

CS 91 USACO

Bronze Division

Unit 3: Problem Solving Using Algorithms



LECTURE 15: CRAFTING WINNING SOLUTIONS

DR. ERIC CHOU

IEEE SENIOR MEMBER



Objectives

- A good way to get a competitive edge is to write down a **game plan** for what you're going to do in a contest round.
- This will help you script out your actions, in terms of what to do both when things go right and when things go wrong.
- This way you can spend your thinking time in the round figuring out programming problems and not trying to figure out what the heck you should do next... it's sort of like precomputing your reactions to most situations.
- **Mental preparation** is also important.

Game Plan

SECTION 1



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)



MATH

- Math (Complexity Calculation)
- A (Algorithm)
- T (Test Cases)
- H (Highest priority to shortest jobs)



Coding a problem

- For each, one at a time:

- Finalize algorithm
- Create test data for tricky cases
- Write data structures
- Code the input routine and test it (write extra output routines to show data?)
- Code the output routine and test it
- Stepwise refinement: write comments outlining the program logic



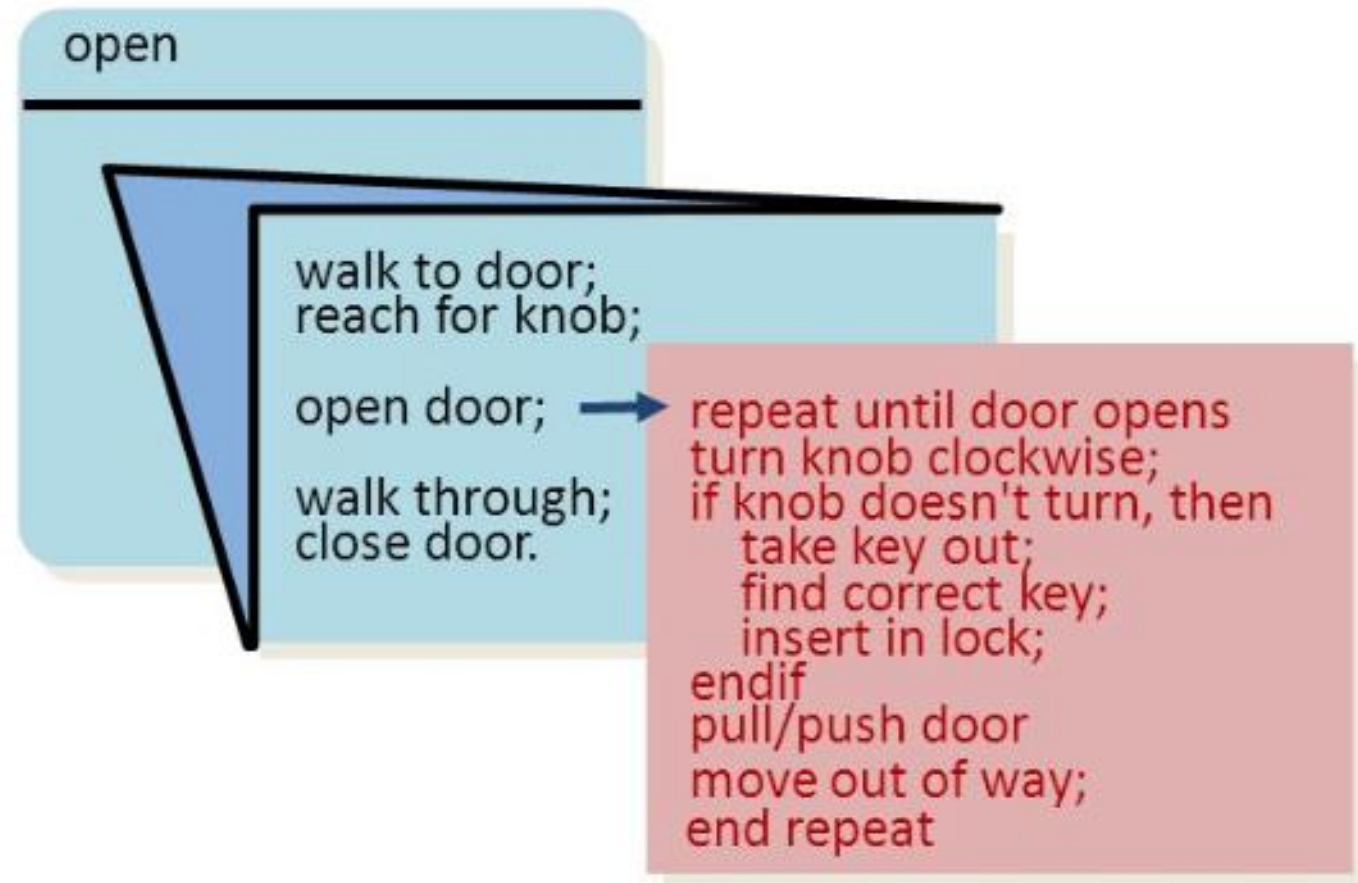
Coding a problem

- For each, one at a time:

- 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)



Stepwise Refinement



Time Management Strategy

SECTION 2



Take Partial Score?

The central question is:

- "When do you spend more time debugging a program, and when do you cut your losses and move on?"



Time management strategy and Damage Control Senarios

Consider these issues:

- How long have you spent debugging it already?
- What type of bug do you seem to have?
- Is your algorithm wrong?
- Does your data structures need to be changed?
- Do you have any clue about what's going wrong?



Time management strategy and Damage Control Senarios

Consider these issues:

- A short amount (20 mins) of debugging is better than switching to anything else; but you might be able to solve another from scratch in 45 mins.
- When do you go back to a problem you've abandoned previously?
- When do you spend more time optimizing a program, and when do you switch?
- Consider from here out - forget prior effort, focus on the future: how can you get the most points in the next hour with what you have?



Have a checklist to use before turning in your solutions:

Code freeze five minutes before end of contest?

- Turn asserts off.
- Turn off debugging output.

Tips & Tricks

SECTION 3



Cool Head



Coding Phase Tips

- Brute force it when you can
- KISS: Simple is smart!
- Hint: focus on limits (specified in problem statement)
- Waste memory when it makes your life easier (if you can get away with it)
