German Collegiate Programming Contest

GCPC Jury

gcpc-jury@nwerc.eu

12. Juni 2010

jury sample solutions

Problem	min. LOC	max. LOC
Absurd prices	17	50
Cheating or Not	86	104
Counterattack	23	60
Field Plan	48	117
Hacking	35	81
Last Minute Constructions	86	167
Lineup	29	76
Polynomial Estimates	24	55
Soccer Bets	17	99
The Two-ball Game	64	86
To score or not to score	61	117
$\sum$	490	1012

Absurd Prices

- interval sizes are up to 10^7
- brute force is too slow

Absurd Prices

- interval sizes are up to 10^7
- brute force is too slow
- idea: generate closest integer which has smaller absurdity than c

Absurd Prices

- interval sizes are up to 10^7
- brute force is too slow
- idea: generate closest integer which has smaller absurdity than c
- must have either a 5 instead of its last non-zero digit, or one more trailing zero.

Absurd Prices

- interval sizes are up to 10^7
- brute force is too slow
- idea: generate closest integer which has smaller absurdity than c
- must have either a 5 instead of its last non-zero digit, or one more trailing zero.
- check if the closest integer with smaller absurdity lies in the intervall $[0.95 \cdot c, 1.05 \cdot c]$

Absurd Prices

- interval sizes are up to 10^7
- brute force is too slow
- idea: generate closest integer which has smaller absurdity than c
- must have either a 5 instead of its last non-zero digit, or one more trailing zero.
- check if the closest integer with smaller absurdity lies in the intervall $[0.95 \cdot c, 1.05 \cdot c]$
- Most common errors: Tried brute force or used wrong bounds (e.g., rounding errors).

Cheating or Not

- Distribute $g \cdot m$ teams evenly over $g \leq 8$ groups.
- First position per group fixed (host/seeded teams)
- Other positions depend only on first position

$\Rightarrow$ No interdependencies between positions $2, 3, \ldots, m$!

Cheating or Not

- Distribute $g \cdot m$ teams evenly over $g \leq 8$ groups.
- First position per group fixed (host/seeded teams)
- Other positions depend only on first position

$\Rightarrow$ No interdependencies between positions $2, 3, \ldots, m$!

Simple Solution

For position $i = 2, 3, \ldots, m$:

- Enumerate all configurations for the i-th positions.
- Count how often each team is in each group.
- P(team t in group k)$= \frac{\text{configurations with team } t \text{ in group } k}{\text{total number of configurations}}$
- Sum up team strengths weighted with probabilities.

Complexity: $O(mg!g)$

Optimization

- Consider two partial configurations.
- If same teams are set, same completions are possible.

! Completion does not depend on order of the teams in the partial configuration.

⇒ Use Dynamic Programming!

- Complexity: $O(m2^g g^2)$

Counter Attack

- 2^n possibilities $\Rightarrow$ naive computation is too slow
- position $b_j \Rightarrow$ either pass from a_{j-1} or run from b_{j-1}
- recursion: $b_j = \min(a_{j-1} + pass^a_{j-1}, b_{j-1} + run^b_{j-1})$ (analogous for a_j)
- avoid duplicate computations by dynamic programming
- reduces complexity to $\Theta(n)$

Counter Attack

- 2^n possibilities $\Rightarrow$ naive computation is too slow
- position b_j $\Rightarrow$ either pass from a_{j-1} or run from b_{j-1}
- recursion: $b_j = \min(a_{j-1} + pass^a_{j-1}, b_{j-1} + run^b_{j-1})$ (analogous for a_j)
- avoid duplicate computations by dynamic programming
- reduces complexity to $\Theta(n)$
- Most common error: Implemented Dijkstra in $O(n^2)$ instead of DP or Dijkstra in $O(n \log n)$.

Field Plan Solution Outline

- Find strongly connected components with Tarjan's algorithm or algorithm of Aho, Hopcroft, Ullman
- Consider DAG of strongly connected components
- If this DAG has exactly one source (SCC with indegree zero), print out the nodes of this SCC
- if not, Yogi made a fault

Field Plan Solution Outline

- Find strongly connected components with Tarjan's algorithm or algorithm of Aho, Hopcroft, Ullman
- Consider DAG of strongly connected components
- If this DAG has exactly one source (SCC with indegree zero), print out the nodes of this SCC
- if not, Yogi made a fault
- Most common error: Didn't use SCCs, tried *n* depth first searches instead.

