Intro to Programming Competitions

Michael Wright and Carson Holgate

Department of Computer Science
North Carolina State University
mdwrigh2@ncsu.edu
clholgat@ncsu.edu

February 22nd, 2011

What Competitions Are We Talking About?

- Competitions we **are** talking about are similar to:
  - ACM ICPC
  - IEEExtreme
  - Facebook's Hacker Cup

What Competitions Are We Talking About?

- Competitions we **are** talking about are similar to:
  - ACM ICPC
  - IEEExtreme
  - Facebook's Hacker Cup
- Competitions we **aren't** talking about:
  - Project Euler
  - Android Developer Challenge
  - International Obfuscated C Contest

Group Work & Reference Materials Allowed

- Group Work
  - Individual
  - Specific Group Members
  - Limited Number of Group Members
  - Unrestricted

Group Work & Reference Materials Allowed

- Group Work
  - Individual
  - Specific Group Members
  - Limited Number of Group Members
  - Unrestricted
- Reference Materials Allowed
  - None
  - Dead Trees
  - Internets
  - Anything

Languages & Time Frame

- Languages
  - We'll talk about these a bit later

Languages & Time Frame

- Languages
  - We'll talk about these a bit later
- Time Frame
  - $<$ 24 hours (typically 3 - 5 hours)
  - $\geq$ 24 hours (typically 24 - 72 hours)

Submission & Feedback

- Submission
  - Source Code
  - Output
  - Both

Submission & Feedback

- Submission
  - Source Code
  - Output
  - Both
- Feedback
  - Immediate Feedback
    - Exceptions Thrown
    - Compile Errors
    - Time Limit Exceeded
    - Correct/Incorrect
  - End of Competition

Run-times & Computers

- Run-times
  - Run-time sensitive
  - Run-time insensitive
  - All competitions are runtime sensitive to some degree (obviously). For some though, runtime is *everything*.
- Computers
  - One
  - Many

Competitors & Expectations

- Competitors
  - Academic - Undergraduates
  - Academic - Students
  - Open
- Expectations
  - Solved Most
  - Solved All

Example - ACM ICPC

- Specific Groups (Exactly 3)
- Dead Trees Available
- Languages Limited to C, C++, Java
- 5 hour time limit
- One computer for all 3 competitors
- "Immediate" feedback (15 minute delay)
- Source submission
- Generally not runtime sensitive
- Academic - Undergraduates
- Winners solve **most** of the problems (2009 ICPC winners solved 9 out of 12)

Example - IEEExtreme

- Limited Group Size (Up to 3)
- Internets
- Every competitor can bring their own computer(s)
- Languages Limited (C, C++, Java)
- 24 hour time limit (problems released every 6 hours)
- Immediate feedback (3-5 minute delay)
- Source submission
- Generally not runtime sensitive
- Academic - Students
- Winners solve all the problems, then ranked by time to completion

Example - Facebook's Hacker Cup

- Individual
- Internets
- No limit to language (free compiler must be available)
- 3 hour time limit
  - Max runtime is 6 minutes, but must be run on **your** system
  - This is a very runtime sensitive competition
- Immediate feedback (3-5 minute delay)
- Source submission
- Academic - Students

Example - ICFP

- Unrestricted group size
- Internets
- No limit to language
- 72 hour time limit
- Both source and answer submission
- Open to anyone
- Only one problem to solve, but open ended, meaning whoever solves it "best", wins
- Runtime is considered, but tends to be the absolute last measure considered (only matters when the other measurements are the same in every other way).

Preparation - Individuals

- There are plenty of previous competitions online - practice using them!
- There's a book, Programming Challenges by Skiena and Revilla
- Know the language you're going to use - we'll talk a bit more about this later
- Familiarize yourself with the most useful algorithms. Implementing them in the language you have to use would be a **great** idea.
- If you can bring in dead trees, figure out which you're bringing in, get to know them a bit
- If it is open internet, bookmark the pages you'll use, close pages that will distract you before you go
- If you can use your computer or bring in electronic resources, consider getting PDFs of the algorithm books (Ctrl-F is your friend)

Preparation - Groups

- Practice with your group!
- Roles for group members can be very useful. Typically broken down into:
  - Programmer - Does all the typing (pick your fastest typer, or whoever knows the language best). Translates the pseudocode to real code, applies optimizations (dynamic programming, etc.)
  - Solver - Determines the high level solution to the problem, maybe writes the pseudocode for it.
  - Debugger - Looks at the pseudocode, tries to generate the test cases for it. If the real code has issues, walks through it to see where there could be errors, again generates test cases to see where it fails
  - These are **NOT** absolute roles. Everyone has to be flexible.

Preparation - Groups

- Practice with your group!
- Roles for group members can be very useful. Typically broken down into:
  - Programmer - Does all the typing (pick your fastest typer, or whoever knows the language best). Translates the pseudocode to real code, applies optimizations (dynamic programming, etc.)
  - Solver - Determines the high level solution to the problem, maybe writes the pseudocode for it.
  - Debugger - Looks at the pseudocode, tries to generate the test cases for it. If the real code has issues, walks through it to see where there could be errors, again generates test cases to see where it fails
  - These are **NOT** absolute roles. Everyone has to be flexible.
- Make sure you get along with your group.

Preparation - Day Of

- Bring whatever materials you're allowed. If books, an algorithms book, a reference for your language, etc.
- Pencils and paper! You can usually work out the smaller problems by hand, makes it easier to generate test cases, write pseudocode, etc.
- Whiteboard
- Boiler plate code if you're allowed (Data structure implementations, generic input reading code, etc.)

