2019 ICPC South Central USA Regional Contest Solution Outlines

The Judges

November 9, 2019

icpc.foundation
ICPC North America Regionals 2019
icpc international collegiate programming contest
icpc global programming tools sponsor
icpc north america sponsor
ICPC South Central USA Regional Contest

I– Some Sum – First solved at 0:03

Description

Given a number N, is it possible that the sum of N consecutive numbers is even, odd, or both?

I– Some Sum – First solved at 0:03

Description

Given a number N, is it possible that the sum of N consecutive numbers is even, odd, or both?

Solution

Since the bounds on the values of the numbers are small (1 to 100), just try all possible sums of length N. Running time: $O(N)$.

There's also an $O(1)$ time solutions by inspecting the value of $N \bmod 4$:

- if $N \bmod 4 = 0$, the answer is even
- if $N \bmod 4 = 2$, the answer is odd
- otherwise, the answer is either

This is because the sum of 4 consecutive numbers is even.

J– This Ain't Your Grandpa's Checkerboard – First solved at 0:07

Description

Given a grid of `B` and `W` characters, is it "valid"?

Invalid: unequal numbers of `B` and `W` values in a row/column, or a run of three of the same values in a row/column.

J– This Ain't Your Grandpa's Checkerboard – First solved at 0:07

Description

Given a grid of `B` and `W` characters, is it "valid"?

Invalid: unequal numbers of `B` and `W` values in a row/column, or a run of three of the same values in a row/column.

Solution

Read the input; check each row/column; report the answer.

Transpose: you can check each row, transpose the whole grid, then check each row (which was a column).

Running time: $O(n)$

H– Pulling their weight – First solved at 0:13

Description

Given a list of weights $(w_1, w_2, \ldots, w_n)$, find the smallest integer t where

$$\sum_{w_i < t} w_i = \sum_{t < w_i} w_i$$

i.e. sum of weights $< t$ equals sum of weights $> t$.
E.g. for $(1, 2, 3, 6)$ the best t is 4.

H– Pulling their weight – First solved at 0:13

Description

Given a list of weights $(w_1, w_2, \ldots, w_n)$, find the smallest integer t where

$$\sum_{w_i < t} w_i = \sum_{t < w_i} w_i$$

i.e. sum of weights $< t$ equals sum of weights $> t$.
E.g. for $(1, 2, 3, 6)$ the best t is 4.

Solution

Insight: t is either a given weight, or a weight plus one.

Consider each unique weight (or wt. plus one) as possible t, in sorted order. For each possible t, determine if it is the solution.

Can be done in $O(n \log n)$ time (for the sort).

H– Pulling their weight – First solved at 0:13

Feasible Solutions

When considering a certain t, if any weights are equal to t, we can just ignore them.

Some lists of integers are invalid inputs:

- If we just make up a list of weights, it might not have a solution.
  - E.g. there is no t for $(1, 2, 4)$.
- But the problem guarantees that every given input has a solution (so $(1, 2, 4)$ cannot occur).

H– Pulling their weight – First solved at 0:13

Solution Strategy Details

Define:

- $(s_1, s_2, \ldots, s_k)$ are the sorted *unique* weights
- $M(w)$ is the number of times w appears in the input (multiplicity)
- $P(i) = \sum_{j=1}^{i} s_i \cdot M(s_i)$ is a prefix sum
- $S = P(k)$ is the sum of the whole sequence

For each i in 1 to k:

- if $P(i) = S - P(i) + s_i \cdot M(s_i)$, then $t = s_i$
  - i.e. prefix and suffix sums are the same, ignoring all values equal to t
- if t is not any s_j and $P(i) = S - P(i)$, then $t = s_i + 1$
  - i.e. prefix and suffix sums are the same; no weight is equal to t

Runtime: $O(n \log n)$ for sorting, and $O(n)$ for everything else.

D– Flipping Patties – First solved at 0:13

Description

Given a list of times when N patties must be flipped, how many cooks are needed to handle all the flipping?

Each cook can do up to 2 things at a time, each pattie needs to be handled 3 times.

D– Flipping Patties – First solved at 0:13

Description

Given a list of times when N patties must be flipped, how many cooks are needed to handle all the flipping?

Each cook can do up to 2 things at a time, each pattie needs to be handled 3 times.

