ACM International Collegiate Programming Contest — Training Session I

Maria Keet

Department of Computer Science, University of Cape Town, South Africa
mkeet@cs.uct.ac.za

August 6, 2016

Outline

1. Introduction
2. Solving problems
3. Sampling of different types of problems
    - Horror dash
    - Help my brother
    - Shopping mall
    - Ensuring Truth or Building a Tower

Outline

1. Introduction
2. Solving problems
3. Sampling of different types of problems
   - Horror dash
   - Help my brother
   - Shopping mall
   - Ensuring Truth or Building a Tower

Topics of training sessions

I. General strategy for reading the problem and solving it; some algorithms; work on problems

II. Problem-solving paradigms; classifying problems; work on problems

III. More problems and algorithms

IV. Practice contest

V. Focus on team strategy; work on problems

VI. Practice contest

Some useful resources

- UVa Online Judge
- CodeForce
- ICPC archive
- Competitive Programming 3 (v1 is freely available, and still useful)
- Skiena's Algorithm Design Manual (a reference book with lots and lots of algorithms)

Other notes for the training sessions

- I can't help you with coding
- Will assist with
    - Problems solving,
    - Pseudo code, and
    - Team strategy
- Bring laptop (at least some of you)
- For the first few sessions you do not need a team, but it will help if you have one in September

Outline

1. Introduction
2. Solving problems
3. Sampling of different types of problems
   - Horror dash
   - Help my brother
   - Shopping mall
   - Ensuring Truth or Building a Tower

Approach (1/2)

- Read description

Approach (1/2)

- Read description
- What is the task?

Approach (1/2)

- Read description
- What is the task?
- What is given?
    - data
    - variables
    - constraint
    - examples
- (cont'd on next page...)

Approach (2/2)

- Solve the problem
    - ‘essentially solve it’

Approach (2/2)

- Solve the problem
  - 'essentially solve it'
    - input data space
    - recognise underlying core issues
    - similarity with other problems

Approach (2/2)

- Solve the problem
  - ‘essentially solve it’
    - input data space
    - recognise underlying core issues
    - similarity with other problems
    - result: a solution on paper, knowing **what** needs to be done

Approach (2/2)

- Solve the problem
    - ‘essentially solve it’
        - input data space
        - recognise underlying core issues
        - similarity with other problems
        - result: a solution on paper, knowing **what** needs to be done
    - solve it algorithmically (**how** to do it—e.g., a sort, OSPF, complexity, ...)

Approach (2/2)

- Solve the problem
  - ‘essentially solve it’
    - input data space
    - recognise underlying core issues
    - similarity with other problems
    - result: a solution on paper, knowing **what** needs to be done
  - solve it algorithmically (**how** to do it—e.g., a sort, OSPF, complexity, ...)
  - code and test it, i.e., **do it** and verify solution

Types of puzzles/problems

- Can be divided along several dimensions

Types of puzzles/problems

- Can be divided along several dimensions
- Regarding the core problem
  - A. maths-y (e.g., probabilities, geometry)
  - B. algorithmically/general (still an elegant solution)
  - C. seeing patterns, brute force, *ad hoc*

Types of puzzles/problems

- Can be divided along several dimensions
- Regarding the core problem
  - A. maths-y (e.g., probabilities, geometry)
  - B. algorithmically/general (still an elegant solution)
  - C. seeing patterns, brute force, *ad hoc*
- The (im-)balance in the ‘what, how, do’:
  - a. conceptually hard, but (relatively) easier to implement
  - b. conceptually (relatively) easy, but laborious to implement
  - c. both relatively hard (happens at the finals)
  - d. both relatively easy (at least one problem in the regionals)

Team composition (1/2)

- **Your team needs strengths in all three areas (what, how, do)** to have a chance to win
- Your team might be able to get away with maths-y only solving, but more probably with algorithmically, and even better both
- Each member probably won't excel in all three components, but *together* you will

Team composition (1/2)

- **Your team needs strengths in all three areas (what, how, do)** to have a chance to win
- Your team might be able to get away with maths-y only solving, but more probably with algorithmically, and even better both
- Each member probably won't excel in all three components, but *together* you will
- What are *you* good at?
- What are *your team mates* good at?
- What is your team *together* as a whole good at?

Team composition (1/2)

- **Your team needs strengths in all three areas (what, how, do)** to have a chance to win
- Your team might be able to get away with maths-y only solving, but more probably with algorithmically, and even better both
- Each member probably won't excel in all three components, but *together* you will
- What are *you* good at?
- What are *your team mates* good at?
- What is your team *together* as a whole good at?
- And what you together are *not* good at: can you skip it? if not, who will learn enough to fill the gap?

