intro

are stack based vms really slower?

3K views • 4 months ago

why my scripting language is already faster than python

74K views • 5 months ago

```
var a = 1

print(a + { a += 1;  a })

// -> 4
```

it took me 1 month to fix this compiler bug

4.1K views • 2 months ago

```
today:
  tips for going fast
```

Advent of Code

{:year 2022}

--- Day 19: Not Enough Minerals ---

Your scans show that the lava did indeed form obsidian!

The wind has changed direction enough to stop sending lava droplets toward you, so you and the elephants exit the cave. As you do, you notice a collection of geodes around the pond. Perhaps you could use the obsidian to create some **geode-cracking robots** and break them open?

Advent of Code

{:year 2022}

--- Day 19: Not Enough Minerals ---

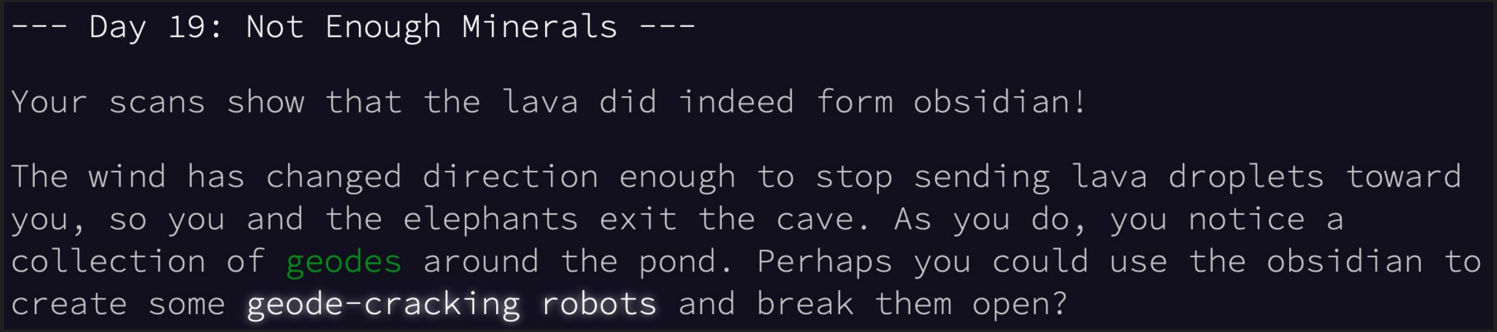
Your scans show that the lava did indeed form obsidian!

The wind has changed direction enough to stop sending lava droplets toward you, so you and the elephants exit the cave. As you do, you notice a collection of geodes around the pond. Perhaps you could use the obsidian to create some **geode-cracking robots** and break them open?

from: 139.2288 s

Advent of Code

{:year 2022}



--- Day 19: Not Enough Minerals ---

Your scans show that the lava did indeed form obsidian!

The wind has changed direction enough to stop sending lava droplets toward you, so you and the elephants exit the cave. As you do, you notice a collection of geodes around the pond. Perhaps you could use the obsidian to create some **geode-cracking robots** and break them open?

```
from:   139.2288 s
to:       0.0027 s
```

🍑

🍑, no

- multi-threading

🍑, no

- multi-threading
- SIMD

🍑, no

- multi-threading
- SIMD
- memory layout

🍑, no

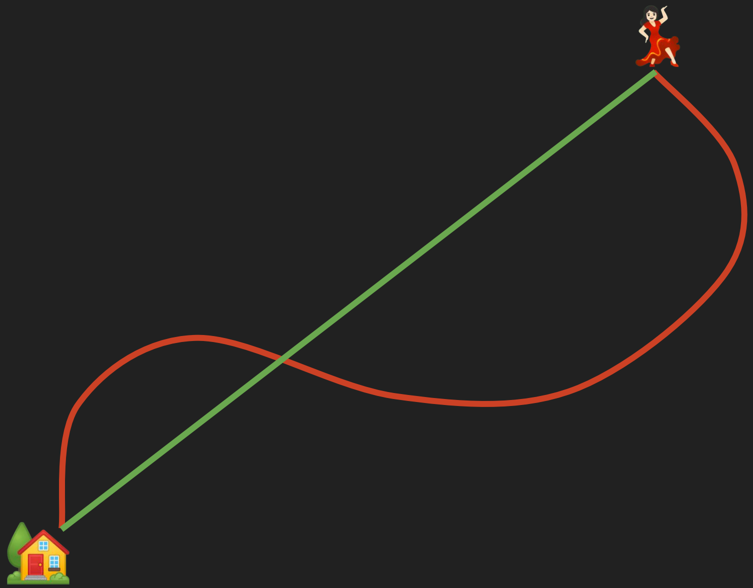
- multi-threading
- SIMD
- memory layout
- data dependencies

doing less

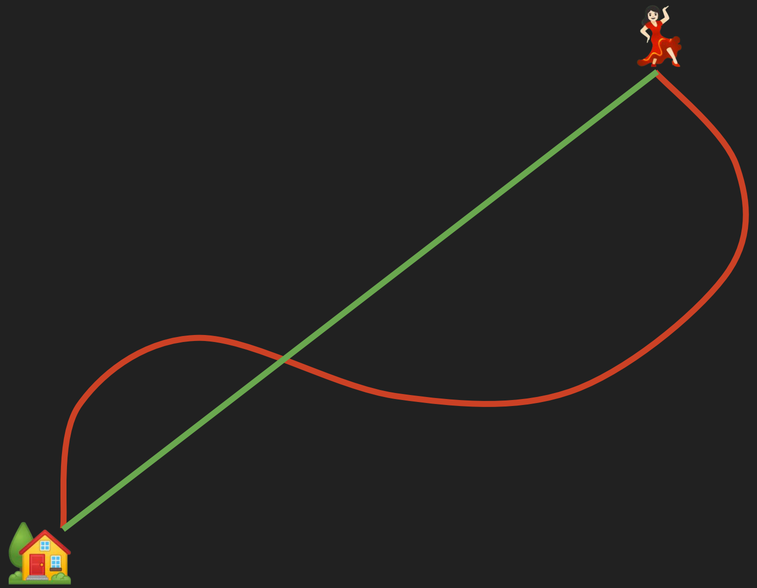
↓ t

$$\downarrow t = \frac{---}{v \uparrow}$$

$$\downarrow t = \frac{x \downarrow}{v \uparrow}$$



$$\downarrow t = \frac{x \downarrow}{v \uparrow}$$



$$\downarrow t = \frac{x \downarrow}{v \uparrow}$$

- multi-threading
- SIMD
- memory layout
- data dependencies

$$\downarrow t = \frac{x \downarrow}{v \uparrow}$$

- algorithms & data structures

- multi-threading
- SIMD
- memory layout
- data dependencies

$$\downarrow t = \frac{x \downarrow}{v \uparrow}$$

- algorithms & data structures
- domain knowledge

- multi-threading
- SIMD
- memory layout
- data dependencies

$$\downarrow t = \frac{x \downarrow}{v \uparrow}$$

- algorithms & data structures
- domain knowledge
- relaxing requirements

- multi-threading
- SIMD
- memory layout
- data dependencies

$$\downarrow t = \frac{x \downarrow}{v \uparrow}$$

- algorithms & data structures
- domain knowledge
- relaxing requirements

- multi-threading
- SIMD
- memory layout
- data dependencies

🚀 tip #1

first, try doing less

🚀 tip #1

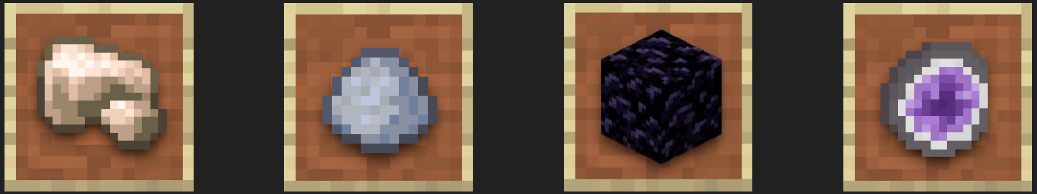
first, try doing less

Rust multi-threading code review

176K views • 1 year ago

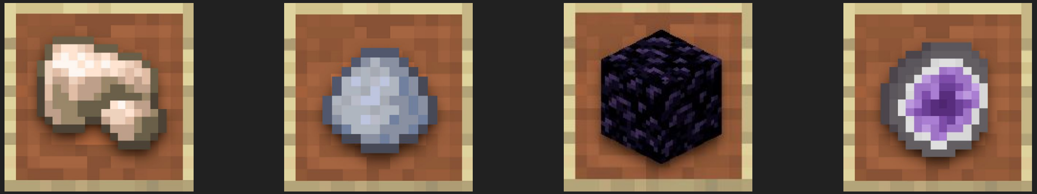
problem description

```
have:
 -   resources
```



have:
- resources

goal:
- maximize number of geodes



have:

- resources
- robots

goal:

- maximize number of geodes

have:
- resources
- robots

goal:
- maximize number of geodes

rules:
- robot -> 1 resource at end of turn

have:
- resources
- robots

goal:
- maximize number of geodes

rules:
- robot -> 1 resource at end of turn
- start with 1 ore robot

have:

- resources
- robots

goal:

- maximize number of geodes

rules:

- robot -> 1 resource at end of turn
- start with 1 ore robot
- can build up to 1 robot per turn

have:
- resources
- robots

goal:
- maximize number of geodes

rules:
- robot -> 1 resource at end of turn
- start with 1 ore robot
- can build up to 1 robot per turn

blueprint 1:

ore robot:	4 ore
clay robot:	2 ore
obsidian robot:	3 ore, 14 clay
geode robot:	2 ore, 7 obsidian

have:

- resources
- robots

goal:

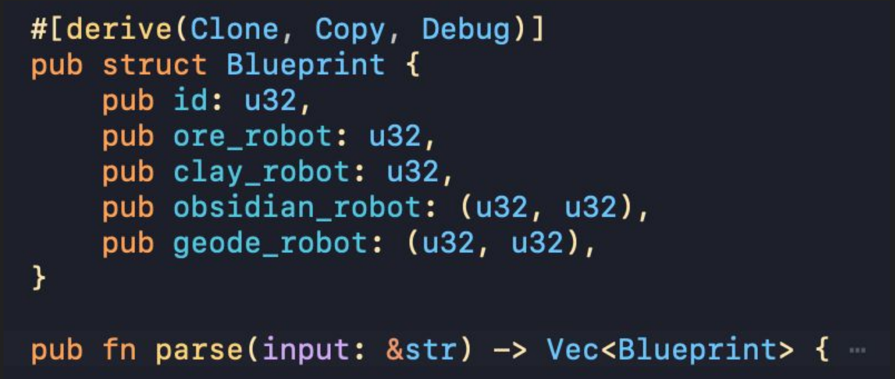
- maximize number of geodes

rules:

- robot -> 1 resource at end of turn
- start with 1 ore robot
- can build up to 1 robot per turn

```
blueprint 1:
  ore robot:         4 ore
  clay robot:        2 ore
  obsidian robot:    3 ore, 14 clay
  geode robot:       2 ore, 7 obsidian
```

solution?



```rust
#[derive(Clone, Copy, Debug)]
pub struct Blueprint {
    pub id: u32,
    pub ore_robot: u32,
    pub clay_robot: u32,
    pub obsidian_robot: (u32, u32),
    pub geode_robot: (u32, u32),
}

pub fn parse(input: &str) -> Vec<Blueprint> { ⋯
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct State {
    minute: u8,
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}
```

```rust
if state.can_build_geode_robot(bp) {
    state = state.build_geode_robot(bp);
}
```

efficient strategy?

1) geode robot is what we want,
   so if can build geode robot, do that.

```
1) geode robot is what we want,
   so if can build geode robot, do that.

2) otherwise, try to build obsidian robot.
   that will help us build geode robots.
```

```
1) geode robot is what we want,
   so if can build geode robot, do that.

2) otherwise, try to build obsidian robot.
   that will help us build geode robots.

3) otherwise, try to build a clay robot.
```

1) geode robot is what we want,
   so if can build geode robot, do that.

2) otherwise, try to build obsidian robot.
   that will help us build geode robots.

3) otherwise, try to build a clay robot.

4) otherwise, try to build an ore robot.

1) geode robot is what we want,
   so if can build geode robot, do that.

2) otherwise, try to build obsidian robot.
   that will help us build geode robots.

3) otherwise, try to build a clay robot.

4) otherwise, try to build an ore robot.