Solution

For each pattie, identify the three times ("events") when it must be handled.

Count all these in a hashtable or array. Find K, the largest number of simultaneous events that must occur.

The answer is $\lceil K/2 \rceil$. Runtime: $O(N)$.

L– Weird Flecks, But OK – First solved at 0:56

Description

Given N points in 3 dimensions, find the smallest circle that encompasses all of them in one of the three orthogonal planes.

L– Weird Flecks, But OK – First solved at 0:56

Description

Given N points in 3 dimensions, find the smallest circle that encompasses all of them in one of the three orthogonal planes.

Solution

Apply an efficient algorithm for smallest-circle three times (once for each plane).

Welzl's algorithm runs in $O(N)$ time (expected). Other approaches also work.

L– Weird Flecks, But OK – First solved at 0:56

Solution Strategy

Preliminaries: a circle can be uniquely defined by either two antipodal points or three non-collinear points.

The math for finding the circle for three points is left to the reader.

Naive approach: for each plane, try all circles defined by every $O(N^2)$ pair and $O(N^3)$ triple. For each circle, check if all N points are inside.

This approach is $O(N^4)$, too slow since $N \leq 5\,000$.

L– Weird Flecks, But OK – First solved at 0:56

Solution Strategy: Binary Search + Angle Sweep

Once we've reduced the problem to two dimensions, we can binary search on the radius of the minimum enclosing circle.

For any valid circle which covers all the points, we can translate it so at least two input points lie on the boundary of the circle. Then, for each input point, consider all circles with that point on its boundary. Each other point is included in some range of angles. If there exists some angle that includes all points, then this radius is attainable.

Runtime: $O(N^2 \log 1\,000)$.

L– Weird Flecks, But OK – First solved at 0:56

Solution Strategy: Ternary Search for the center

Consider the following function: $f(x, y)$ is the minimum radius of a circle that has center at (x, y) and covers all the points in the input. Evaluating $f(x, y)$ once can be done in $O(N)$ time.

We've now reduced our problem to finding the minimum value of $f(x, y)$. We can do this by doing two nested ternary searches: for a fixed x, the value of $f(x, y)$ is convex.

One heuristic to speed up this solution is to first take the convex hull of the input points before computing values of $f(x, y)$.

Runtime: $O(N \log^2 1\,000)$.

L– Weird Flecks, But OK – First solved at 0:56

Solution Strategy: Welzl's algorithm

Start with all points in set P and empty set R. Here R defines a set of boundary-defining points (up to 3).

Welzl(P, R):

- If P is empty or $|R| = 3$, return smallest circle defined by R.
- Choose a point $p \in P$ at random.
- Recurse with $P \backslash \{p\}$ and R.
- If the resulting circle C contains p, return C.
- Otherwise, try again (recurse) with $P \backslash \{p\}$ and $R \cup \{p\}$, and return that circle.

Running time: $O(N)$ expected time.

C– Code Names – First solved at 1:13

Description

Given a list ($n \leq 500$) of anagram words with no duplicate letters, find the size of the largest "swap-free" set.

Two words are NOT swap-free if they differ by swapping two letters (e.g. `abcdef` and `aecdbf` are NOT swap-free).

C– Code Names – First solved at 1:13

Description

Given a list ($n \leq 500$) of anagram words with no duplicate letters, find the size of the largest "swap-free" set.

Two words are NOT swap-free if they differ by swapping two letters (e.g. `abcdef` and `aecdbf` are NOT swap-free).

Solution

This appears to be an NP-hard problem – finding a maximum independent set. But there is more structure to the problem.

We can reduce this to a maximum bipartite matching (maximum flow) problem.

Identify the bipartite structure, construct the edges, find a maximum matching using network flow.

C– Code Names – First solved at 1:13

Solution Strategy

Define: two words are neighbors if they are NOT swap-free.

Identify the bipartite graph structure.

- Construct a graph with edges between neighbors.
- Each pair of words are anagrams – think permutations.
- Permutations have *parity* – equal to the parity of the number of *inversions*.
- Swapping (any) two letters changes the parity. Thus,
  - Two words *cannot be neighbors* if they have the same parity.
  - Two words *might* be neighbors if they have different parity.

Construct the graph in $O(N^2)$ time.