- Don't delete your extra debugging output, comment it out
- Optimize progressively, and only as much as needed
- Keep all working versions!



Debugging Tips

- Whitespace is good,
- Use meaningful variable names,
- Don't reuse variables,
- Stepwise refinement,
- COMMENT BEFORE CODE.



Code Alternative Choice

- Avoid pointers if you can
- Avoid dynamic memory like the plague: statically allocate everything.
- Try not to use floating point; if you have to, put tolerances in everywhere (never test equality)



Comments

- Not long prose, just brief notes
- Explain high-level functionality: `++i; /* increase the value of i by */` is worse than useless
- Explain code trickery
- Delimit & document functional sections
- As if to someone intelligent who knows the problem, but not the code
- Anything you had to think about
- Anything you looked at even once saying, "now what does that do again?"
- Always comment order of array indices



After Contests

- Keep a log of your performance in each contest: successes, mistakes, and what you could have done better; use this to rewrite and improve your game plan!

Complexity

SECTION 4



Basics and Order Notation

- The fundamental basis of complexity analysis revolves around the notion of "big oh" notation, for instance: $O(N)$.
- This means that the algorithm's execution speed or memory usage will double when the problem size doubles.
- An algorithm of $O(N^2)$ will run about four times slower (or use 4x more space) when the problem size doubles.
- Constant-time or space algorithms are denoted $O(1)$. This concept applies to time and space both; here we will concentrate discussion on time.



Basics and Order Notation

- 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

SECTION 5



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
- $2^{10} \sim \text{approx} \sim 10^3$



Rules of Thumb

- 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.



Examples

A single loop with N iterations is $O(N)$:

```
1  sum = 0
2  for i = 1 to n
3      sum = sum + i
```

A double nested loop is often $O(N^2)$:

```
# fill array a with N elements
1 for i = 1 to n-1
2     for j = i + 1 to n
3         if (a[i] > a[j])
            swap (a[i], a[j])
```



Examples

- Note that even though this loop executes $N \times (N+1) / 2$ iterations of the if statement, it is $O(N^2)$ since doubling N quadruples the execution times.
- Consider this well-balanced binary tree with four levels:



- An algorithm that traverses a general binary tree will have complexity $O(2^N)$.

Solution Paradigms

SECTION 6



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

Simpler/Smaller

- 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.