Solutions PO finals 2026

1 Duplantis

Duplantis

Authors: Nils Gustafsson, Björn Martinsson

Description

In the following N contests, Duplantis can jump between 0 and a_i centimeters. His current world record is v. Your task is to find r, the maximum number of additional world records he can beat. You should also find the r integers $b_1, b_2, \ldots, b_r$, where b_i is the highest possible final world record if Duplantis sets i more world records.

Duplantis

Description

In the following N contests, Duplantis can jump between 0 and a_i centimeters. His current world record is v. Your task is to find r, the maximum number of additional world records he can beat. You should also find the r integers $b_1, b_2, \ldots, b_r$, where b_i is the highest possible final world record if Duplantis sets i more world records.

Insight 1: In order to beat as many world records as possible, the optimal strategy is to greedily increase the record by 1 centimeter as soon as it is possible to do so.

Duplantis

Description

In the following N contests, Duplantis can jump between 0 and a_i centimeters. His current world record is v. Your task is to find r, the maximum number of additional world records he can beat. You should also find the r integers $b_1, b_2, \ldots, b_r$, where b_i is the highest possible final world record if Duplantis sets i more world records.

Insight 1: In order to beat as many world records as possible, the optimal strategy is to greedily increase the record by 1 centimeter as soon as it is possible to do so.

Insight 2: In order to set a record that is as high as possible while beating exactly k records, the optimal strategy is to set $k - 1$ records as fast as possible, and then pick the maximum a_i afterwards.

Duplantis

```
n,v = map(int, input().split())
a = [int(i) for i in input().split()]
ans = []
for i in range(n):
  if a[i] > v:
    ans.append(max(a[i:]))
    v += 1
print(len(ans))
print(*ans)
```

But this is $O(N^2)$, how to find maximum quickly?

Duplantis

Precompute suffix maximums!

```
n,v = map(int, input().split())
a = [int(i) for i in input().split()]
ans = []
for i in range(n):
  if a[i] > v:
    ans.append(max(a[i:]))
    v += 1
print(len(ans))
print(*ans)
```

But this is $O(N^2)$, how to find maximum quickly?

Precompute suffix maximums! Iterate from $N-1$ to 0 and keep track of the maximum number seen. Put these numbers in a list. Then, you can simply look up the maximum value in this list instead of calling $\max()$ every time.

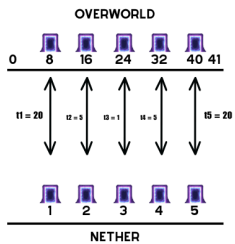
2 Minecraft Speedrunning

Minecraft Speedrunning

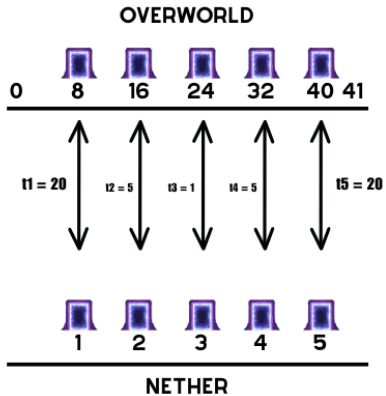
Authors: Harry Zhang, Joshua Andersson

Description

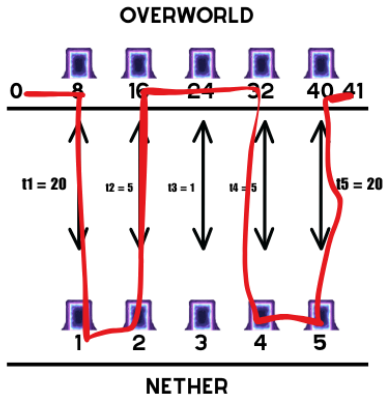
We want to get from 0 to S as fast as possible. There are N portals along the integer number line, that take you to a compressed number line. Portal at coordinate $8x_i$ sends you to coordinate x_i in the compressed number line, and takes t_i seconds to travel by.



Minecraft Speedrunning (ii)



Minecraft Speedrunning (iii)



Minecraft Speedrunning: (N = 2)

We will never walk through the same portal twice.

If there are 2 portals, there are only 2 optimal paths. (Either go through no portals directly to S, or go through the closest portal and walk back out through the furthest away portal.)

We can calculate both costs, and print min.

```
n,s = map(int,input().split())
pos = []
tim = []

for _ in range(n):
    x,t = map(int,input().split())
    pos.append(x)
    tim.append(t)

print(min(s, pos[0]*8 + tim[0] + (pos[1]-pos[0]) + tim[1] + (s - pos[1]*8)))
```

Minecraft Speedrunning: Insight!

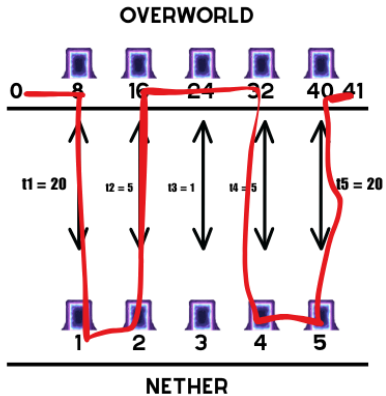
!!!