C– Code Names – First solved at 1:13

Solution Strategy

Once we've defined the bipartite structure on N words, the maximum independent set can be found by taking $N - M$ where M is the maximum number of matches in the bipartite graph.

Run maximum bipartite matching using any algorithm in $O(N^3)$ time worst-case to find M.

G– On Average They're Purple – First solved at 1:15

Description

Given a graph, how can Anna color the edges (red/blue) to force Bob to change colors the maximum number of times? Bob is trying to minimize the number of color changes while going between two designated nodes.

At first glance this feels like a difficult problem. But there's a simple solution hiding inside.

G– On Average They're Purple – First solved at 1:15

Description

Given a graph, how can Anna color the edges (red/blue) to force Bob to change colors the maximum number of times? Bob is trying to minimize the number of color changes while going between two designated nodes.

At first glance this feels like a difficult problem. But there's a simple solution hiding inside.

Solution

Just find the shortest path from source to destination using breadth-first search. If the path length is k then the answer is $k - 1$.

G– On Average They're Purple – First solved at 1:15

Solution Strategy

Proof idea that the shortest path finds the answer:

Consider starting at the source node. Anna colors every edge touching that node red (say), to maximize the chance of changing colors at a subsequent edge.

Now move one hop away, along any edge. Anna wants to color all newly-reachable edges blue, to force a color change.

This argument generalizes to k hops away.

Thus, the shortest path of length k changes colors $k-1$ times, which is the most Anna can force Bob to do.

Runtime for BFS: $O(N+M)$ for N nodes and M edges.

B– Birthday Paradox – First solved at 1:17

Description

Given a list of integers, find the probability of this list occurring.

Here each integer represents the number of people sharing some birthday.

Assume each person has 1/365 chance of any given birthday.

B– Birthday Paradox – First solved at 1:17

Description

Given a list of integers, find the probability of this list occurring.

Here each integer represents the number of people sharing some birthday.

Assume each person has 1/365 chance of any given birthday.

Solution

The solution can be derived using the multinomial probability distribution, and counting the number of possible reorderings.

B– Birthday Paradox – First solved at 1:17

Solution Strategy

Let $p = 1/365$ in what follows, and input values be $X = (x_1, x_2, \ldots, x_k)$.

The multinomial probability distribution P_M answers the question "What is the probability of observing all x_i on *fixed* days?" That is, the first k days of the year.

$$P_M(X) = \frac{n!}{\prod_{i=1}^{k} x_i!} p^n$$

where $n = \sum_{i=1}^{k} x_i$.

However, this isn't enough – we cannot assume these people's birthdays are the first k days of the year.

So how many different ways can we arrange their birthdays?

B– Birthday Paradox – First solved at 1:17

Solution Strategy

There are $\binom{365}{k}$ ways to choose k days out of 365.

Additionally, we can reorder the x_i's:

- if the x_i's are unique, then there are $k!$ orderings;
- if some x_i's are the same, then there are fewer...

Let $C = (c_1, c_2, \ldots, c_j)$ be the *frequency counts* of each x_i value. E.g. if $X = (1, 1, 1, 5, 5)$ then $C = (3, 2)$. Then the number of unique ways to reorder the x_i's is

$$\frac{(\sum c_j)!}{\prod(c_j!)}$$

which also arises from the multinomial distribution.

B– Birthday Paradox – First solved at 1:17

Solution Strategy

Putting this all together, the answer is:

$$P_M(X) \cdot \binom{365}{k} \cdot \frac{(\sum c_j)!}{\prod(c_j!)}$$

We can do all of this in log-space (the `lgamma` function is useful here).

Running time: $O(k)$ for k inputs.

F– Get Outta Dodge – First solved at 1:58

Description

Given a grid of walls and open spaces, what is the minimum number of walls that must be removed so that all locations can exit to the outside?

Complication: the grid is rotated 45 degrees.

F– Get Outta Dodge – First solved at 1:58

Description

Given a grid of walls and open spaces, what is the minimum number of walls that must be removed so that all locations can exit to the outside?

Complication: the grid is rotated 45 degrees.

Solution

For a graph with C components, you must remove $C - 1$ walls to make the graph connected. The underlying problem can be solved with Union-Find (Disjoint Sets), Flood-Fill, or repeated DFS/BFS to compute the number of components.

Dealing with the grid orientation makes this non-trivial.