5) repeat until reached time limit.

```rust
pub fn solve(bp: &Blueprint, limit: u8) -> u32 {
    let mut state = State::new();
    for _ in 0..limit {
        if state.can_build_geode_robot(bp) {
            state = state.step().build_geode_robot(bp);
        }
        else if state.can_build_obsidian_robot(bp) {
            state = state.step().build_obsidian_robot(bp);
        }
        else if state.can_build_clay_robot(bp) {
            state = state.step().build_clay_robot(bp);
        }
        else if state.can_build_ore_robot(bp) {
            state = state.step().build_ore_robot(bp);
        }
        else {
            state = state.step();
        }
    }

    state.geode
}
```

no work 😔

no work 😔

problem: always builds robots (greedy).

no work 😔

problem: always builds robots (greedy).

need to account for future actions.

🚀 tip #2

start with an algorithm that is *definitely* correct

(even if it may be too slow)

have:

- resources
- robots

goal:

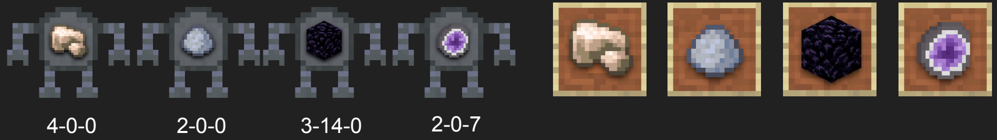
- maximize number of geodes

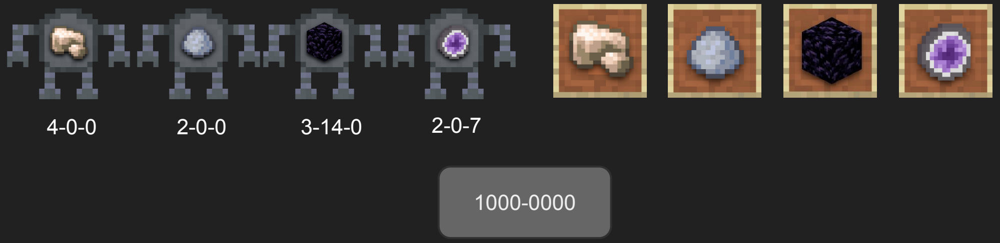
rules:

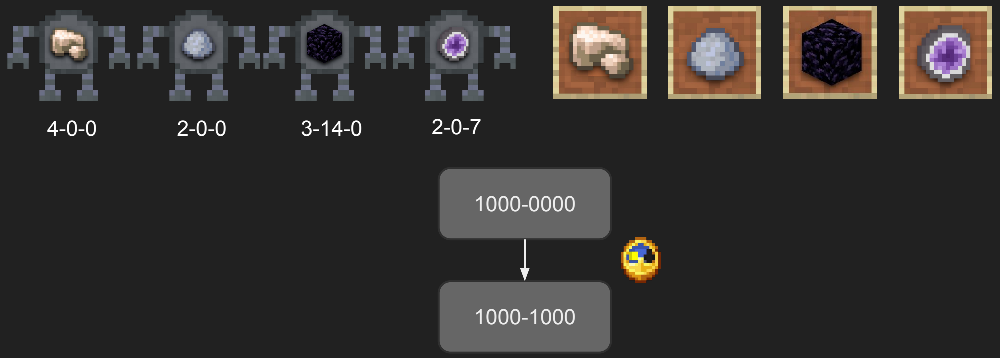
- robot -> 1 resource at end of turn
- start with 1 ore robot
- can build up to 1 robot per turn

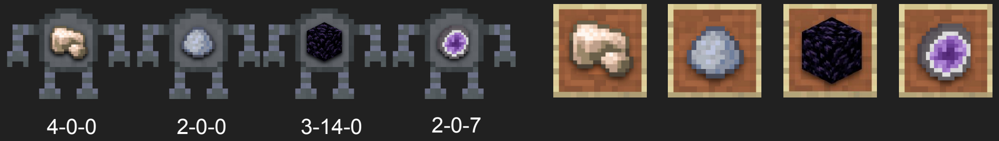
🤔 exercise:
what’s an algorithm to find the maximum number of geodes, that’s definitely correct.

brute force

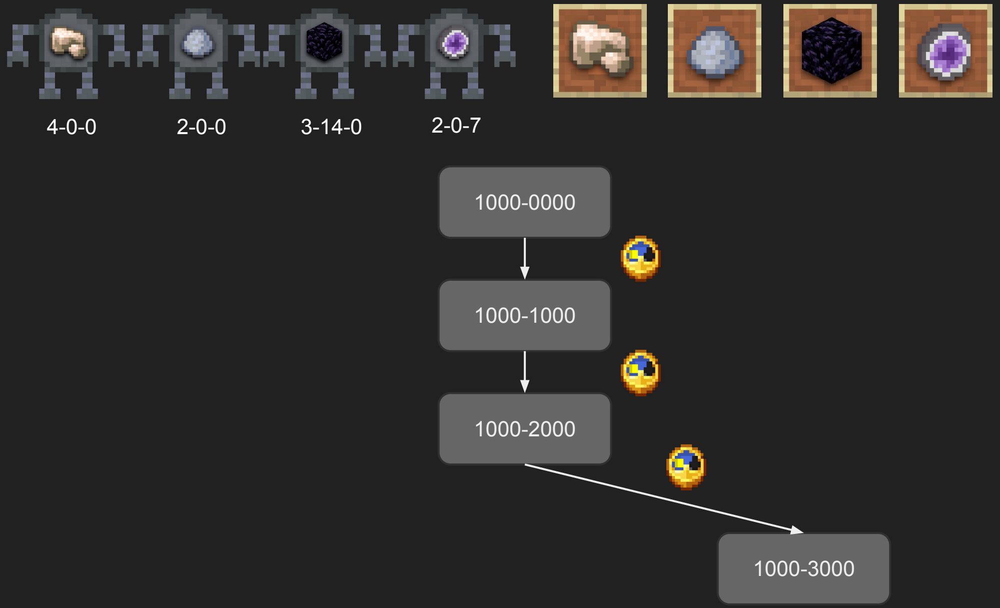

4-0-0
2-0-0
3-14-0
2-0-7

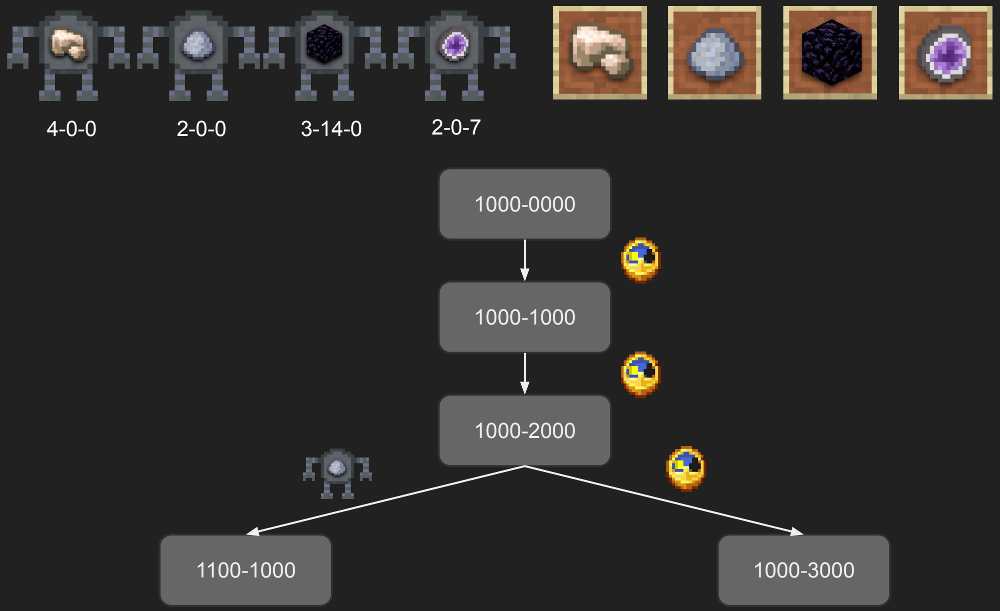

4-0-0
2-0-0
3-14-0
2-0-7
1000-0000

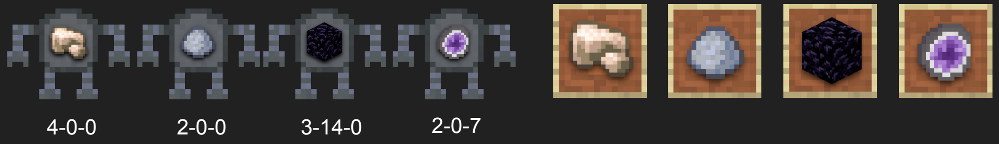

4-0-0
2-0-0
3-14-0
2-0-7
1000-0000
1000-1000

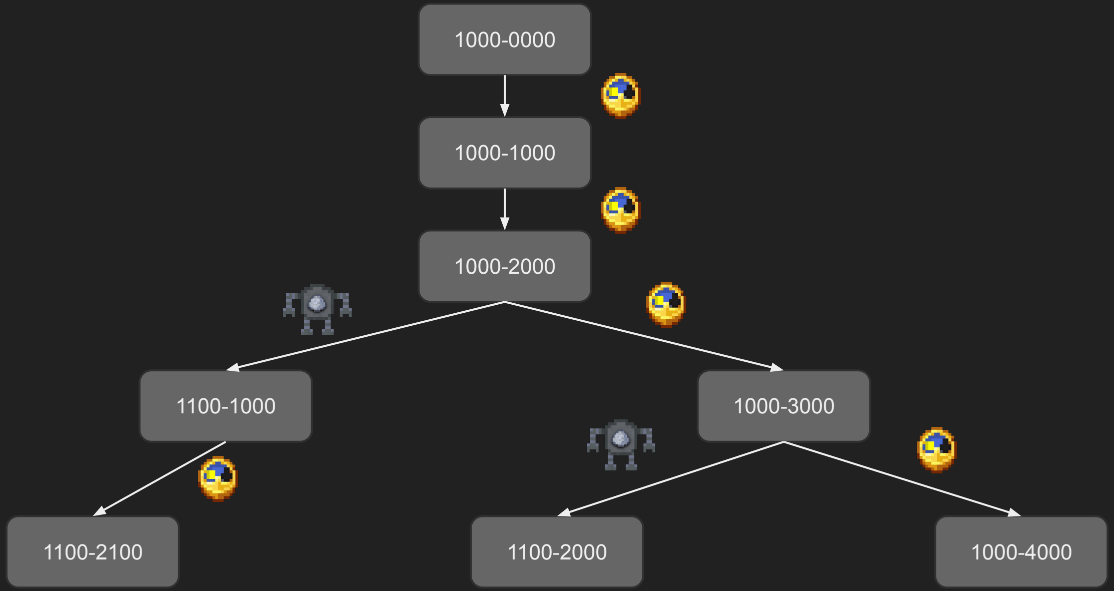


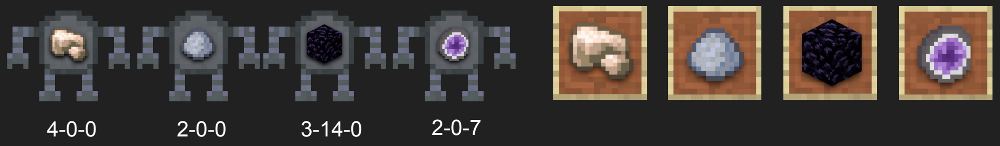
$$\begin{tikzcd}
\boxed{1000\text{-}0000} \arrow[d] \\
\boxed{1000\text{-}1000} \arrow[d] \\
\boxed{1000\text{-}2000}
\end{tikzcd}$$

