In class, we described an algorithm for constraint propagation and search that used an explicit assignment function (referred to as σ). Today, we'll describe an equivalent algorithm that doesn't require an assignment function, which is instead implicit in the domains. We'll refer to it as the "implied assignment" version of the algorithm. It is as follows:

`solve`(`problem`: unary and binary constraints)

- `domains` = `init_domains`()
- `unary_sat`(`problem`, `domains`) (apply unary constraints)
- initialize `stack` as an empty stack
- while `true`:
  - `result` = `binary_prop`(`problem`, `domains`) (propagate binary constraints)
  - if `result` reports no conflict:
    - if |`domains`[`var`] = 1| for all `var`, this is a consistent assigment
    - `orig_domains` = `copy`(`domains`)
    - `var`, `val` = `search`(`domains`)
    - add (`var`, `val`, `orig_domains`) to `stack`
  - otherwise:
    - if `stack` is empty, there is no consistent assignment
    - `backtrack`(`stack`, `domains`)

`unary_sat`(`problem`: unary and binary constraints, `domains`: tile domains)

- for each unary constraint `c` ∈ `problem` on variable `var`:
  - restrict `domains`[`var`] to satisfy `c`

`binary_prop`(`problem`: unary and binary constraints, `domains`: tile domains)

- do until `domains` doesn't change:
  - for each binary constraint `c` ∈ `problem` on variables `var1`, `var2`:
    - if there exists `val1` ∈ `domains`[`var1`] which doesn't satisfy `c` for any `val2` ∈ `domains`[`var2`]:
      - remove `val1` from `domains`[`var1`]
      - if |`domains`[`var1`]| = 0, there is a conflict

`search`(`domains`: domains)

- `var` = an unassigned spot (according to `domains`)
- `val` = a possible num for `var` (according to `domains`)
- restrict the domain of `var` to `val`.

`backtrack`(`stack`: history stack, `domains`: domains)

- pop (`var`, `val`, `orig_domains`) from `stack`
- replace `domains` with `orig_domains`
- remove `val` from the domain of `var`

As specifically applied to Sudoku, we have:

solve(problem: unary constraints)

- domains = init_domains()
- restrict_domains(problem, domains)
- initialize stack as an empty stack
- while true:
  - if propagate(domains) reports no conflict:
    - if |domains[spot]| = 1 for all spots, this is a consistent assigment
    - orig_domains = copy(domains)
    - spot, num = search(domains)
    - add (spot, num, orig_domains) to stack
  - otherwise:
    - if stack is empty, there is no consistent assignment
    - domains = backtrack(stack)

propagate(domains: tile domains)

- do until domains doesn't change:
  - for each spot on the board:
    - if |domains[spot]| = 1, remove spot's num from its peers' domains
    - if the above caused any domain to be empty, there is a conflict

search(domains: domains)

- spot = an unassigned spot (according to domains)
- num = a possible num for spot (according to domains)
- restrict the domain of spot to num.

backtrack(stack: history stack, domains: domains)

- pop (spot, num, orig_domains) from stack
- replace domains with orig_domains
- remove num from the domain of spot

To show this algorithm's version of `binary_prop` (`propagate`) is correct, we must show that at termination, we have established arc-consistency.

- Suppose that after `propagate`(`domains`), there exists peers `spot1`, `spot2` violating arc-consistency.
- This requires some $\texttt{num1} \in \texttt{domains}[\texttt{spot1}]$ such that $\texttt{num1} = \texttt{num2}$ for all $\texttt{num2} \in \texttt{domains}[\texttt{spot2}]$.
- However, this can only be true if $|\texttt{domains}[\texttt{spot2}]| = 1$, in which case `num1` would have been removed from `domains`[`spot1`] in `propagate`(`domains`), a contradiction.

We can also see that no value is removed from a domain unless it specifically violates arc-consistency. Therefore, `propagate`(`domains`) correctly establishes arc-consistency without removing anything more than necessary.

Check out `demo.pdf` to see this algorithm in action on a game of "Quadoku" (a 4×4 version of Sudoku).

Having run through the demo, you'll notice that our algorithm is pretty inefficient in a few ways! Think about how you can optimize it.

Note that there's a lot of freedom in how you implement `search`. Consider the following: assuming that a solution exists for `domains`, we have that, for any `spot`, if we were to select a `num` at random from `domains`[`spot`], the probability of selecting correctly (i.e. selecting a `num` that is assigned in some solution) is bounded below by:

$$\Pr(\texttt{num} \sim \texttt{domains}[\texttt{spot}] \text{ is correct}) \geq \frac{1}{|\texttt{domains}[\texttt{spot}]|},$$

which follows from the fact that at least one such `num` is correct. Therefore, it's reasonable to optimize for this lower bound by selecting

$$\arg\min_{\texttt{spot}} |\texttt{domains}[\texttt{spot}]|.$$