Team composition (2/2)

- Who will do what?
- Dividing tasks aided by a *team manager*, who does (at least):

Team composition (2/2)

- Who will do what?
- Dividing tasks aided by a *team manager*, who does (at least):
    - Keeps track of who is doing what
    - Makes sure problems are not forgotten
    - Keeps track of time

Team composition (2/2)

- Who will do what?
- Dividing tasks aided by a *team manager*, who does (at least):
  - Keeps track of who is doing what
  - Makes sure problems are not forgotten
  - Keeps track of time
- pair solving option:
- pair programming; One person codes, the other (‘co-pilot’) watches:
- ‘fresh eyes’

Team composition (2/2)

- Who will do what?
- Dividing tasks aided by a *team manager*, who does (at least):
  - Keeps track of who is doing what
  - Makes sure problems are not forgotten
  - Keeps track of time
- pair solving option:
  - bouncing off ideas toward solution
  - possibly putting different solved pieces together
- pair programming; One person codes, the other (‘co-pilot’) watches:
- ‘fresh eyes’

Team composition (2/2)

- Who will do what?
- Dividing tasks aided by a *team manager*, who does (at least):
  - Keeps track of who is doing what
  - Makes sure problems are not forgotten
  - Keeps track of time
- pair solving option:
  - bouncing off ideas toward solution
  - possibly putting different solved pieces together
- pair programming; One person codes, the other ('co-pilot') watches:
  - Understands the solution
  - Points out bugs
  - Looks things up for pilot
- 'fresh eyes'

Team composition (2/2)

- Who will do what?
- Dividing tasks aided by a *team manager*, who does (at least):
  - Keeps track of who is doing what
  - Makes sure problems are not forgotten
  - Keeps track of time
- pair solving option:
  - bouncing off ideas toward solution
  - possibly putting different solved pieces together
- pair programming; One person codes, the other ('co-pilot') watches:
  - Understands the solution
  - Points out bugs
  - Looks things up for pilot
- 'fresh eyes'
  - Sometimes you get stuck in a dead-end looking for a solution
  - the third person looks at the problem 'untarnished'

and finally

- Solve ‘the easiest’ problem first, i.e.,
  - The one easiest *to your team*
  - Looking at what others submit first may, or may not, be a good heuristic to choose what’s easiest

and finally

- Solve ‘the easiest’ problem first, i.e.,
    - The one easiest *to your team*
    - Looking at what others submit first may, or may not, be a good heuristic to choose what’s easiest
- Don’t submit without doing some testing (recall: time penalty for incorrect submissions!)

and finally

- Solve 'the easiest' problem first, i.e.,
  - The one easiest *to your team*
  - Looking at what others submit first may, or may not, be a good heuristic to choose what's easiest
- Don't submit without doing some testing (recall: time penalty for incorrect submissions!)
- Don't get hung up on one problem for too long, but don't keep jumping between problems without solving any of them either

and finally

- Solve ‘the easiest’ problem first, i.e.,
  - The one easiest *to your team*
  - Looking at what others submit first may, or may not, be a good heuristic to choose what’s easiest
- Don’t submit without doing some testing (recall: time penalty for incorrect submissions!)
- Don’t get hung up on one problem for too long, but don’t keep jumping between problems without solving any of them either
- There is no harm in last-minute attempts (recall: the number of problems solved is more important than time taken)

Outline

1. Introduction

2. Solving problems

3. Sampling of different types of problems
   - Horror dash
   - Help my brother
   - Shopping mall
   - Ensuring Truth or Building a Tower

Horror dash

- UVa problem 11799
- type: (very) basic algorithmic and easy (in the grand scheme of things of ICPC problems)
- As exercise: use aforementioned methodological steps
- Solve it—at least the ‘what’-part—in 15 minutes

Help my brother

- UVa problem 11161
- type: maths, little to code, has a straightforward solution (but possibly TLE) and a more advanced one
- First exercise: understand the problem
    - map the problem space (what is asked for, examples, input space, output, ...)
    - what is needed?

Solution (direction)

- Combinatorics, within discrete mathematics
- In contests—typically easy and quick to code, but 1) finding the formula takes good knowledge of maths, and 2) there may be a snag on performance of the algorithm

Solution (direction)

- Combinatorics, within discrete mathematics
- In contests—typically easy and quick to code, but 1) finding the formula takes good knowledge of maths, and 2) there may be a snag on performance of the algorithm
- This problem: need to use Fibonacci sequences: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..., then take the median
- Brute force for each set?
- Seems inefficient. But how else?