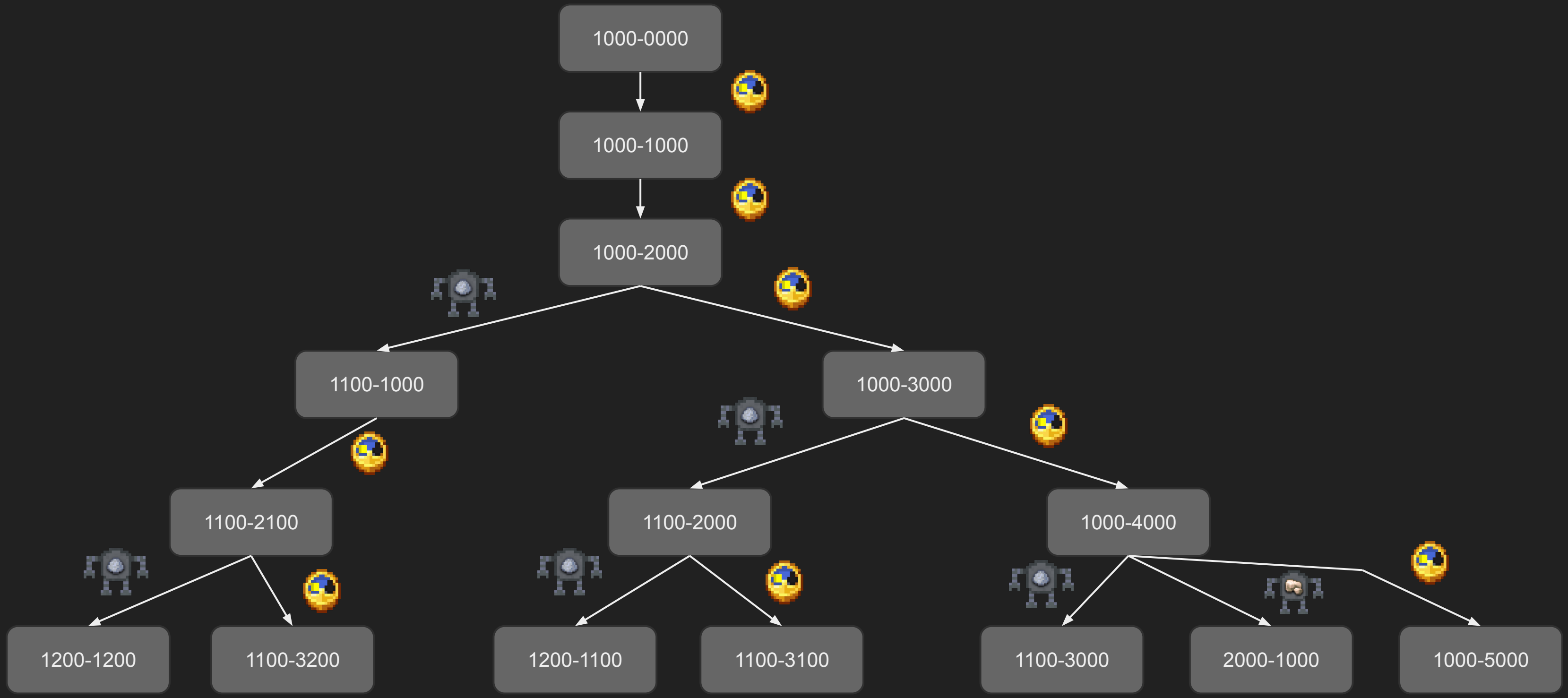

4-0-0
2-0-0
3-14-0
2-0-7
1000-0000
1000-1000
1000-2000
1000-3000


4-0-0
2-0-0
3-14-0
2-0-7
1000-0000
1000-1000
1000-2000
1100-1000
1000-3000


4-0-0
2-0-0
3-14-0
2-0-7


1000-0000
1000-1000
1000-2000
1100-1000
1000-3000
1100-2100
1100-2000
1000-4000


4-0-0
2-0-0
3-14-0
2-0-7


1000-0000
1000-1000
1000-2000
1100-1000
1000-3000
1100-2100
1100-2000
1000-4000
1200-1200
1100-3200
1200-1100
1100-3100
1100-3000
2000-1000
1000-5000

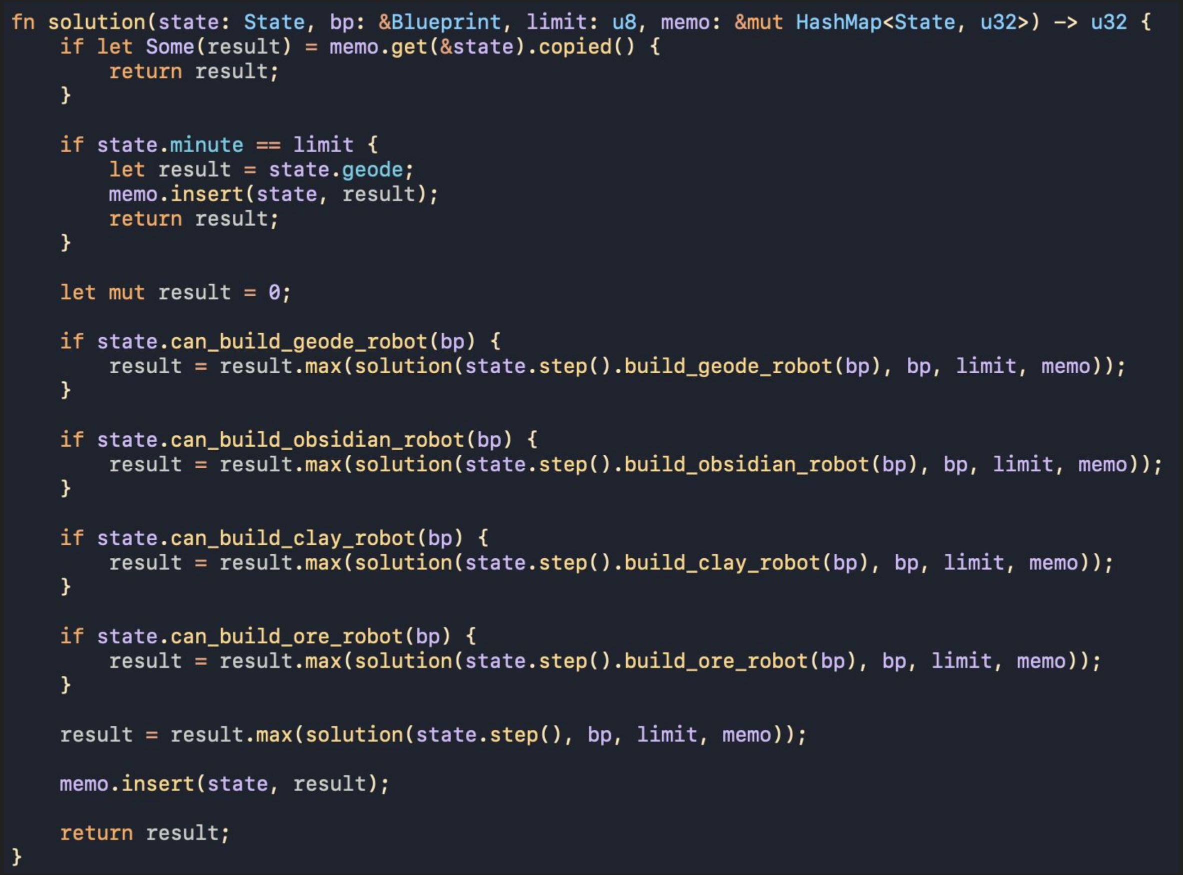
```rust
fn solution(state: State, bp: &Blueprint, limit: u8) -> u32 {
    if state.minute == limit {
        return state.geode;
    }

    let mut result = 0;

    if state.can_build_geode_robot(bp) {
        result = result.max(solution(state.step().build_geode_robot(bp), bp, limit));
    }

    if state.can_build_obsidian_robot(bp) {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit));
    }

    if state.can_build_clay_robot(bp) {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit));
    }

    if state.can_build_ore_robot(bp) {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit));
    }

    result = result.max(solution(state.step(), bp, limit));

    return result;
}
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8) -> u32 {
    if state.minute == limit {
        return state.geode;
    }

    let mut result = 0;

    if state.can_build_geode_robot(bp) {
        result = result.max(solution(state.step().build_geode_robot(bp), bp, limit));
    }

    if state.can_build_obsidian_robot(bp) {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit));
    }

    if state.can_build_clay_robot(bp) {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit));
    }

    if state.can_build_ore_robot(bp) {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit));
    }

    result = result.max(solution(state.step(), bp, limit));

    return result;
}
```

execution time: DNF 😔

```rust
fn solution(state: State, bp: &Blueprint, limit: u8) -> u32 {
    if state.minute == limit {
        return state.geode;
    }

    let mut result = 0;

    if state.can_build_geode_robot(bp) {
        result = result.max(solution(state.step().build_geode_robot(bp), bp, limit));
    }

    if state.can_build_obsidian_robot(bp) {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit));
    }

    if state.can_build_clay_robot(bp) {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit));
    }

    if state.can_build_ore_robot(bp) {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit));
    }

    result = result.max(solution(state.step(), bp, limit));

    return result;
}
```

🤔 exercise:
what’s a generic optimization we could try here?



```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<State, u32>) -> u32 {
    if let Some(result) = memo.get(&state).copied() {
        return result;
    }

    if state.minute == limit {
        let result = state.geode;
        memo.insert(state, result);
        return result;
    }

    let mut result = 0;

    if state.can_build_geode_robot(bp) {
        result = result.max(solution(state.step().build_geode_robot(bp), bp, limit, memo));
    }

    if state.can_build_obsidian_robot(bp) {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit, memo));
    }

    if state.can_build_clay_robot(bp) {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit, memo));
    }

    if state.can_build_ore_robot(bp) {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit, memo));
    }

    result = result.max(solution(state.step(), bp, limit, memo));

    memo.insert(state, result);

    return result;
}
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<State, u32>) -> u32 {
    if let Some(result) = memo.get(&state).copied() {
        return result;
    }

    if state.minute == limit {
        let result = state.geode;
        memo.insert(state, result);
        return result;
    }

    let mut result = 0;

    if state.can_build_geode_robot(bp) {
        result = result.max(solution(state.step().build_geode_robot(bp), bp, limit, memo));
    }

    if state.can_build_obsidian_robot(bp) {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit, memo));
    }

    if state.can_build_clay_robot(bp) {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit, memo));
    }

    if state.can_build_ore_robot(bp) {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit, memo));
    }

    result = result.max(solution(state.step(), bp, limit, memo));

    memo.insert(state, result);

    return result;
}
```