Pitfalls

- Input - Read the specification *very* carefully
  - Pay special attention to whether it can accept multiple test cases in one file
  - Pay attention to possible edge cases
- Whitespace
  - Typically you'll receive example output. Pay special attention to all whitespace, particularly if dealing with it is part of the problem
- Unless you know that runtime is a big part of the competition, optimize for programmer efficiency first, and then optimize runtime efficiency later if you need to
- Know your primitive max and min values. You don't want to have an unnoticed overflow on an otherwise correct algorithm

Common Languages

- C
- C++
- Java

Uncommon, but useful languages

- Python/Perl/Ruby
  - These aren't uncommon languages, just uncommon to be on the list of restricted languages for programming contests today
- LISP
- Haskell/OCaml/ML

C/C++

- Pros:
  - Fast. If execution time is important this is probably your language of choice.
  - C is very small, easy to know just about all of it
  - Most algorithms have reference implementation in C and C++
  - C++ has a huge standard lib. Learn it, love it
  - A common competition language
- Cons:
  - Not typesafe
  - Not all code is necessarily portable
  - C has very few built-in data structures, meaning you'll have to write most by hand
  - Bugs can be subtle
  - Not easily read (an issue for group competitions)

Java

- Pros:
  - Common language for competitions
  - Large standard library
  - Relatively typesafe
  - Memory managed
  - Faster than dynamic languages, safer than C/C++
  - More portable than C/C++
- Cons:
  - Slower than C/C++
  - Very verbose
  - Typesafety typically means you have to write more code
  - Somewhat easily read

Python/Perl/Ruby

- Pros:
  - Programmer time efficient
  - Usually fast "enough"
  - Can be very readable
  - REPL makes it easy to test particular components, quickly run test cases
  - Large standard libs, custom objects can use easily
- Cons
  - Relatively slow
  - Lots of diversity, not everyone on your team will know a particular dynamic language
  - Not commonly available in competitions with a restricted language set

Others

- LISP
  - Still a consistent winner in the AI problem space
  - Higher order functions are incredibly useful
  - Unfortunately, this is rarely an option, and most people don't know it
  - Probably not worth learning for competitions, but useful if you already know it
- Haskell/OCaml/ML
  - A winner in some of the longer competitions (namely ICFP...)
  - Typesafety gives you a lot more confidence your code
  - Compiled code is very fast (comparable to C in some cases), and it is easier to reason in a recursive manner if you're used to this style
  - Not something I'd learn for competitions, and likely of limited use (a LISP is probably the better option for competitions, if you want a similar language)

Preface

Preface

Thought Processes

- Brute force solutions are okay - use them when it saves you time
- Recursive solutions are great. If you can make them tail recursive, there is little to no performance hit (with the right compiler). If the function is deterministic based on input, they're **very** easy to memoize, especially with dynamic languages
- Again, optimize only when necessary. Premature optimization is typically unnecessary and can cost you lots of time
- Break it up into discrete steps if possible. If you have multiple computers, agree on an output from one of the steps and you can parallelize the problem.

Data Structures

- Know your basic data structures cold
- Data structures, in my opinion, are the most fundamental piece of a knowledge for a competition.
- You should know things like:
  - Linked Lists
  - Arrays
  - Stacks
  - Queues
  - Dictionaries

Linked Lists

- Singly Linked
- Doubly Linked
- Circular

Node Example

```
class Node():
    def __init__(self, val=None, cdr = None):
        self.val = val
        self.cdr = cdr

    def __str__(self):
        return str(self.val)
```

LL Example

```
class LinkedList():
    def __init__(self):
        self.head = None

    def insert(self, node=None):
        if not self.head:
            self.head = node
            return
        curr_node = self.head
        while curr_node.cdr:
            pass
        curr_node.cdr = node
```

LL Usage

```
lst = LinkedList()
lst.insert(Node(1))
lst.insert(Node(2))

# [1,-]----> [2 , NULL]
```

Arrays

- You should know these
- Great for lookup, or if you know how many items you're going to have (or if the max is reasonable)

Stacks

- LIFO or FILO data structure
- Arrays in some languages (e.g. python) can be used as stacks with their push and pop methods
- Good for things like depth-first traversal of graphs, backtracking solutions, etc.
- Also can be used to do things like check matching pairs
  - Example: Matching Parens
  - Push '(' onto stack, on ')' pop it off.
  - If you try to pop on an empty stack, or at the end of the string the stack isn't empty, fail
  - Otherwise all parens are matching, which is success.

Queues

- Regular Queue
  - A FIFO data structure
  - Useful for breadth first searches in graphs, etc.
- Priority Queue
  - A data structure ordered by a specific attribute first, and then by the time they're inserted (FIFO)
  - Example: Hospitals with patients
  - Typically easier just to work with a sorted array or list, but if you need an efficient data structure for doing lots of inserts, this is the better choice

Dictionaries

- Content-based retrieval, rather than position based
- Typically implemented with hash tables, but not necessarily
- Useful if your language has it, probably not worth implementing if it doesn't

Strings

- Don't bother memorizing string searching algorithms (KMP, etc.)
- DO know how to process strings as input
  - Reading stdin, files
  - Extracting data from strings (str.split(),atoi(), scanf(), int())

Sorting

- Know the performance implications of sorting a list, performing actions (insert, lookup) on a sorted data structure
- Usually not necessary to memorize sorting algorithms
- If you are going to memorize one, make it quicksort