Algorithm to Pseudocode

> This is very very basic pseudocode in an attempt to understand these algorithms.

Basic Program Structure

```
// Load the clauses from the source problem file
loadClauses();

// Evaluate and simplify clauses before main computation
initialEval();

// Loop through rest of clauses and evaluate the satisfiability
processClauses();
```

Algo 1 - SAT by backtracking

Give non-empty clauses C_1^...^C_m on $n > 0$ boolean variables $x_1 ... x_n$, represented as above, this algorithm finds a solution if and only if the clauses are satisfiable. It records its current progress in an array $m_1 ... m_n$ of "moves", who's significance is explained below.

1. [Initialise.] Set $a \leftarrow m$ and $d \leftarrow 1$. (Here a represents the number of active clauses, and d represents the depth-plus-one in an implicit search tree.)
2. [Choose.] Set L ← 2d. If C(L) ≲ C(L + 1), set L ← L + 1. Then set md ← (L & 1) + 4[C(L ⊕ 1) = 0]. Terminate successfully if C(L) = a.
3. [Remove ¬L.] Delete ¬L from all active clauses; but go to 5 if that would make a clause empty. (We want to ignore ¬L, because we are making L true.)
4. [Deactivate L's clauses.] Suppress all clauses that contain L. (Those clauses are now satisfied.). Then set a ← a - C(L), d ← d + 1, and return to step 2.
5. [Try again.] If md < 2, set md ← 3 - md, L ← 2d + (md & 1), and go back to step 3.
6. [Backtrack.] Terminate unsuccessfully if d = 1 (the clauses are unsatisfiable). Otherwise set d ← d - 1 and L ← 2d + (md & 1).
7. [Reactivate L's clauses.] Set a ← a + C(L), and unsuppress all clauses that contain L. (Those clauses are now unsatisfied, because L is no longer true.)
8. [Unremove ¬L.] Reinstate ¬L in all the active clauses that contain it. Then go back to step 5.

```
// This will contain the pseudocode for the above algorithm.
```

> WIP

Algo 2 - SAT by watching

> This algorithm works best with smaller problems.

Give non-empty clauses C_1^...^C_m on $n > 0$ boolean variables $x_1 ... x_n$, represented as above, this algorithm finds a solution if and only if the clauses are satisfiable. It records its current progress in an array $m_1 ... m_n$ of "moves", who's significance is explained below.

1. [Initialize.] Set $d ← 1$
2. [Rejoice or choose.] If $d > n$, terminate successfully. Otherwise set $m_d ← [W_{2d} = 0 \ or \ W_{2d+1} ≠ 0]$ and $l ← 2d + m_d$
3. [Remove $\bar{l}$ if possible.] For all j such that $\bar{l}$ is watched in C_j, watch another literal of C_j. But go to B5 if that can't be done.
4. [Advance.] Set $W_{\bar{l}} ← 0, d ← d + 1$, and return to B2.
5. [Try again.] If $m_d < 2$, set $m_d ← 3 - m_d$, $l ← 2d + (m_d & 1)$, and go to B3.
6. [Backtrack.] Terminate unsuccessfully if $d = 1$ (the clauses are unsatisfiable). Otherwise set $d ← d - 1$ and go back to B5.

Algo 3 - SAT by cyclic DPLL

> "explained above" means the previous section from the book. Which in this instance is page 32.

Given non-empty clauses $C_1 ∧...∧ C_m$ on $n > 0$ Boolean variables $x_1...x_n$, represented with lazy data structures and an active ring as explained above, this agorithm finds a solution if and only if the clauses are satisfiable. It records its current progress in an array $h_1...h_n$ of indices and an array $m_0...m_n$ of "moves," whose significance is explained below.

1. [Initialize.] Set $m_0 ← d ← h ← t ← 0$, and do the following for $k = n, n - 1, ..., 1:$ Set $x_k ← -1$ (denoting an unset value); if $W_{2k} ≠ 0$ or $W_{2k+1} ≠ 0$, set NEXT(k) $← h, h ← k$, and if $t = 0$ also set $t ← k$. Finally, if $t ≠ 0$, complete the active ring by setting NEXT(t) $← h$.
2. [Success?] Terminate if $t = 0$ (all clauses are satisfied). Otherwise set $k ← t$.
3. [Look for unit clauses.] Set $h ←$ NEXT(k) and use the subroutine in exercise 129 to compute $f ← [2h\ is\ a\ unit] + 2[2h + 1\ is\ a\ unit].$ If $f = 3$, go to D7. If $f = 1\ or\ 2$, set $m_{d+1} ← f + 3, t ← k$, and go to D5. Otherwise, if $h ≠ t$, set $k ← h$ and repeat this stage.
4. [Two-way branch.] Set $h ←$ NEXT(t) and $m_{d+1} ← [W_{2h} = 0\ or\ W_{2h+1} ≠ 0]$