F– Get Outta Dodge – First solved at 1:58

Solution Strategy

One possible representation:

```
  /\    /\    /\    /\
 /  \  /  \  /  \  /  \
/0,0 \/0,1 \/0,2 \/0,3 \
\    /\    /\    /\    /
 \  /  \  /  \  /  \  /
  \/1,0 \/1,1 \/1,2 \/
  /\    /\    /\    /\
 /  \  /  \  /  \  /  \
/2,0 \/2,1 \/2,2 \/2,3 \
\    /\    /\    /\    /
 \  /  \  /  \  /  \  /
  \/    \/    \/    \/
```

F– Get Outta Dodge – First solved at 1:58

Solution Strategy

Disjoint Sets: every grid cell is a disjoint set; the outside is a special set.

Visit every possible wall location; join cells that are lacking a wall.

The number of disjoint sets remaining is the number of walls that must be removed.

Runtime: $O(RC)$ for R rows and C columns.

E– Full Depth Morning Show – First solved at 2:42

Description

You are given a tree with N nodes, where every node has a tax value t_u and each edge has some weight w_i. The cost of a path between nodes u and v is equal to $(t_u + t_v)\,\mathrm{dist}(u, v)$. For each node u, compute the sum of the costs of all paths to all other nodes v.

E– Full Depth Morning Show – First solved at 2:42

Description

You are given a tree with N nodes, where every node has a tax value t_u and each edge has some weight w_i. The cost of a path between nodes u and v is equal to $(t_u + t_v)\,\mathrm{dist}(u, v)$. For each node u, compute the sum of the costs of all paths to all other nodes v.

Solution

The above expression can be broken up into $t_u\,\mathrm{dist}(u, v) + t_v\,\mathrm{dist}(u, v)$. Compute for some arbitrary root in $O(N)$ time. Then computing the answer for a neighboring node can be done in $O(1)$ time.

Formula

Fix some node u as the root, compute two quantities:

$$a_u = \sum_v \mathrm{dist}(u, v)$$

$$b_u = \sum_v t_v \, \mathrm{dist}(u, v)$$

E– Full Depth Morning Show – First solved at 2:42

Formula

Fix some node u as the root, compute two quantities:

$$a_u = \sum_v \operatorname{dist}(u, v)$$

$$b_u = \sum_v t_v \operatorname{dist}(u, v)$$

Solution Strategy

Then the answer for node u is just $t_u a_u + b_u$.

How do we compute $a_{u'}$ and $b_{u'}$ for some neighbor u' of u?

E– Full Depth Morning Show – First solved at 2:42

Formula

Fix some node u as the root, compute two quantities:

$$a_u = \sum_v \text{dist}(u, v)$$

$$b_u = \sum_v t_v \, \text{dist}(u, v)$$

Solution Strategy

Then the answer for node u is just $t_u a_u + b_u$.

How do we compute $a_{u'}$ and $b_{u'}$ for some neighbor u' of u?

E– Full Depth Morning Show – First solved at 2:42

Formula

Fix some node u as the root, compute two quantities:

$$a_u = \sum_v \text{dist}(u, v)$$

$$b_u = \sum_v t_v \, \text{dist}(u, v)$$

Solution Strategy (continued)

Let w be the length of the edge between u and u'. For all nodes in the subtree of u' when the tree is rooted at u, their distance to the root decreases by w. For all other nodes, the distance increases by w. If we let $\text{size}(u)$ be the size of the subtree rooted at u, then

$$a_{u'} = a_u + w(N - \text{size}(u)) - w \, \text{size}(u)$$

E– Full Depth Morning Show – First solved at 2:42

Formula

Fix some node u as the root, compute two quantities:

$$a_u = \sum_v \operatorname{dist}(u, v)$$

$$b_u = \sum_v t_v \operatorname{dist}(u, v)$$

Solution Strategy (continued)

Similarly, if we let $\operatorname{tax}(u)$ be the sum of the tax values of all nodes in the subtree rooted at u, then

$$b_{u'} = b_u + w(\operatorname{tax}(\text{root}) - \operatorname{tax}(u)) - w \operatorname{tax}(u)$$

E– Full Depth Morning Show – First solved at 2:42

Formula

Fix some node u as the root, compute two quantities:

$$a_u = \sum_v \operatorname{dist}(u, v)$$

$$b_u = \sum_v t_v \operatorname{dist}(u, v)$$

Runtime

Since these values can be updated in $O(1)$, walking and updating the tree and computing all values takes $O(N)$ time in total.

