

Greedy algorithms determine minimum number of coins to give while [making change](https://en.wikipedia.org/wiki/Change-making_problem). These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using only coins with values {1, 5, 10, 20}. The coin of the highest value, less than the remaining change owed, is the local optimum.

making the locally optimal choice at each stage

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

**Game Plan For A Contest Round**

Read through ALL the problems FIRST; sketch notes with algorithm, complexity, the numbers, data structs, tricky details, ...

* Brainstorm many possible algorithms - then pick the stupidest that works!
* DO THE MATH! (space & time complexity, and plug in actual expected and worst case numbers)
* Try to break the algorithm - use special (degenerate?) test cases
* Order the problems: shortest job first, in terms of your effort (shortest to longest: done it before, easy, unfamiliar, hard)

Coding a problem - For each, one at a time:

* Finalize algorithm
* Create test data for tricky cases
* Write data structures
* Stepwise refinement: write comments outlining the program logic
* Fill in code and debug *one section at a time*
* Get it working & verify correctness (use trivial test cases)
* Try to break the code - use special cases for code correctness
* Optimize progressively - only as much as needed, and keep all versions (use hard test cases to figure out actual runtime)
* Don't delete your extra debugging output, comment it out
* don't reuse variables
* COMMENT BEFORE CODE.
  + Comment Anything you had to think about
* Try not to use floating point; if you have to, put tolerances in everywhere (never test equality)

Complexity:

One deduces the O() run time of a program by examining its loops. The most nested (and hence slowest) loop dominates the run time and is the only one mentioned when discussing O() notation. A program with a single loop and a nested loop (presumably loops that execute *N* times each) is O(*N 2*), even though there is also a O(*N*) loop present.

Of course, recursion also counts as a loop and recursive programs can have orders like O(*b N*), O(*N!*), or even O(*N N*).

**Rules of thumb**

* When analyzing an algorithm to figure out how long it might run for a given dataset, the first rule of thumb is: modern (2004) computers can deal with 100M actions per second. In a five second time limit program, about 500M actions can be handled. Really well optimized programs might be able to double or even quadruple that number. Challenging algorithms might only be able to handle half that much. Current contests usually have a time limit of 1 second for large datasets.
* 16MB maximum memory use
* 210 ~approx~ 10 3
* If you have *k* nested loops running about *N* iterations each, the program has O(*N k*) complexity.
* If your program is recursive with *b* recursive calls per level and has *l* levels, the program O(*b l*) complexity.
* Bear in mind that there are *N!* permutations and *2 n* subsets or combinations of *N* elements when dealing with those kinds of algorithms.
* The best times for sorting *N* elements are O(*N* log *N*).
* **DO THE MATH!** Plug in the numbers.

#### Solution Paradigms

##### Generating vs. Filtering

Programs that generate lots of possible answers and then choose the ones that are correct (imagine an 8-queen solver) are *filters*. Those that hone in exactly on the correct answer without any false starts are *generators*. Generally, filters are easier (faster) to code and run slower. Do the math to see if a filter is good enough or if you need to try and create a generator.

##### Precomputation

Sometimes it is helpful to generate tables or other data structures that enable the fastest possible lookup of a result. This is called *precomputation* (in which one trades space for time). One might either compile precomputed data into a program, calculate it when the program starts, or just remember results as you compute them. A program that must translate letters from upper to lower case when they are in upper case can do a very fast table lookup that requires no conditionals, for example. Contest problems often use prime numbers - many times it is practical to generate a long list of primes for use elsewhere in a program.

##### Decomposition (The Hardest Thing At Programming Contests)

While there are fewer than 20 basic algorithms used in contest problems, the challenge of combination problems that require a combination of two algorithms for solution is daunting. Try to separate the cues from different parts of the problem so that you can combine one algorithm with a loop or with another algorithm to solve different parts of the problem independently. Note that sometimes you can use the same algorithm twice on different (independent!) parts of your data to significantly improve your running time.

##### Symmetries

Many problems have symmetries (e.g., distance between a pair of points is often the same either way you traverse the points). Symmetries can be 2-way, 4-way, 8-way, and more. Try to exploit symmetries to reduce execution time.

For instance, with 4-way symmetry, you solve only one fourth of the problem and then write down the four solutions that share symmetry with the single answer (look out for self-symmetric solutions which should only be output once or twice, of course).

##### Forward vs. Backward

Surprisingly, many contest problems work far better when solved backwards than when using a frontal attack. Be on the lookout for processing data in reverse order or building an attack that looks at the data in some order or fashion other than the obvious.

##### Simplification

Some problems can be rephrased into a somewhat different problem such that if you solve the new problem, you either already have or can easily find the solution to the original one; of course, you should solve the easier of the two only. Alternatively, like induction, for some problems one can make a small change to the solution of a slightly smaller problem to find the full answer.