::trans_u8_i32(chunk));
  let offsets = simd::load(&NNZ_TABLE.table[nnz_mask]);
  // precomputed array of bit indexes ^^^^^^^^^^^^^^^
  // transforms 0b10001101 -> [0, 2, 3, 7, ..., ..., ..., ...]
  let indexes = simd::add(simd::splat(chunk_idx * LANES), offsets);
  simd::store(&mut nonzero_list[nnz_count], indexes);
  nnz_count += u32::count_ones(nnz_mask);
}
```

MOVE-GENERATION: TABLE-LOOKUP

With bitboards, calculating the moves of a piece can be trivial.

```
static KNIGHT_ATTACKS : [SquareSet; 64] = const {
  let mut attacks = [SquareSet::EMPTY; 64];

  for sq in SquareSet::ALL {
    attacks[sq] = knight_attacks_slow(sq);
  }

  attacks
};
```

MOVE-GENERATION: SLIDING ATTACKS

Bishops, rooks, and queens move along rays, which can be obstructed.

```
                                ↓ ↓ ↓ ↓
fn bishop_attacks(from : Square, occupied : SquareSet) -> SquareSet {
  /* ??? */                     ↑ ↑ ↑ ↑
}
```

Thankfully, only the area of the board covered by the rays of the piece matters for the attack-set. For rooks, there are only 12 relevant “blockers”.

2^{12} patterns × 64 squares × 8 bytes-per-bitboard = 2MB.

LOOKUP TABLE FEASIBLE

MOVE-GENERATION: SLIDING ATTACK LOOKUPS

PROBLEM Given a bitboard and a mask, compute an index in 0..4096.

MOVE-GENERATION: SLIDING ATTACK LOOKUPS

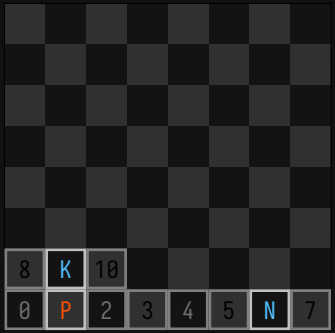
PROBLEM Given a bitboard and a mask, compute an index in 0..4096.

0x4020110A000A1120

||

0xADE704180420EF97

→

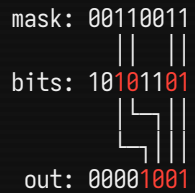


0x242

PEXT

The PEXT instruction takes a bitset and a mask, and transfers the masked bits to contiguous low order bit positions in the output.

```
pext(10101101, 00110011) == 00001001
```



MAGIC BITBOARDS

The problem of mapping occupancy-masks to table indexes can also be solved by *perfect hashing*.

```
pub struct MagicEntry {
  mask  : SquareSet,
  magic : u64,
}
pub fn diag_attacks(sq : Square, blockers : SquareSet) -> SquareSet {
  let entry    = &DIAG_TABLE[sq];
  let blockers = blockers & entry.mask;
  let data     = blockers.wrapping_mul(entry.magic);
  let idx      = (data >> (64 - 9)) as usize;
  DIAG_ATTACKS[sq][idx]
}
```