K– TLNDS – First solved at 3:47

Description

Given a tree with each vertex having an integer label, what is the length of the longest non-decreasing path, and how many paths are there of that length?

The vertices on the path do not have to be consecutive.

K– TLNDS – First solved at 3:47

Description

Given a tree with each vertex having an integer label, what is the length of the longest non-decreasing path, and how many paths are there of that length?

The vertices on the path do not have to be consecutive.

Solution

Similar to longest increasing subsequence, but on a tree.

Persistent Array solution: Keep track of the longest paths as you perform a depth-first traversal.

Segment Tree solution: Use Heavy-Light decomposition.

With appropriate data structures, this runs in $O(n \log n)$ or $O(n \log^2 n)$.

K– TLNDS – First solved at 3:47

Solution Strategy: Persistent Array

For simplicity, consider just one branch of the tree – this is the same as the array in longest increasing subsequence.

Straightforward dynamic programming requires $O(N^2)$ time; too slow.

For each new vertex label u_j we need to quickly find the longest path ending with a value $u_i \leq u_j$ where $i < j$.

One good solution: a stack of stacks. (This is a form of a *persistent array*.)

K– TLNDS – First solved at 3:47

Solution Strategy: Persistent Array

Stack of stacks:

- Each stack stores (vertex label, multiplicity) pairs.
- Keep as many stacks as the length of the (current) longest path.
- When we find a lower-valued label to end a path of length k, push that label onto stack k.

Naively, this gives an $O(N^2)$ solution, so we need a faster way.

K– TLNDS – First solved at 3:47

Solution Strategy: Persistent Array

As we descend the tree, at label u_i:

- Search for the stack k with the largest top value $\leq u_i$.
  - Add a stack if we find a new longer path.
- Compute the multiplicity of u_i using the sum of multiplicities of suitable entries in stack $k - 1$.
  - Each stack entry keeps the multiplicity prefix sum to compute this quickly.
- Push u_i and the multiplicity onto stack k.
- As we return from recursion (up the tree), pop off u_i.

Use binary search for finding k, and again for finding the multiplicity in stack $k - 1$.

Runtime: $O(n \log n)$.

K– TLNDS – First solved at 3:47

Solution Strategy: Heavy Light Decomposition

"Activate" nodes in increasing order of label. For each node, keep a single pair of (length, multiplicity).

To solve this on an array, we would query prefixes on a segment tree which combines pairs with

$$\text{merge}((l_1, m_1), (l_2, m_2)) = \begin{cases} (l_1, m_1) & (l_1 > l_2) \\ (l_2, m_2) & (l_1 < l_2) \\ (l_1, m_1 + m_2) & (l_1 = l_2) \end{cases}$$

To query all ancestors, we can use heavy-light decomposition, which breaks up our query into $O(\log n)$ queries in a single segment tree.

Runtime: $O(n \log^2 n)$.

A– All is Well – First solved at 3:51

Description

Given: a grid containing waypoints, find the shortest way to get ‘close enough’ to all waypoints (in order).

Each (non-waypoint) grid cell i takes a fixed amount of time c_i to enter.

A– All is Well – First solved at 3:51

Description

Given: a grid containing waypoints, find the shortest way to get ‘close enough’ to all waypoints (in order).

Each (non-waypoint) grid cell i takes a fixed amount of time c_i to enter.

Solution

Weighted graph problem: each cell is connected to its neighbors, with incoming edge cost c_i for cell i.

Run (sparse) Dijkstra’s algorithm on the whole graph. Dijkstra’s state contains:

- cost to get to this location (primary key)
- remaining waypoints (secondary)
- location (row and column)

A– All is Well – First solved at 3:51

Comments

State space: there are up to $n = 26 \cdot 100 \cdot 100 = 260\,000$ states.

Sparse Dijkstra's runs in $O(n \log n)$ time

Dense Dijkstra and All-Pairs Shortest Path are likely too slow.

Easy mistakes:

- not realizing that the watchman can yell at multiple people from the same cell
- prioritizing the number of remaining waypoints over the distance