execution time: 140 s 🥳

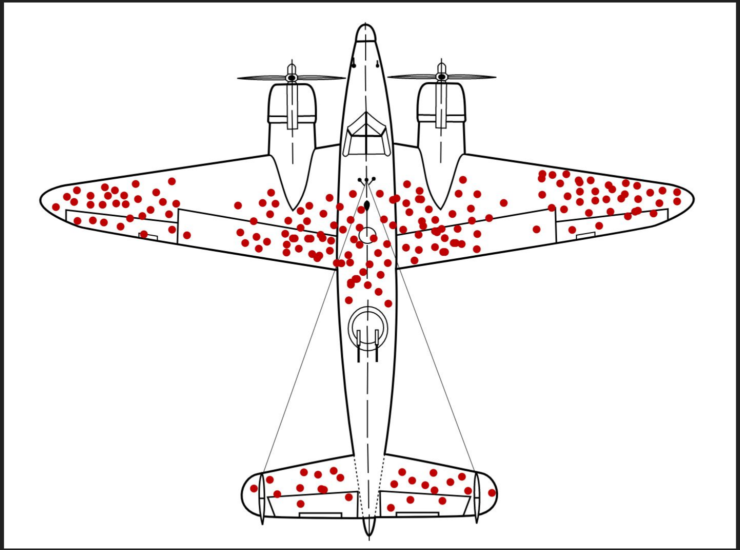
🚀 tip #3

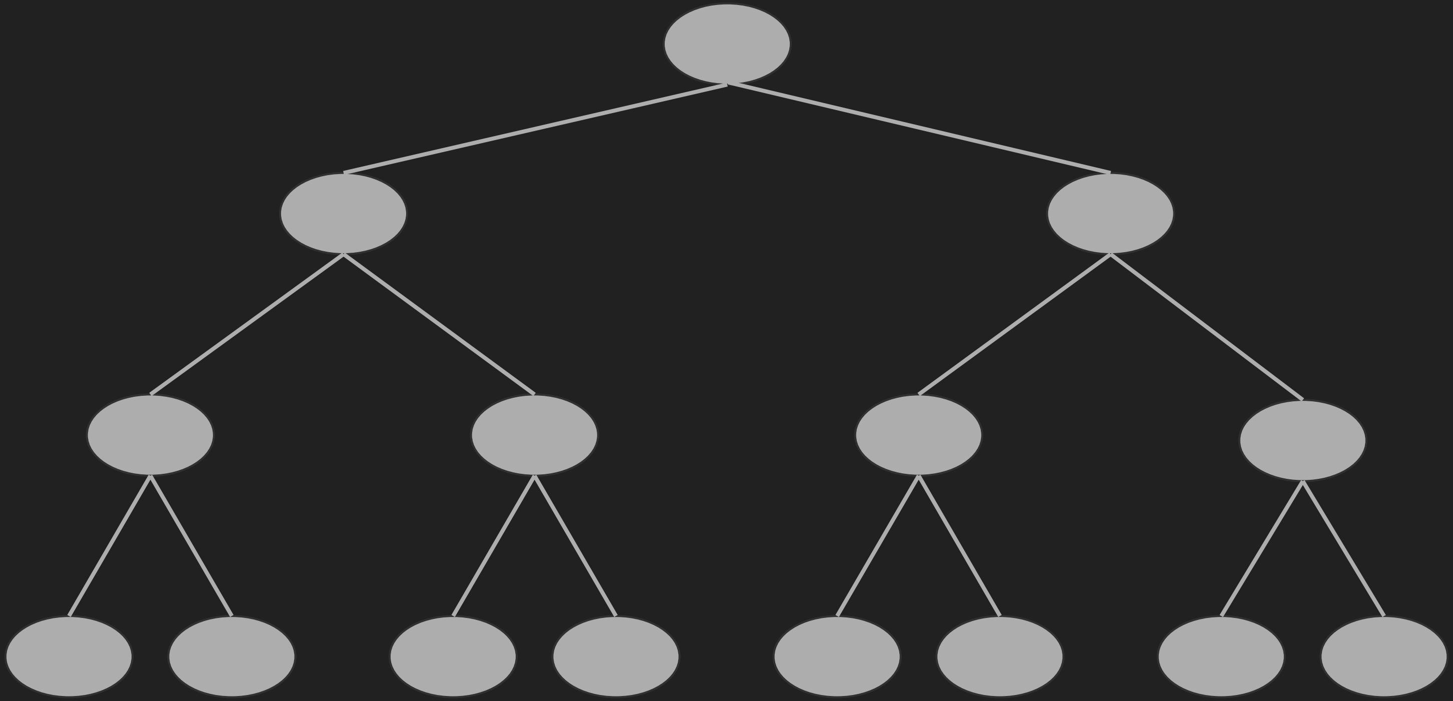
use caches to avoid doing *expensive* work multiple times

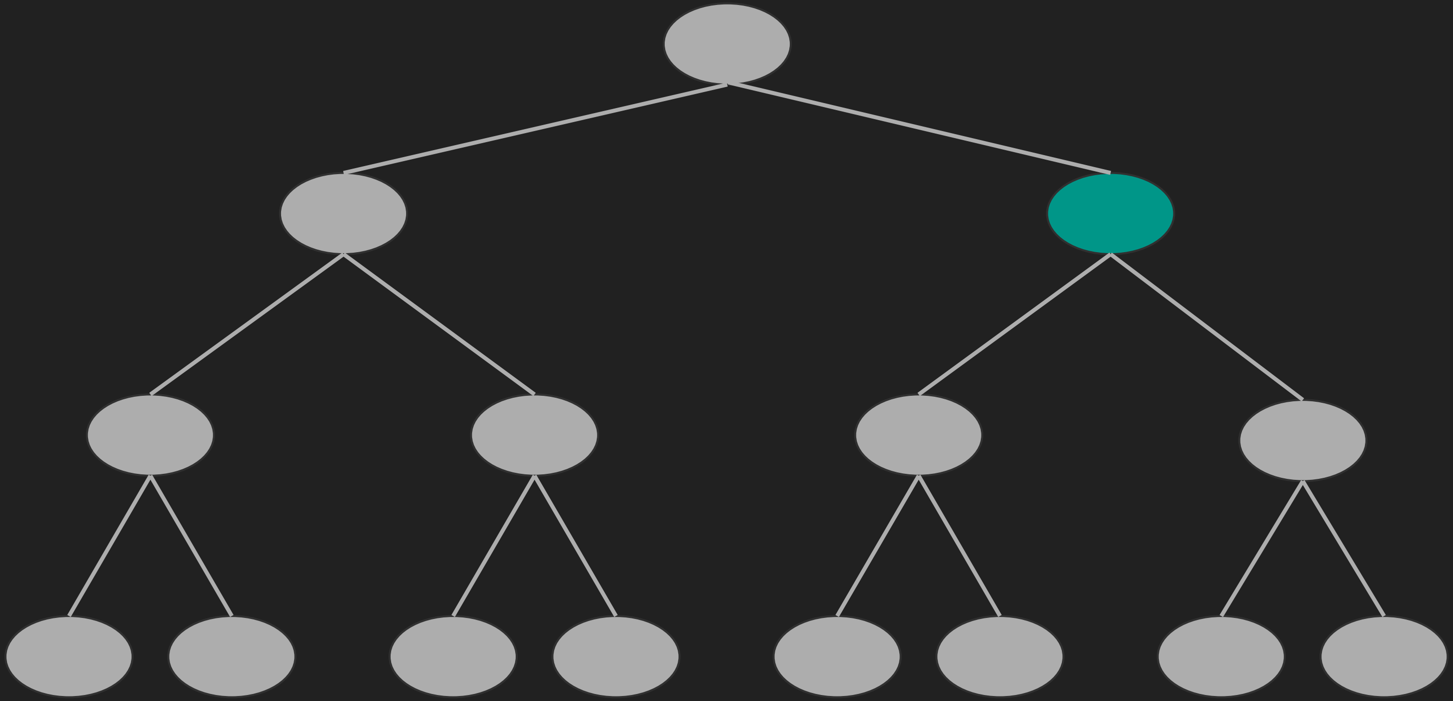
```rust
struct Stats {
    memo_refs: u64,
    memo_hits: u64,
    states_visited: u64,
}
```

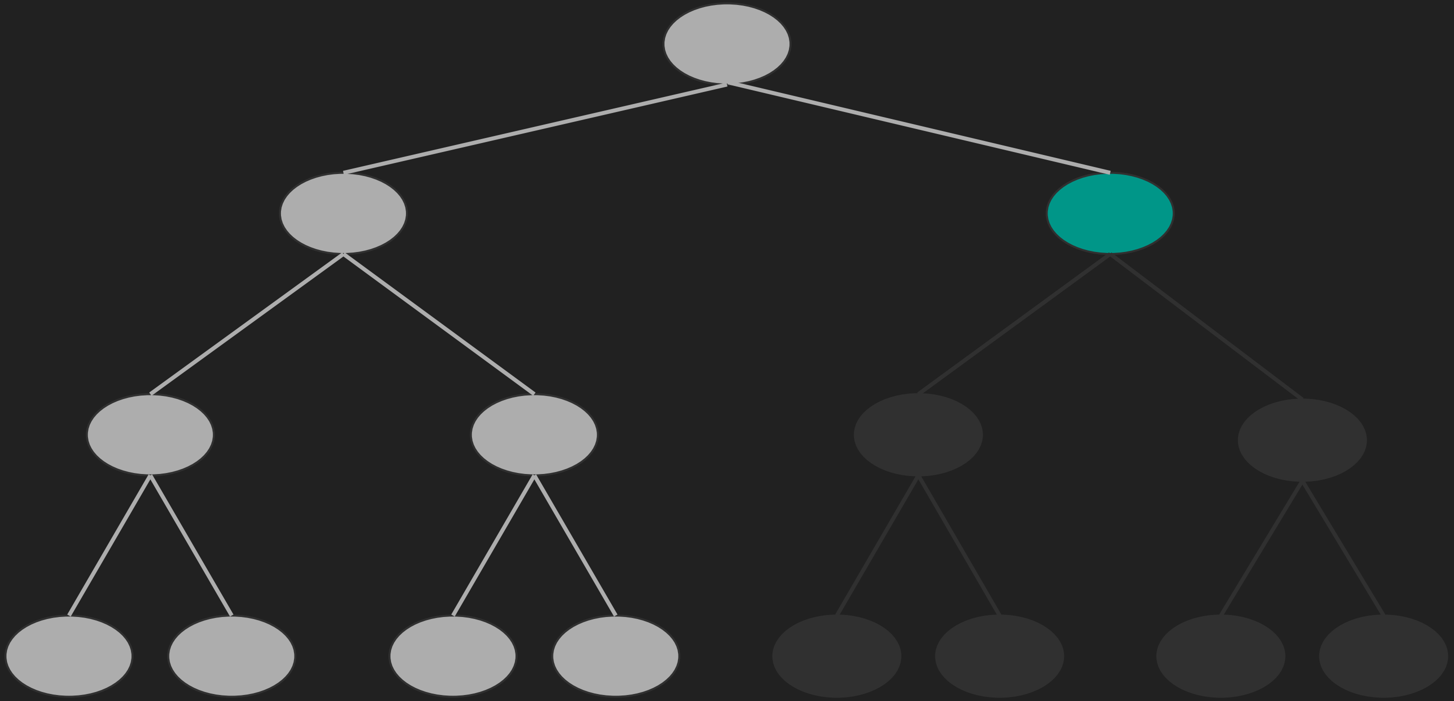
```
memo refs: 834,619,249
memo hits: 398,856,779 (48%)
memo refs: 834,619,249
memo hits: 398,856,779 (48%)
```

🤔 exercise:
does the cache only result in a 2x speedup?