Hacking

- A text of length n can contain at most $n - m + 1$ different substrings of length m
- But there exist k^m many strings of length m with the first k letters of the alphabet

Hacking

- A text of length n can contain at most $n - m + 1$ different substrings of length m
- But there exist k^m many strings of length m with the first k letters of the alphabet
- There must be a string of length $\leq log(n)/log(k) + 1$ which does not occur in the given string

Hacking

- A text of length n can contain at most $n - m + 1$ different substrings of length m
- But there exist k^m many strings of length m with the first k letters of the alphabet
- There must be a string of length $\leq log(n)/log(k) + 1$ which does not occur in the given string
- Determine the first substring of length $l := \lfloor log(n)/log(k) \rfloor + 1$ which does not occur in the string
- Number of such substrings is $\leq n \cdot k$

Hacking

- Substrings can be seen as numbers in base k with l digits
- Use a boolean table of size k^l to store which substrings occur in the string

Hacking

- Substrings can be seen as numbers in base k with l digits
- Use a boolean table of size k^l to store which substrings occur in the string
- Use Rabin-Karp algorithm to determine in $O(n)$ the hash values of the substrings of length l which occur in the string

Hacking

- Substrings can be seen as numbers in base k with l digits
- Use a boolean table of size k^l to store which substrings occur in the string
- Use Rabin-Karp algorithm to determine in $O(n)$ the hash values of the substrings of length l which occur in the string
- Reconstruct substring from first hash value in the table which did not occur in the string.

Hacking

- Substrings can be seen as numbers in base k with l digits
- Use a boolean table of size k^l to store which substrings occur in the string
- Use Rabin-Karp algorithm to determine in $O(n)$ the hash values of the substrings of length l which occur in the string
- Reconstruct substring from first hash value in the table which did not occur in the string.
- Most common error: Implementation too slow, e.g. building a trie with all substrings of length m and searching for strings not present.

Last Minute Constructions

- The input is a tree combined with a set of directed edges and two distinct nodes, the source s and the target t of a route.
- Goal is to find a node-disjunct path from s to t using all directed edges.

Last Minute Constructions

- The processing can be reduced to a path construction using a depth-first-search.

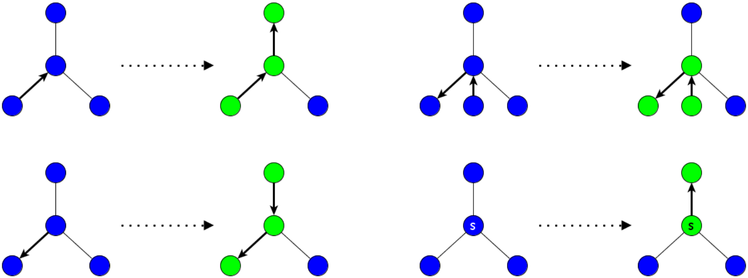
Last Minute Constructions

- The processing can be reduced to a path construction using a depth-first-search.
- Rooting the tree at t eliminates special treatment for t during the processing.

Last Minute Constructions

- The processing can be reduced to a path construction using a depth-first-search.
- Rooting the tree at t eliminates special treatment for t during the processing.
- Decide for every sub-tree if the path needs to enter it or exit it. Use special treatment for s.

Last Minute Constructions



- To check whether all tunnels have been used follow the constructed path.

Lineup

- Any player is proficient in at most 5 positions
- $\Rightarrow$ there are at most $5^7 \cdot 4! = 1\,875\,000$ valid positions

Lineup

- Any player is proficient in at most 5 positions
- $\Rightarrow$ there are at most $5^7 \cdot 4! = 1\,875\,000$ valid positions
- just do a brute-force search over all valid positions

Lineup

- Any player is proficient in at most 5 positions
- $\Rightarrow$ there are at most $5^7 \cdot 4! = 1\,875\,000$ valid positions
- just do a brute-force search over all valid positions
- Most common error: Computed $\mathrm{sum} \leftarrow \mathrm{sum} + \mathrm{backtrack}(\mathrm{pos} + 1)$ but returned 0 for impossible solutions. This can result in higher sums than for the correct solution.

