algo

tl;dr

Brute force / exhaustive search

```
/*
 * Generate and test all possible
 * solutions.
 *
 * Stop when one works, or when you've
 * examined them all.
 *
 * Don't work too hard at reducing the
 * search space; "easy" optimizations
 * only.
 */
```

Decrease and conquer
…by constant amount

```
/*
 * Define problem recursively in terms
 * of subproblem of size N-c. Usually
 * c==1 or c==2.
 *
 * Extend result to obtain answer to
 * original problem.
 *
 * Recursively defined but often coded
 * iteratively (bottom-up).
 */
```

Decrease and conquer
…by constant factor

```
/*
 * Define problem recursively in terms
 * of subproblem of size N/k. Usually
 * k==2.
 *
 * Solve only one subproblem.
 *
 * Result often is already the answer
 * to original problem.
 */
```

Divide and conquer

```
/*
 * Define problem recursively in terms
 * of A subproblems of size N/B. Often
 * A==B.
 *
 * Solve all subproblems.
 *
 * Combine results to obtain answer to
 * original problem.
 */
```

Transform and conquer

```
/*
 * Often involves pre-sorting.
 *
 * Sorting the input first often allows
 * the creation of a more efficient
 * algorithm.
 */
```

Solving problems with graph algorithms

```
/*
 * Strategy 1: Modify a known graph
 * algorithm.
 *
 * Strategy 2: Use a known graph
 * algorithm as a subroutine.
 *
 * ... or do both!
 *
 */
```

Graph algorithm “Strategy 2”

```
/*
 * 1. Represent the problem in a clever
 *    way as a graph.
 *
 * 2. Run a graph algorithm.
 *
 * 3. Use output from step 2 to obtain
 *    answer to the original problem.
 */
```

Greedy algorithms

```
/*
 * Iteratively build a solution by
 * adding another item at each step.
 *
 * Always choose the best item that is
 * available and feasible.
 *
 * Key is ranking the items in a way
 * that is efficient to calculate.
 */
```

Dynamic programming

```
/*
 * Define problem recursively, solving
 * subproblems of smaller size.
 *
 * Avoid recalculation of identical
 * subproblems by storing and re-using
 * their solutions.
 */
```

Backtracking

```
/*
 * Builds a solution incrementally
 *
 * Explore (DFS-like) a state space of
 * partial solutions
 *
 * What is a state?
 * What is a dead-end?
 * Promising state => continue
 * Not promising state => backtrack
 */
```