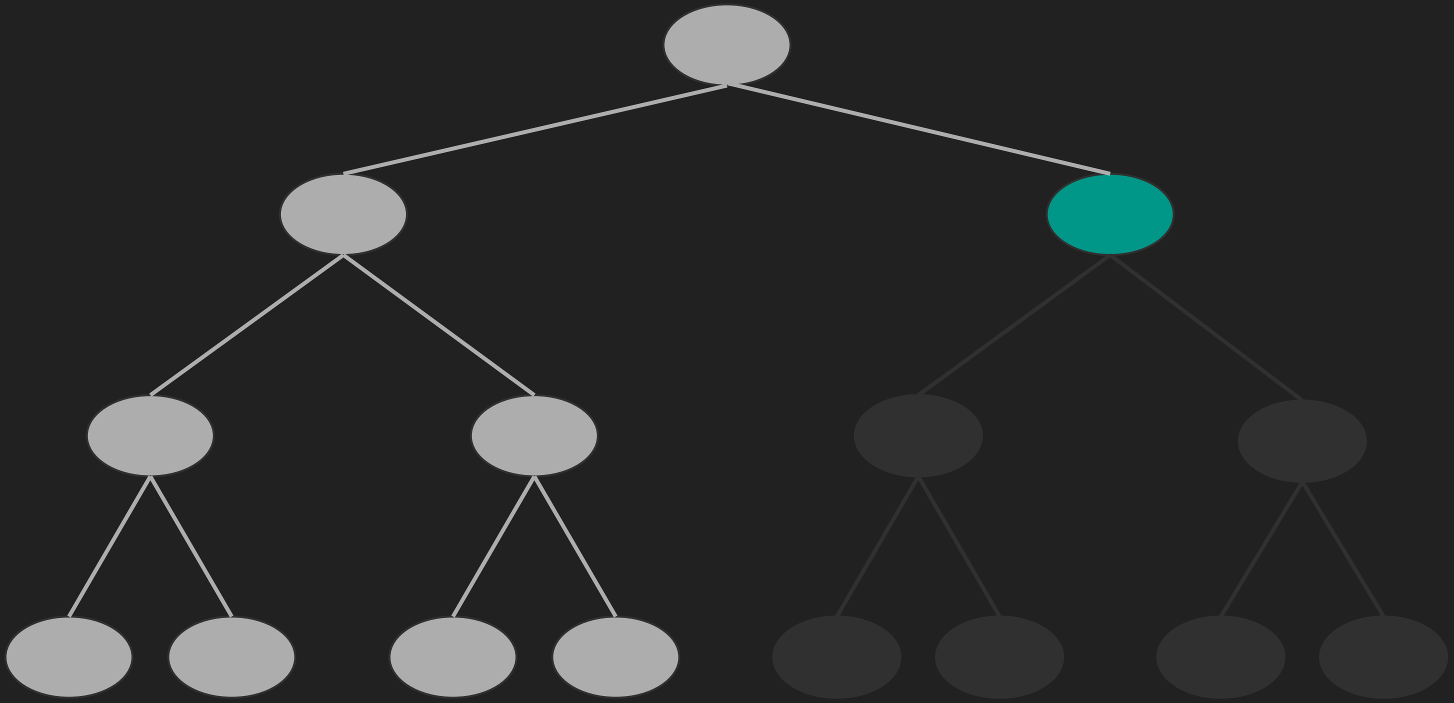


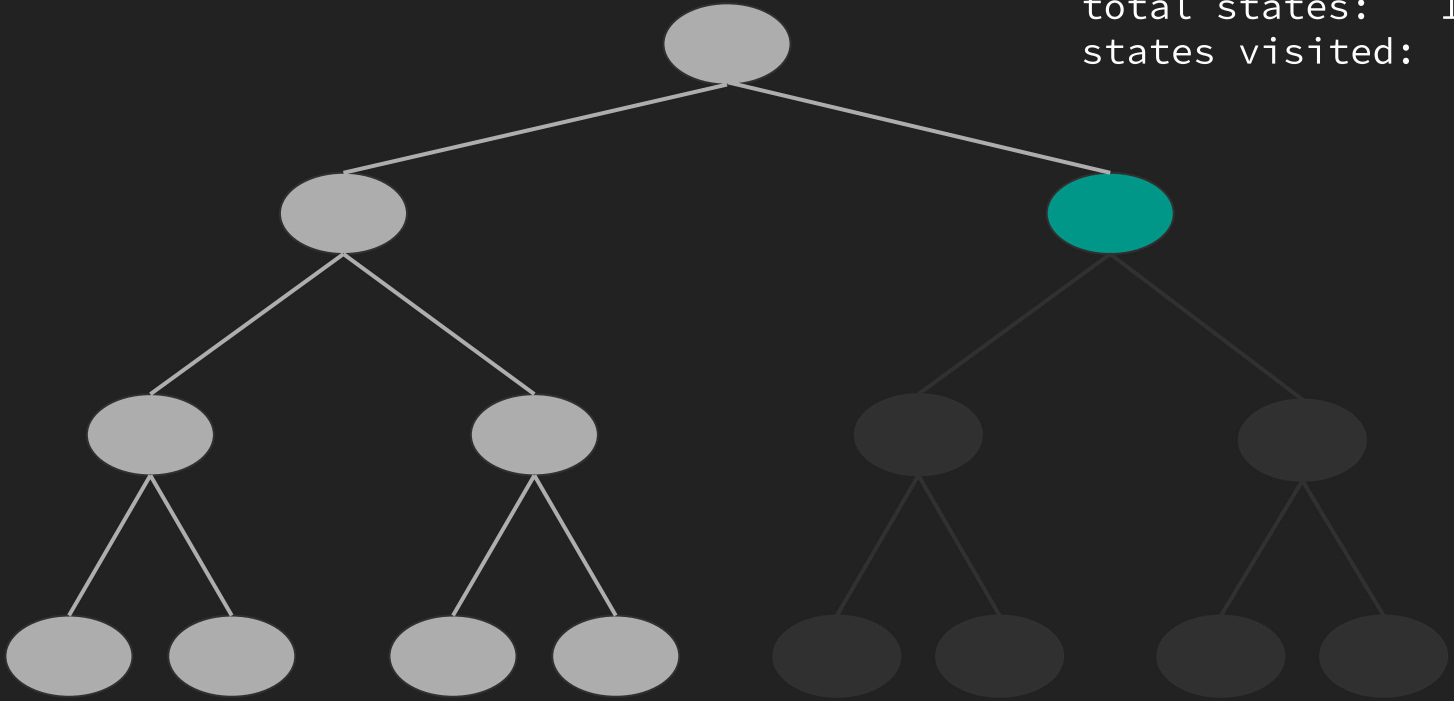






```
memo refs: 9
memo hits: 1 (11%)
```





```
memo refs: 9
memo hits: 1 (11%)

total states:   15
states visited:  9 (60%)
```

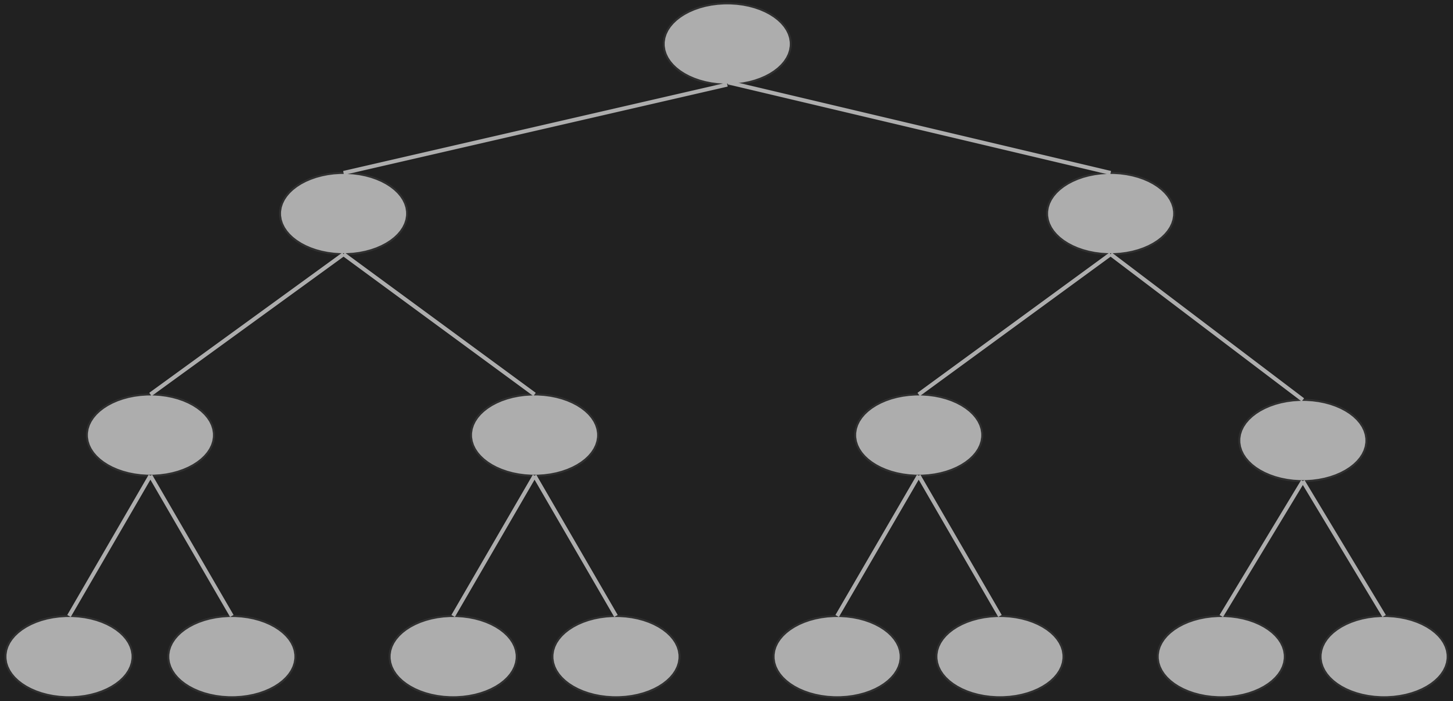
```
 memo refs:               834,619,249
 memo hits:               398,856,779 (48%)

total states:    434,570,542,645
states skipped:  433,735,923,396 (99.8%)
```

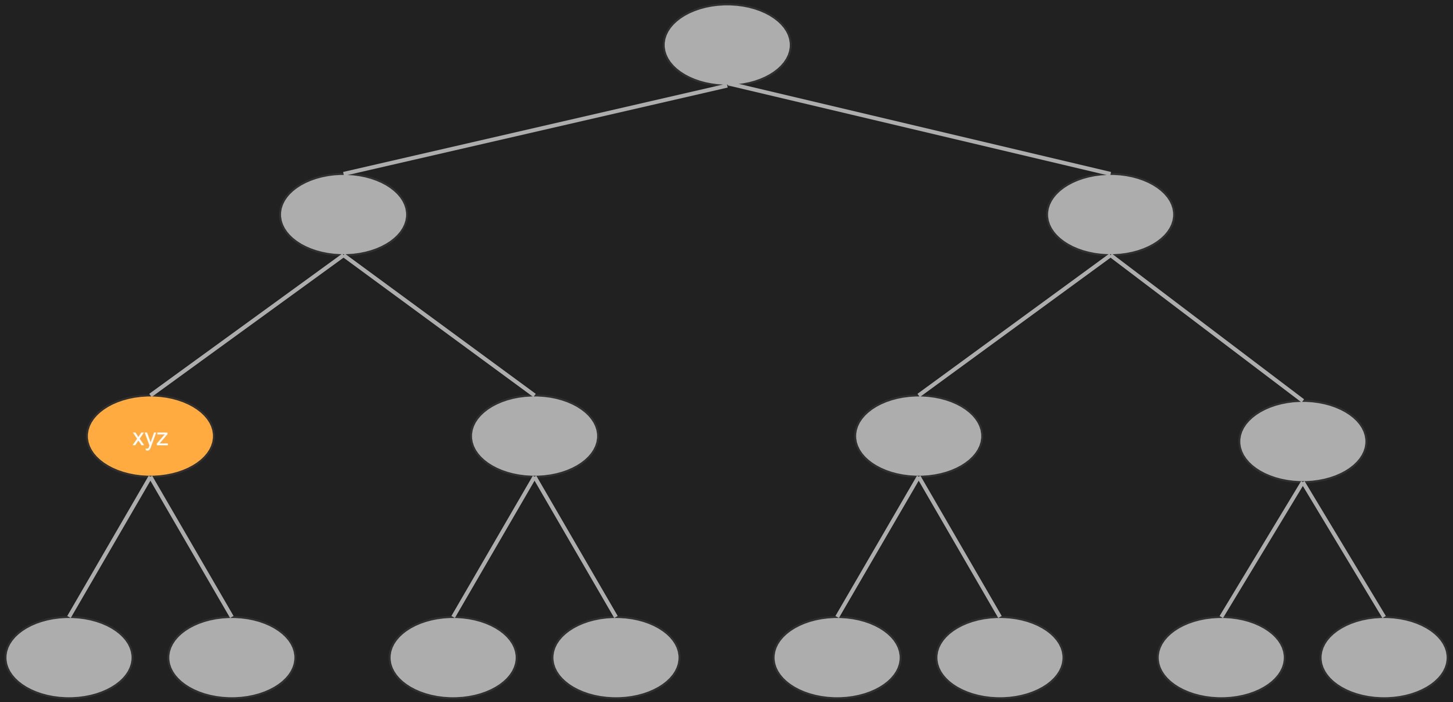
🚀 tip #4

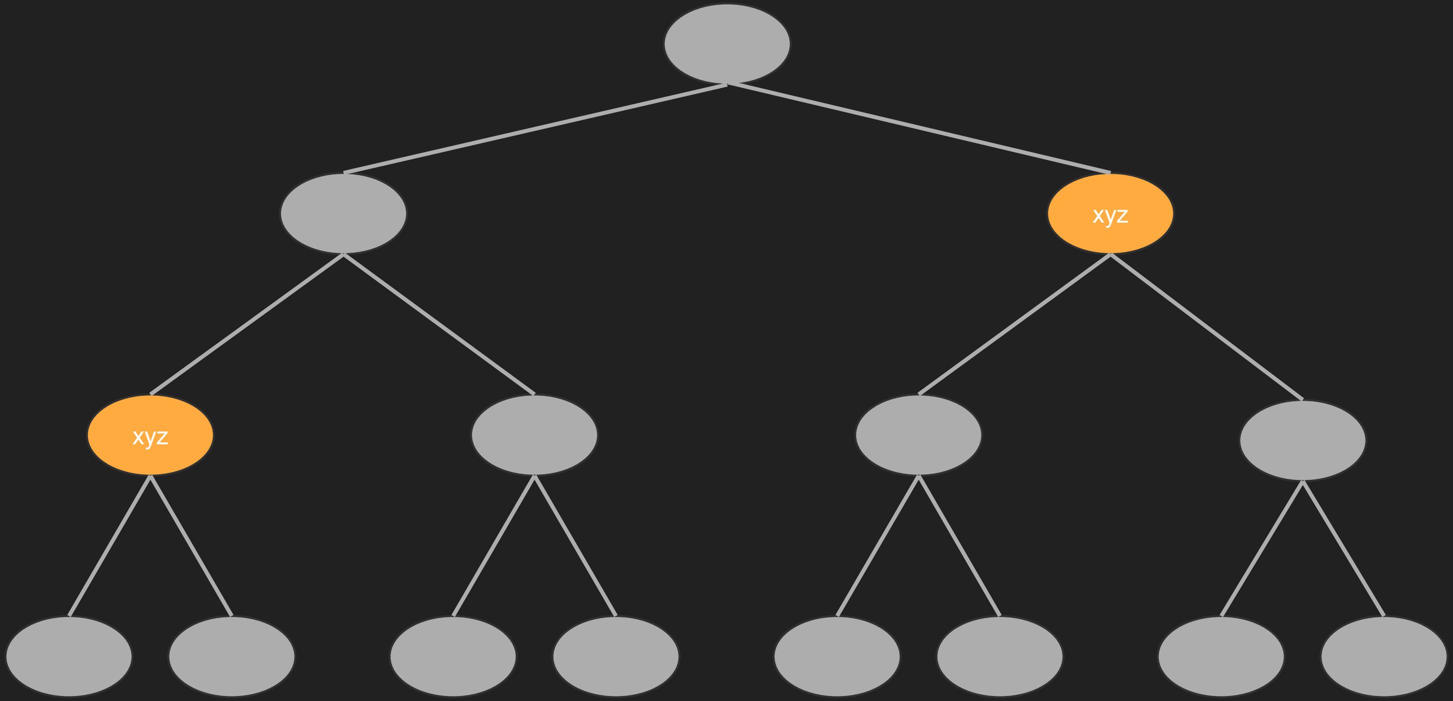
dynamic programming is just recursion + caching
CHANGE MY MIND
imgflip.com