Polynomial Estimates

- Always “YES” with 4 or less given values

Polynomial Estimates

- Always “YES” with 4 or less given values
- Otherwise, compute coefficients and check values

Polynomial Estimates

- Always "YES" with 4 or less given values
- Otherwise, compute coefficients and check values
- Linear equations on paper. Solution:

$$
\begin{aligned}
a &= 6x_1 \\
b &= -11x_1 + 18x_2 - 9x_3 + 2x_4 \\
c &= 6x_1 - 15x_2 + 12x_3 - 3x_4 \\
d &= -x_1 + 3x_2 - 3x_3 + x_4
\end{aligned}
$$

Polynomial Estimates

- Always "YES" with 4 or less given values
- Otherwise, compute coefficients and check values
- Linear equations on paper. Solution:

$$
\begin{aligned}
a &= 6x_1 \\
b &= -11x_1 + 18x_2 - 9x_3 + 2x_4 \\
c &= 6x_1 - 15x_2 + 12x_3 - 3x_4 \\
d &= -x_1 + 3x_2 - 3x_3 + x_4
\end{aligned}
$$

- Then $p(x) = (a + bx + cx^2 + dx^3)/6$

Polynomial Estimates

- There is an easier way

Polynomial Estimates

- There is an easier way
- 3rd derivative of a degree 3 polynomial is constant

Polynomial Estimates

- There is an easier way
- 3rd derivative of a degree 3 polynomial is constant
- Similar idea: Differences

Polynomial Estimates

- There is an easier way
- 3rd derivative of a degree 3 polynomial is constant
- Similar idea: Differences
- Given x_i, compute $x_i' = x_{i+1} - x_i$

Polynomial Estimates

- There is an easier way
- 3rd derivative of a degree 3 polynomial is constant
- Similar idea: Differences
- Given x_i, compute $x_i' = x_{i+1} - x_i$
- Iterate

Polynomial Estimates

- There is an easier way
- 3rd derivative of a degree 3 polynomial is constant
- Similar idea: Differences
- Given x_i, compute $x'_i = x_{i+1} - x_i$
- Iterate
- Check if $x'''_1 = x'''_2 = \cdots = x'''_{n-3}$

Polynomial Estimates

- There is an easier way
- 3rd derivative of a degree 3 polynomial is constant
- Similar idea: Differences
- Given x_i, compute $x'_i = x_{i+1} - x_i$
- Iterate
- Check if $x'''_1 = x'''_2 = \cdots = x'''_{n-3}$
- Most common errors: Forgot to read x_i if $n \leq 4$ or used wrong solution for the linear equations system.

Soccer Bets

- No Brainer
- World Champion $\Leftrightarrow$ won all matches

Soccer Bets

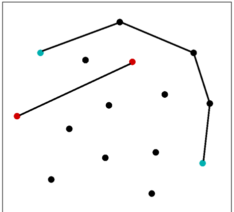
- No Brainer
- World Champion $\Leftrightarrow$ won all matches
- Most common error: Forgot to reset data structures between test cases.

The Two-ball Game

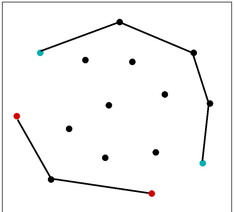
- testing all possible paths $s_1 \rightsquigarrow t_1$ and $s_2 \rightsquigarrow t_2$ is too slow

The Two-ball Game

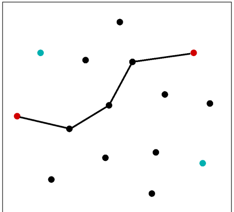
- testing all possible paths $s_1 \rightsquigarrow t_1$ and $s_2 \rightsquigarrow t_2$ is too slow
- idea:



at least one terminal inside: POSSIBLE



all terminals on CH – case 1: POSSIBLE



all terminals on CH – case 2: IMPOSSIBLE

The Two-ball Game

- solution:
  - compute the convex hull $\mathcal{H}$ as a list of its vertices ordered along its boundary clockwise (or counterclockwise)
  - answer IMPOSSIBLE if $s_1, t_1, s_2, t_2 \in \mathcal{H}$ and the points alternate on $\mathcal{H}$ like e.g. $\ldots s_1 \ldots s_2 \ldots t_1 \ldots t_2 \ldots$ or $\ldots s_2 \ldots t_1 \ldots t_2 \ldots s_1 \ldots$, etc.
- time complexity: $\mathcal{O}(n \log n)$

To Score Or Not To Score

- Directed graph
- Edge existence depends on distance to next opponent player
  - Computing e.g. with `Line2D.ptSegDist(opponent)` in Java
  - Each edge needs to be checked with each opponent

To Score Or Not To Score

- Directed graph
- Edge existence depends on distance to next opponent player
  - Computing e.g. with `Line2D.ptSegDist(opponent)` in Java
  - Each edge needs to be checked with each opponent
- Compute either max-flow between player with ball and goal
  - If *flow* ≥ 2 goal is possible
- Or make repeated DFS from source to target, each time ignoring one other player of the playing team
  - If goal is not reachable in one case, no goal is possible

Award Ceremony