Finalise the solution

- What about finding the largest number in the input, generate a Fibonacci sequence of that length, once, and then reuse that list for finding the answers to the shorter sets?

Finalise the solution

- What about finding the largest number in the input, generate a Fibonacci sequence of that length, once, and then reuse that list for finding the answers to the shorter sets?
- Or is there, perhaps, a way to compute the *n*-th Fibonacci number (where n is large) efficiently?

Finalise the solution

- What about finding the largest number in the input, generate a Fibonacci sequence of that length, once, and then reuse that list for finding the answers to the shorter sets?
- Or is there, perhaps, a way to compute the n-th Fibonacci number (where n is large) efficiently?
    - Yes! in $O(\log n)$ time using the efficient matrix power (Sect. 9.21 in CP3)

Shopping mall

- SWERC 2013 problem; and of the mini-contest of May 7, 2015
- Type: algorithmic
- Use aforementioned methodological steps and solve first the ‘what’-part, then the ’how’, and go back to the what, if needed
- Conceptually non-trivial but not extremely hard (if you know the algorithm...); somewhat laborious to implement

Shopping mall: toward a solution

- Task: calculate the shortest path between pairs of locations in the mall

Shopping mall: toward a solution

- Task: calculate the shortest path between pairs of locations in the mall
- What is the input like?

Shopping mall: toward a solution

- Task: calculate the shortest path between pairs of locations in the mall
- What is the input like?
  - N places ($N \leq 200$)
  - M connections ($N - 1 \leq M \leq 1000$)
  - floor level, coordinates x, y of the places
  - type of connection between points: `walking`, `stairs`, `lift`, or `escalator`
  - each connection type has a weight (see text)
  - constraint: same floor is always `walking`
  - actual input to the algorithm: two places, implicitly numbered by the sequence they're presented in the sample input

Shopping mall: toward a solution

- This input data clearly is a graph
- There are many graph processing algorithms
- Need a directed graph (with the restrictions of the problem)
- Hint from the task: *shortest path* between pairs of locations

Shopping mall: toward a solution

- This input data clearly is a graph
- There are many graph processing algorithms
- Need a directed graph (with the restrictions of the problem)
- Hint from the task: *shortest path* between pairs of locations
- Options: which shortest path algorithms?
  - Dijkstra
  - Floyd-Warshall
  - (and Bellman-Ford, and ...)
- What's the difference anyway?

Shopping mall: toward a solution

- What’s the difference anyway?
- Dijsktra: *single source* shortest path (Skiena Section 6.3.1)
- Floyd-Warshall for *All-Pairs* Shortest Path (and less code and practically faster cf Dijkstra); e.g.,

Shopping mall: toward a solution

- What's the difference anyway?
- Dijsktra: *single source* shortest path (Skiena Section 6.3.1)
- Floyd-Warshall for *All-Pairs* Shortest Path (and less code and practically faster cf Dijkstra); e.g.,
  - to find the 'center' vertex in a graph (minimizing the longest or average distance to all the other nodes),
  - or when you need to know the longest shortest-path distance over all pairs of vertices from one end to the other.
  - Refer to Skiena Section 6.3.2 for details

Shopping mall: toward a solution

- What's the difference anyway?
- Dijsktra: *single source* shortest path (Skiena Section 6.3.1)
- Floyd-Warshall for *All-Pairs* Shortest Path (and less code and practically faster cf Dijkstra); e.g.,
    - to find the 'center' vertex in a graph (minimizing the longest or average distance to all the other nodes),
    - or when you need to know the longest shortest-path distance over all pairs of vertices from one end to the other.
    - Refer to Skiena Section 6.3.2 for details
- For this problem, either one will do.

Tower of ASCII

- UVa problem 10333
- type: tedious...
- mainly to get you to do elaborate output formatting

Ensuring truth

- UVa problem 11357
- type: looks ‘scary’, but it isn’t
- Systematically work through the description to figure out what is really asked for
- It does help to know a bit of propositional logic, and BNF

Ensuring truth

- UVa problem 11357
- type: looks ‘scary’, but it isn’t
- Systematically work through the description to figure out what is really asked for
- It does help to know a bit of propositional logic, and BNF

⇒ only one clause needs to be satisfied to get TRUE. A clause can be satisfied if for all variables in the clause, its negation is not in the clause too.

Scheduled training dates

- Aug 6: 10:00-16:00
- **Aug 13: 10:00-16:00**
- Aug 27: 10:0-16:00
- Sept 3/10: 10:00-16:00
- Sept 17: 10:00-16:00
- Sept 24: 10:00-16:00 or Oct 1: 10:00-16:00
- Date of the regionals: TBD ("some Saturday between mid Sept and end Oct")