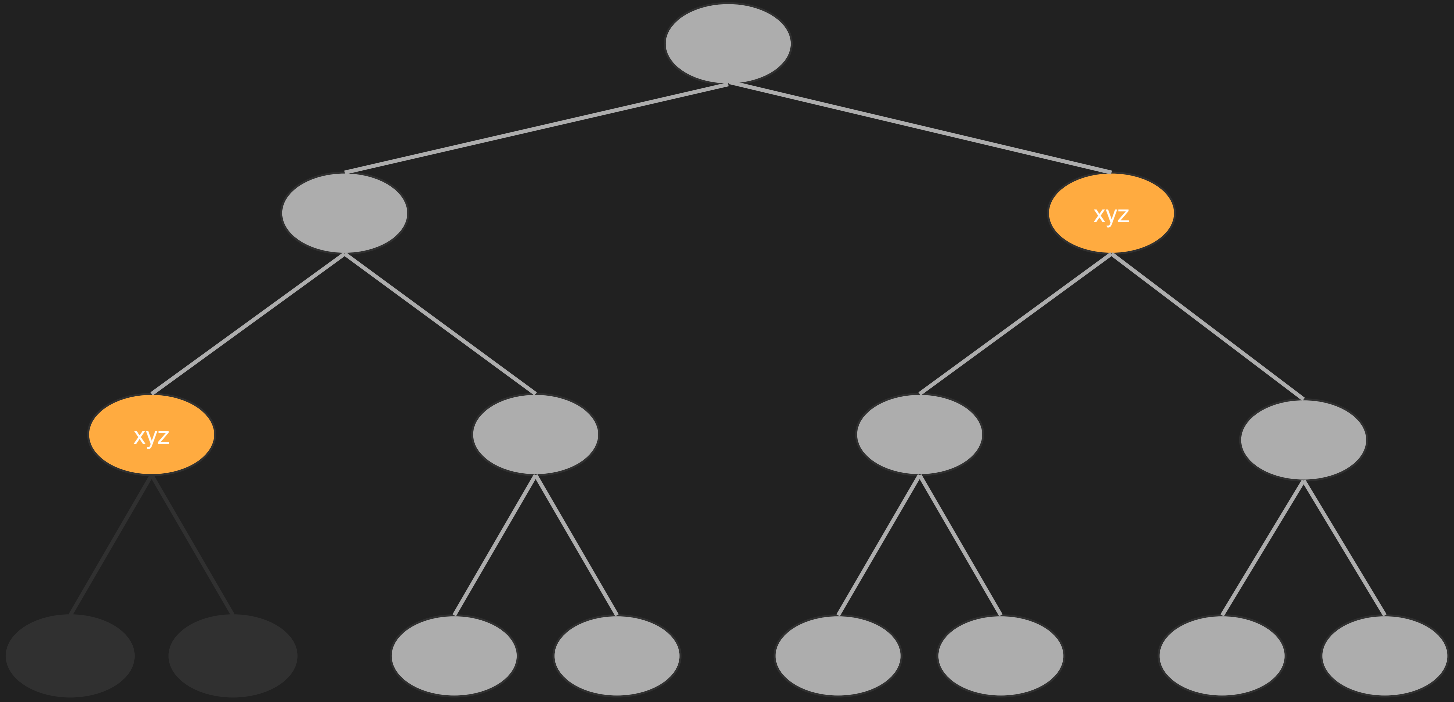
domain knowledge

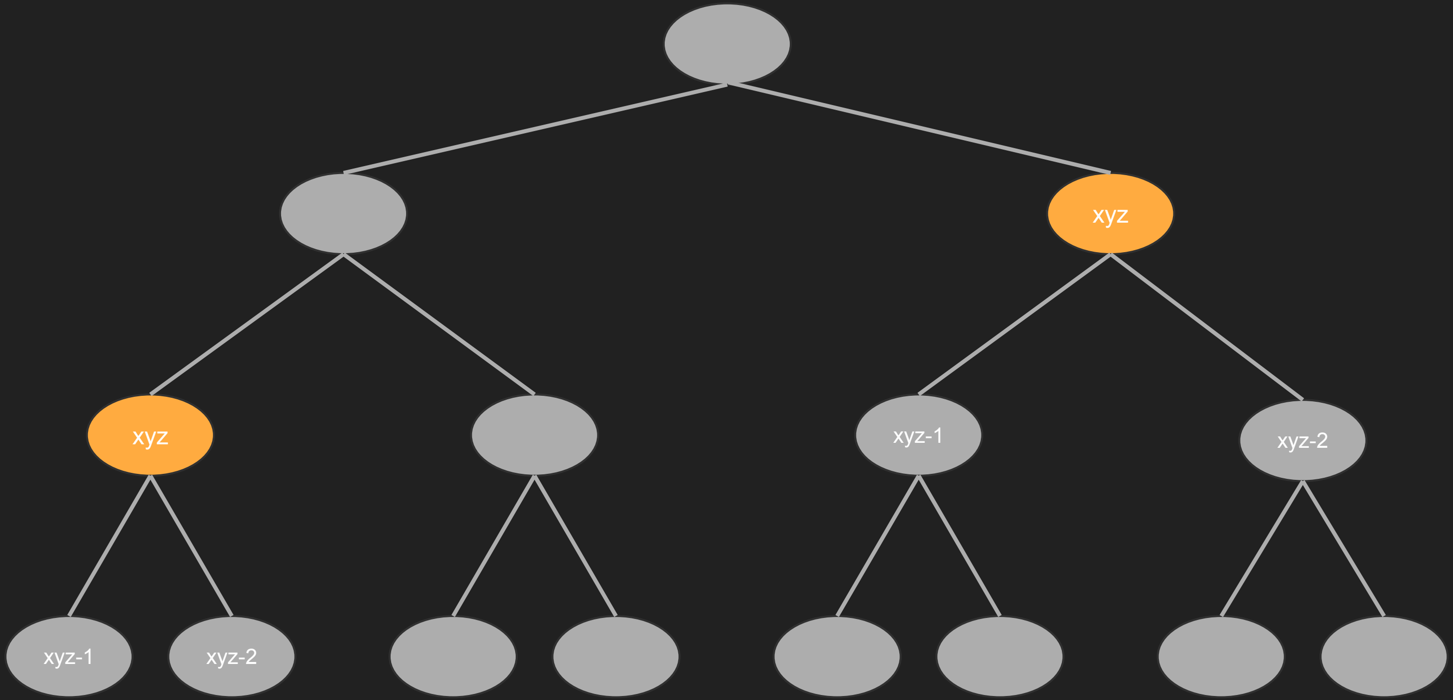
more time > less time




xyz


xyz
xyz


xyz
xyz


xyz
xyz
xyz-1
xyz-2
xyz-1
xyz-2

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}

#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct State {
    minute: u8,
    pack:   Pack,
}
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}

#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct State {
    minute: u8,
    pack:   Pack,
}
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<Pack, (u32, u8)>) -> u32 {
    if let Some((result, minute)) = memo.get(&state.pack).copied() {
        if state.minute >= minute {
            return result;
        }
    }
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}

#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct State {
    minute: u8,
    pack:   Pack,
}
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<Pack, (u32, u8)>) -> u32 {
    if let Some((result, minute)) = memo.get(&state.pack).copied() {
        if state.minute >= minute {
            return result;
        }
    }
```

execution time: 70 s (2x) 🥳

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u8,
    clay_robot:     u8,
    obsidian_robot: u8,
    geode_robot:    u8,
    ore:      u8,
    clay:     u8,
    obsidian: u8,
    geode:    u8,
}
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u8,
    clay_robot:     u8,
    obsidian_robot: u8,
    geode_robot:    u8,
    ore:      u8,
    clay:     u8,
    obsidian: u8,
    geode:    u8,
}
```

execution time: 59 s 👍

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u8,
    clay_robot:     u8,
    obsidian_robot: u8,
    geode_robot:    u8,
    ore:      u8,
    clay:     u8,
    obsidian: u8,
    geode:    u8,
}
```

execution time: 59 s 👍

1) cpu cache?

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u32,
    clay_robot:     u32,
    obsidian_robot: u32,
    geode_robot:    u32,
    ore:      u32,
    clay:     u32,
    obsidian: u32,
    geode:    u32,
}
```

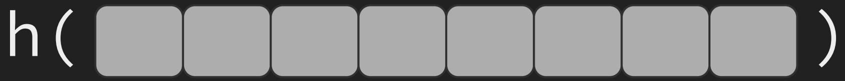
```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
struct Pack {
    ore_robot:      u8,
    clay_robot:     u8,
    obsidian_robot: u8,
    geode_robot:    u8,
    ore:      u8,
    clay:     u8,
    obsidian: u8,
    geode:    u8,
}
```

execution time: 59 s 👍

1) cpu cache?
2) hashing overhead.

```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

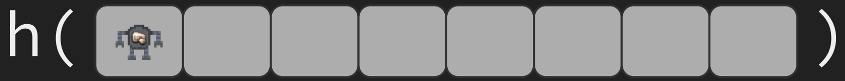


```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```

```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```

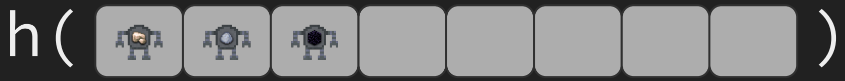


```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```

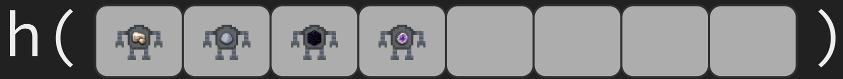
```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```



```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```



```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```

```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```

h(

)

```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```

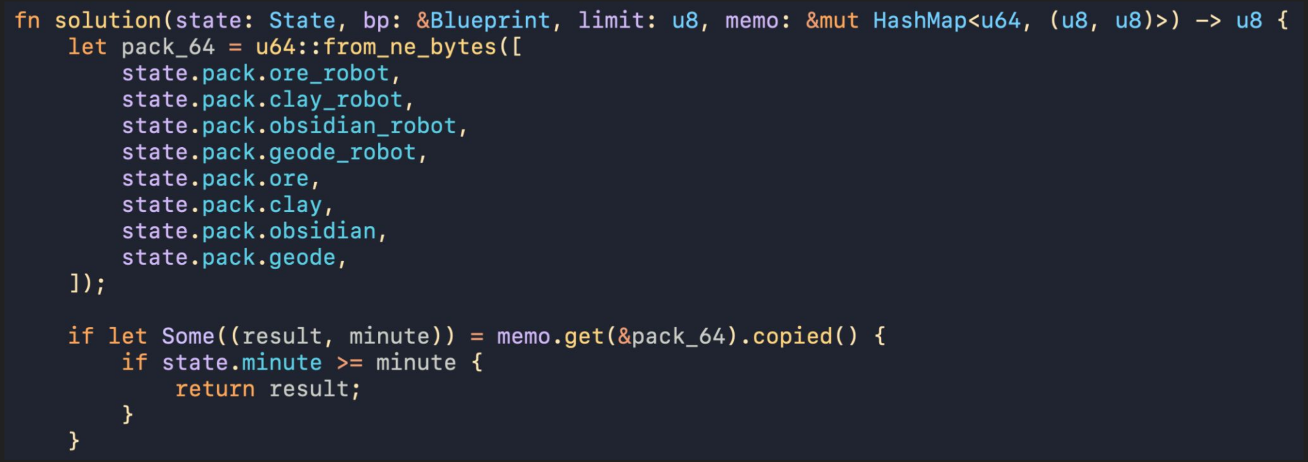
```rust
impl core::hash::Hash for Pack {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.ore_robot.hash(state);
        self.clay_robot.hash(state);
        self.obsidian_robot.hash(state);
        self.geode_robot.hash(state);
        self.ore.hash(state);
        self.clay.hash(state);
        self.obsidian.hash(state);
        self.geode.hash(state);
    }
}
```

```rust
fn hash(&mut self, bytes: &[u8]) {
    if bytes.len() % 8 != 0 {
        // put into internal buffer.
    }

    // ...
}
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<u64, (u8, u8)>) -> u8 {
    let pack_64 = unsafe { core::mem::transmute(state.pack) };

    if let Some((result, minute)) = memo.get(&pack_64).copied() {
        if state.minute >= minute {
            return result;
        }
    }
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<u64, (u8, u8)>) -> u8 {
    let pack_64 = unsafe { core::mem::transmute(state.pack) };

    if let Some((result, minute)) = memo.get(&pack_64).copied() {
        if state.minute >= minute {
            return result;
        }
    }
```

execution time: 44 s 🥳



```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<u64, (u8, u8)>) -> u8 {
    let pack_64 = u64::from_ne_bytes([
        state.pack.ore_robot,
        state.pack.clay_robot,
        state.pack.obsidian_robot,
        state.pack.geode_robot,
        state.pack.ore,
        state.pack.clay,
        state.pack.obsidian,
        state.pack.geode,
    ]);

    if let Some((result, minute)) = memo.get(&pack_64).copied() {
        if state.minute >= minute {
            return result;
        }
    }
```

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, Default)]
#[repr(C)]
struct Pack {
    ore_robot:      u8,
    clay_robot:     u8,
    obsidian_robot: u8,
    geode_robot:    u8,
    ore:      u8,
    clay:     u8,
    obsidian: u8,
    geode:    u8,
}
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8, memo: &mut HashMap<u64, (u8, u8)>) -> u8 {
    let pack_64 = u64::from_ne_bytes([
        state.pack.ore_robot,
        state.pack.clay_robot,
        state.pack.obsidian_robot,
        state.pack.geode_robot,
        state.pack.ore,
        state.pack.clay,
        state.pack.obsidian,
        state.pack.geode,
    ]);

    if let Some((result, minute)) = memo.get(&pack_64).copied() {
        if state.minute >= minute {
            return result;
        }
    }
```

🚀 tip #5

use bigger hashmap keys?

🚀 tip #5

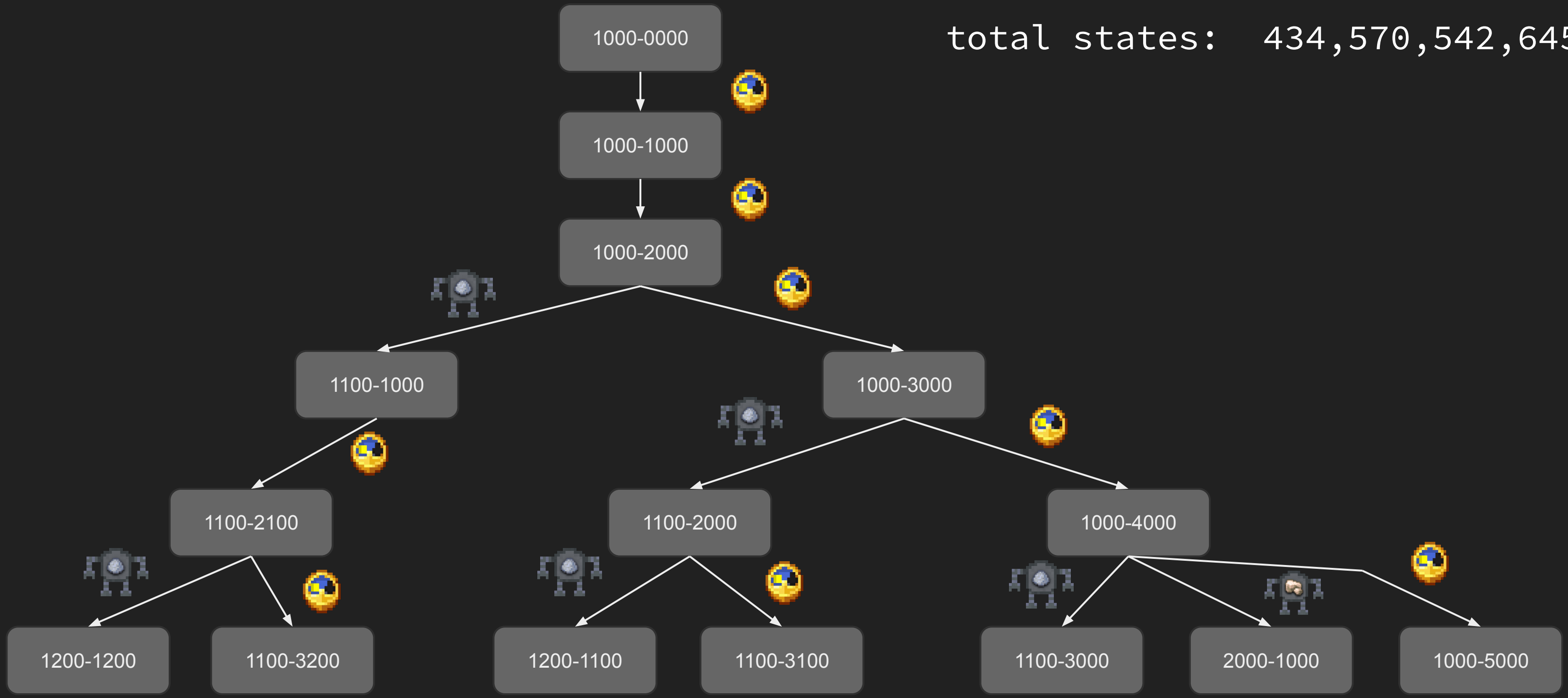
understand how abstractions work,

so you can use them effectively

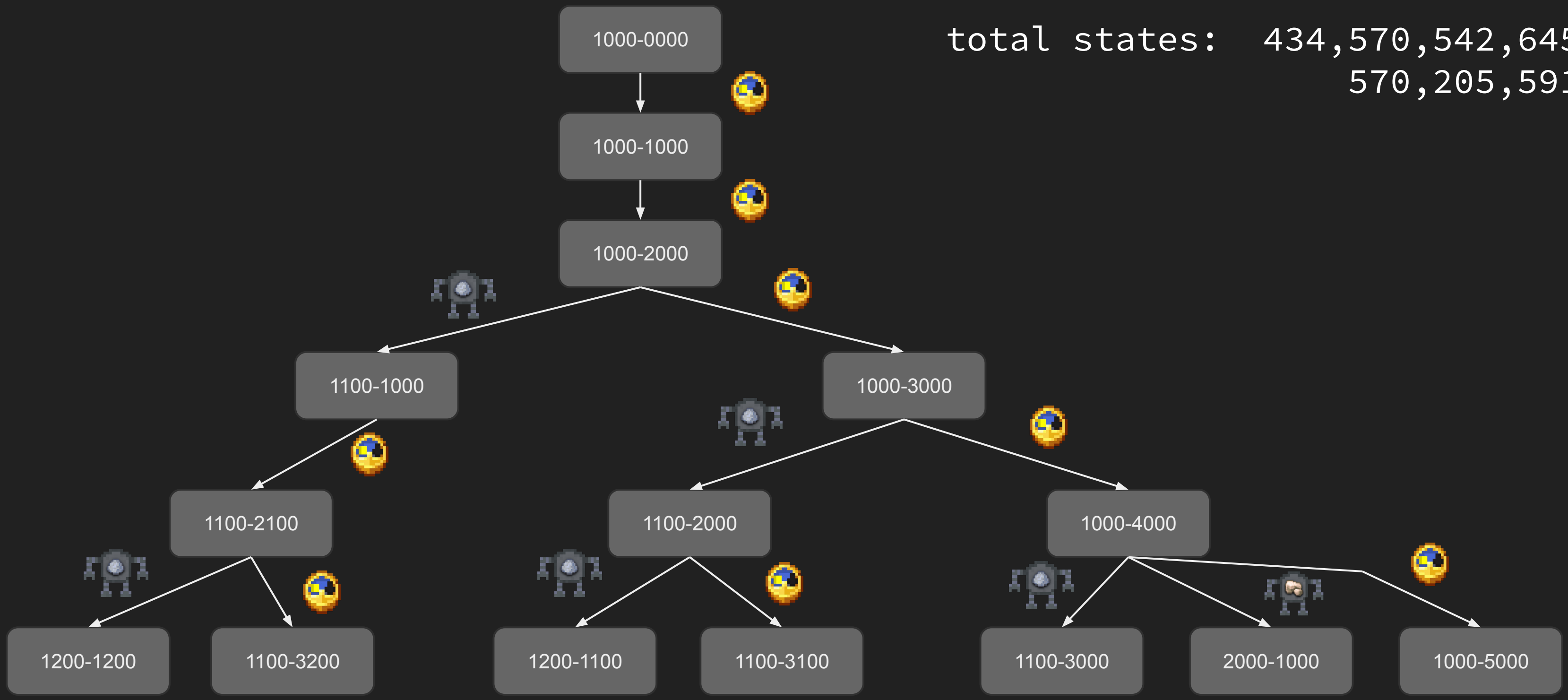
🤪

🤪

`44 s  ->  24 ms`



total states: 434,570,542,645



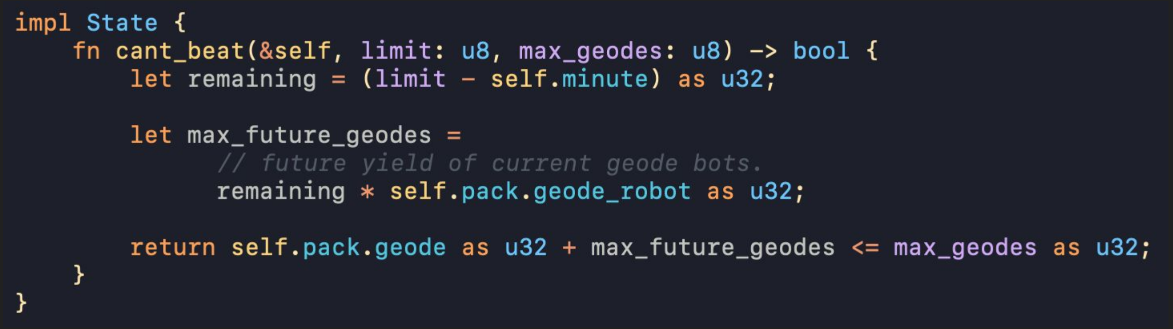
```
total states:  434,570,542,645
                   570,205,591
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8,
    memo: &mut HashMap<u64, (u8, u8)>, max_result: &mut u8
) -> u8 {
```

```rust
fn solution(state: State, bp: &Blueprint, limit: u8,
    memo: &mut HashMap<u64, (u8, u8)>, max_result: &mut u8
) -> u8 {
    if state.cant_beat(limit, *max_result) {
        return 0;
    }

    // ...
}
```

```rust
impl State {
    fn cant_beat(&self, limit: u8, max_geodes: u8) -> bool {
```

```rust
impl State {
    fn cant_beat(&self, limit: u8, max_geodes: u8) -> bool {
        let max_future_geodes = ?;

        return self.pack.geode as u32 + max_future_geodes <= max_geodes as u32;
    }
}
```



```rust
impl State {
    fn cant_beat(&self, limit: u8, max_geodes: u8) -> bool {
        let remaining = (limit - self.minute) as u32;

        let max_future_geodes =
              // future yield of current geode bots.
              remaining * self.pack.geode_robot as u32;

        return self.pack.geode as u32 + max_future_geodes <= max_geodes as u32;
    }
}
```

```rust
impl State {
    fn cant_beat(&self, limit: u8, max_geodes: u8) -> bool {
        let remaining = (limit - self.minute) as u32;

        let max_future_geodes =
              // future yield of current geode bots.
              remaining * self.pack.geode_robot as u32
              // max future yield, if we build one geode bot
              // on all future turns.
            + remaining*(remaining-1)/2;

        return self.pack.geode as u32 + max_future_geodes <= max_geodes as u32;
    }
}
```

```rust
impl State {
    fn cant_beat(&self, limit: u8, max_geodes: u8) -> bool {
        let remaining = (limit - self.minute) as u32;

        let max_future_geodes =
              // future yield of current geode bots.
              remaining * self.pack.geode_robot as u32
              // max future yield, if we build one geode bot
              // on all future turns.
            + remaining*(remaining-1)/2;

        return self.pack.geode as u32 + max_future_geodes <= max_geodes as u32;
    }
}
```

execution time: 3.36s 🥳

```
states visited
  brute force:        434,570,542,645
  basic cache:            834,619,249
  earlier result:         570,205,591
  max_result:              82,929,763
```

```rust
impl State {
    fn cant_beat(&self, limit: u8, max_geodes: u8) -> bool {
        let remaining = (limit - self.minute) as u32;

        let max_future_geodes =
              // future yield of current geode bots.
              remaining * self.pack.geode_robot as u32
              // max future yield, if we build one geode bot
              // on all future turns.
            + remaining*(remaining-1)/2;

        return self.pack.geode as u32 + max_future_geodes <= max_geodes as u32;
    }
}
```

execution time: 3.36s 🥳

have:

- resources
- robots

goal:

- maximize number of geodes

rules:

- robot -> 1 resource at end of turn
- start with 1 ore robot
- can build up to 1 robot per turn

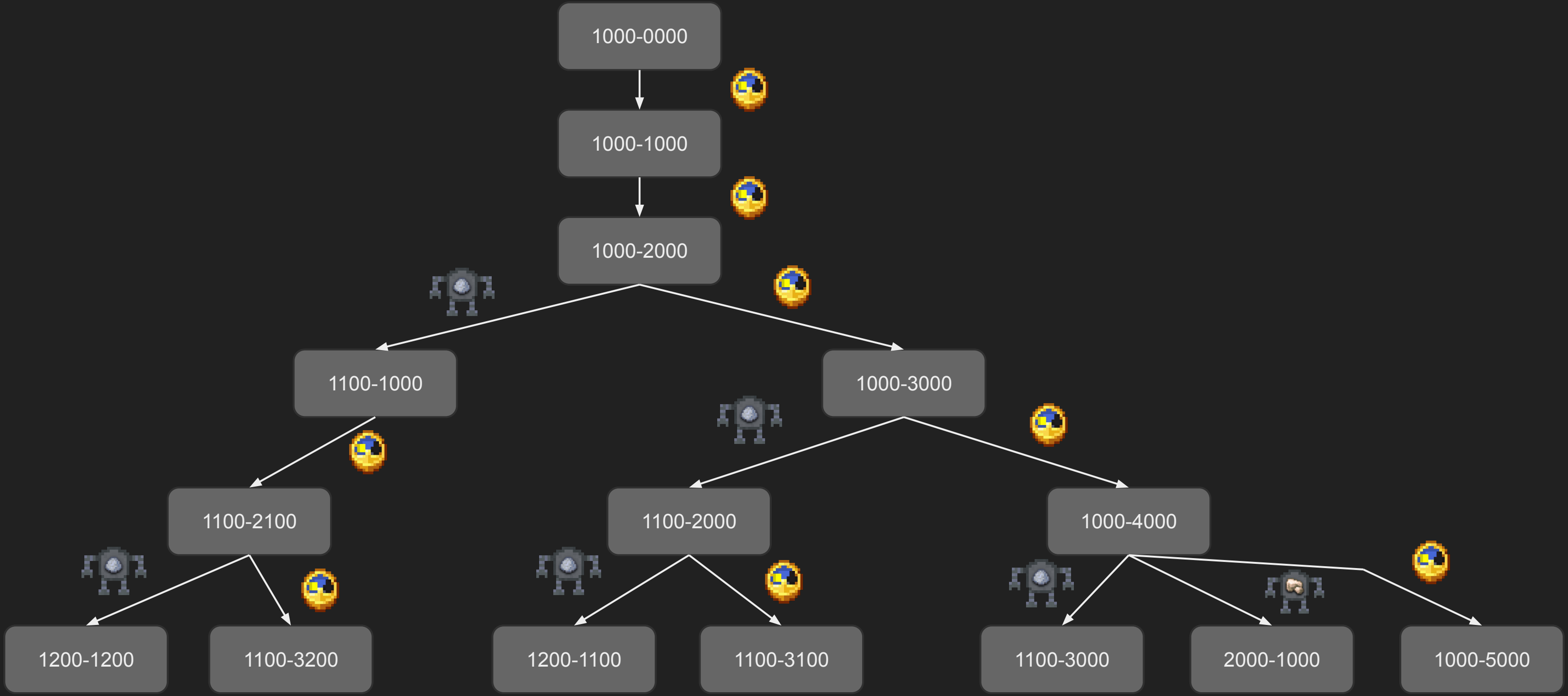
```
blueprint 1:
  ore robot:          4 ore
  clay robot:         2 ore
  obsidian robot:     3 ore, 14 clay
  geode robot:        2 ore, 7 obsidian
```

```rust
if state.can_build_geode_robot(bp) {
    result = result.max(solution(state.step().build_geode_robot(bp), bp, limit, memo, max_result));
}

if state.can_build_obsidian_robot(bp) {
    // can only build one bot per turn.
    // don't need more bots, if we're producing enough,
    // so we can build the most expensive bot on each turn.
    if state.pack.obsidian_robot < bp.max_obsidian_cost() {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit, memo, max_result));
    }
}

if state.can_build_clay_robot(bp) {
    if state.pack.clay_robot < bp.max_clay_cost() {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit, memo, max_result));
    }
}

if state.can_build_ore_robot(bp) {
    if state.pack.ore_robot < bp.max_ore_cost() {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit, memo, max_result));
    }
}

result = result.max(solution(state.step(), bp, limit, memo, max_result));
```

```
states visited
  brute force:      434,570,542,645
  basic cache:          834,619,249
  earlier result:       570,205,591
  max_result:            82,929,763
  enough bots:           12,741,390
```

```
if state.can_build_geode_robot(bp) {
    result = result.max(solution(state.step().build_geode_robot(bp), bp, limit, memo, max_result));
}

if state.can_build_obsidian_robot(bp) {
    // can only build one bot per turn.
    // don't need more bots, if we're producing enough,
    // so we can build the most expensive bot on each turn.
    if state.pack.obsidian_robot < bp.max_obsidian_cost() {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit, memo, max_result));
    }
}

if state.can_build_clay_robot(bp) {
    if state.pack.clay_robot < bp.max_clay_cost() {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit, memo, max_result));
    }
}

if state.can_build_ore_robot(bp) {
    if state.pack.ore_robot < bp.max_ore_cost() {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit, memo, max_result));
    }
}

result = result.max(solution(state.step(), bp, limit, memo, max_result));
```

execution time: 417 ms 🥳


1000-0000
1000-1000
1000-2000
1100-1000
1000-3000
1100-2100
1100-2000
1000-4000
1200-1200
1100-3200
1200-1100
1100-3100
1100-3000
2000-1000
1000-5000

```rust
fn solution(state: State, bp: &Blueprint, limit: u8,
    memo: &mut HashMap<u64, (u8, u8)>, max_result: &mut u8,
    can_ore: bool, can_clay: bool, can_obsidian: bool
) -> u8 {
```

```rust
let mut result = 0;

if state.can_build_geode_robot(bp) {
    result = result.max(solution(state.step().build_geode_robot(bp), bp, limit, memo, max_result, true, true, true));
}

let mut new_can_obsidian = true;
if state.can_build_obsidian_robot(bp) {
    new_can_obsidian = false;

    if can_obsidian && state.pack.obsidian_robot < bp.max_obsidian_cost() {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

let mut new_can_clay = true;
if state.can_build_clay_robot(bp) {
    new_can_clay = false;

    if can_clay && state.pack.clay_robot < bp.max_clay_cost() {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

let mut new_can_ore = true;
if state.can_build_ore_robot(bp) {
    new_can_ore = false;

    if can_ore && state.pack.ore_robot < bp.max_ore_cost() {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

result = result.max(solution(state.step(), bp, limit, memo, max_result, new_can_ore, new_can_clay, new_can_obsidian));
```

```rust
let mut result = 0;

if state.can_build_geode_robot(bp) {
    result = result.max(solution(state.step().build_geode_robot(bp), bp, limit, memo, max_result, true, true, true));
}

let mut new_can_obsidian = true;
if state.can_build_obsidian_robot(bp) {
    new_can_obsidian = false;

    if can_obsidian && state.pack.obsidian_robot < bp.max_obsidian_cost() {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

let mut new_can_clay = true;
if state.can_build_clay_robot(bp) {
    new_can_clay = false;

    if can_clay && state.pack.clay_robot < bp.max_clay_cost() {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

let mut new_can_ore = true;
if state.can_build_ore_robot(bp) {
    new_can_ore = false;

    if can_ore && state.pack.ore_robot < bp.max_ore_cost() {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

result = result.max(solution(state.step(), bp, limit, memo, max_result, new_can_ore, new_can_clay, new_can_obsidian));
```

execution time: 23.99 ms 🥳

```
states visited
  brute force:       434,570,542,645
  basic cache:           834,619,249
  earlier result:        570,205,591
  max_result:             82,929,763
  enough bots:            12,741,390
  no idling:                 566,020
```

```rust
let mut result = 0;

if state.can_build_geode_robot(bp) {
    result = result.max(solution(state.step().build_geode_robot(bp), bp, limit, memo, max_result, true, true, true));
}

let mut new_can_obsidian = true;
if state.can_build_obsidian_robot(bp) {
    new_can_obsidian = false;

    if can_obsidian && state.pack.obsidian_robot < bp.max_obsidian_cost() {
        result = result.max(solution(state.step().build_obsidian_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

let mut new_can_clay = true;
if state.can_build_clay_robot(bp) {
    new_can_clay = false;

    if can_clay && state.pack.clay_robot < bp.max_clay_cost() {
        result = result.max(solution(state.step().build_clay_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

let mut new_can_ore = true;
if state.can_build_ore_robot(bp) {
    new_can_ore = false;

    if can_ore && state.pack.ore_robot < bp.max_ore_cost() {
        result = result.max(solution(state.step().build_ore_robot(bp), bp, limit, memo, max_result, true, true, true));
    }
}

result = result.max(solution(state.step(), bp, limit, memo, max_result, new_can_ore, new_can_clay, new_can_obsidian));
```

execution time: 23.99 ms 🥳

```
                    states visited   time
brute force:       434,570,542,645    DNF
basic cache:           834,619,249    140 s
earlier result:        570,205,591     70 s
u8                     570,205,591     44 s
max_result:             82,929,763   3359 ms
enough bots:            12,741,390    417 ms
no idling:                 566,020     24 ms
```

```
                    states visited   time
brute force:       434,570,542,645    DNF
basic cache:           834,619,249    140 s
earlier result:        570,205,591     70 s
u8                     570,205,591     44 s
max_result:             82,929,763   3359 ms
enough bots:            12,741,390    417 ms
no idling:                 566,020     24 ms
?                                ?      3 ms
```

```
                    states visited   time
brute force:       434,570,542,645    DNF
basic cache:           834,619,249    140 s
earlier result:        570,205,591     70 s
u8                     570,205,591     44 s
max_result:             82,929,763   3359 ms
enough bots:            12,741,390    417 ms
no idling:                 566,020     24 ms
?                                ?      3 ms
```

🤔 exercise:
given this data, what might the last optimization be?

```
                    states visited   time
brute force:       434,570,542,645    DNF
basic cache:           834,619,249    140 s
earlier result:        570,205,591     70 s
u8                     570,205,591     44 s
max_result:        7x   82,929,763   3359 ms  13x
enough bots:       6x   12,741,390    417 ms   8x
no idling:        22x      566,020     24 ms  17x
?                                ?      3 ms
```

🤔 exercise:
given this data, what might the last optimization be?

```
                    states visited   time      cache hits
brute force:       434,570,542,645    DNF             n/a
basic cache:           834,619,249    140 s           48%
earlier result:        570,205,591     70 s           48%
u8                     570,205,591     44 s           48%
max_result:             82,929,763   3359 ms          46%
enough bots:            12,741,390    417 ms          39%
no idling:                 566,020     24 ms           5%
?                                ?      3 ms            ?
```

🤔 exercise:
given this data, what might the last optimization be?

	states visited	time	cache hits
brute force:	434,570,542,645	DNF	*n/a*
basic cache:	834,619,249	140 s	48%
earlier result:	570,205,591	70 s	48%
u8	570,205,591	44 s	48%
max_result:	82,929,763	3359 ms	46%
enough bots:	12,741,390	417 ms	39%
no idling:	566,020	24 ms	5%
no memo table:	674,356	3 ms	*n/a*

doing less