Minecraft Speedrunning: Insight! (ii)

We will at most walk through portals twice (overworld -> nether and then back).

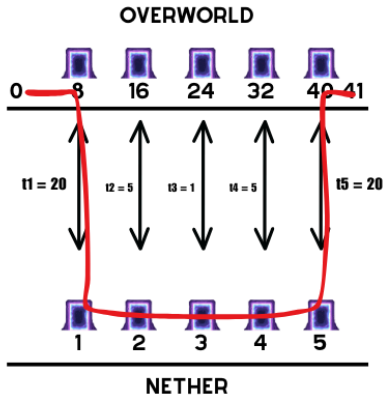
Since distances are much shorter in the nether, we want to travel there as much as possible.

Minecraft Speedrunning: Insight! (iii)



$$8 + 20 + 1 + 5 + 16 + 5 + 1 + 20 + 1 = \text{a lot}$$

Minecraft Speedrunning: Insight! (iv)



8 + 20 + 4 + 20 + 1 = a lot less

Minecraft Speedrunning: $N \leq 2000$

Since we know the "shape" of the optimal solution, we can try all possible such solutions.

For each pair of portals (i, j), calculate the time it would take to enter the portal i and exit portal j.

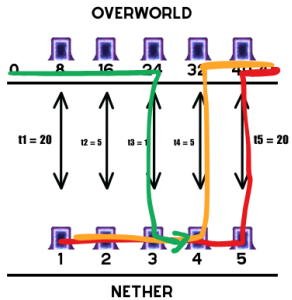
Time: $O(N^2)$

Minecraft Speedrunning: Full Solution

We use the same idea. For each portal i, we are only interested in the portal j that it should exit out of, such that the total time is minimized.

If we answer for each portal from right to left, we notice the following:

Minecraft Speedrunning: Full Solution (ii)



If we want to answer for portal 3, we can either follow the orange path or the red path, and we will just pick the one that is just strictly shorter.

While we are moving along, and want to answer for portal 2, the optimal one out of orange and red will still remain optimal.

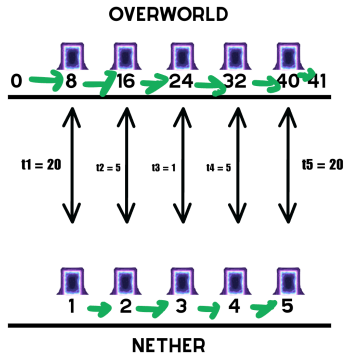
Minecraft Speedrunning: Full Solution (iii)

We can thus always keep track fo the optimal exiting portal, and thus get the following linear solution.

```
ans = s
out = n-1

for enter in range(n-2,-1,-1):
    ans = min(ans,
        pos[enter]*8 + tim[enter] + (pos[out]-pos[enter]) + tim[out] + (s-pos[out]*8))

    if tim[enter] + (s-pos[enter]*8) < (pos[out]-pos[enter]) + tim[out] + (s-
pos[out]*8):
        out = enter

print(ans)
```

Minecraft Speedrunning: Another Solution!

We can represent the problem as a graph, and draw edges between adjacent portals. The weights on the edges between portals are the distances between the portals.



In this model, we want to find the shortest path from 0 to S. This can be found using Dijkstra's algorithm, which runs in $O(N \log N)$, since the number of nodes and edges is this graph is $O(N)$.

3 Island Counting

Island Counting

Author: Joshua Andersson

Description

Every country has islands of different areas. If we only count islands whose areas are $\geq A$, which count has the most islands? Answer Q such question.

Island Counting: $O(TQ)$

For each question, iterate through all islands and count those that are big enough. Finally, output the country with most islands.

```
for each question:
  country_island_count = [0] * n
  for (island_size, country) in island:
    if island_size >= A:
      country_island_counts[country] += 1
  print(country with most islands)
```

Island Counting: $A \leq 10$

There are at most 10 different queries, so compute the answer for each one and save the answers. We can then answer queries in $O(1)$.

Island Counting: $N \leq 25$

For each country, create a list of all island sizes. Sort this list. We can count the number of islands that are big enough in this island with a binary search: find the index of the last value $\geq A$. Using this index, we can count the number of islands that are big enough in $O(1)$. Do this for each island for each query. $O(QN \log(T))$

Island Counting: full solution

We will sort all queries by A in decreasing order, and then answer all of them at once.

Maintain the list `country_island_count`. To answer the largest query, A_1, add all islands that are $\geq A_1$ to `country_island_count`. Also maintain the current best country and update it whenever we add to `country_island_count`. The next query will have smaller value, so we can reuse the work done so far. If the first query has value A_1 and the second has value A_2, islands with size x where $A_2 \geq x < A_1$. If we do this carefully, avoiding redoing work, we get $O(T)$ work total.

$O(Q \log(Q) + N \log(N))$.

Implementation

To implement, I would recommend you to work with a list of all pairs (island size, country index), sorted in decreasing order by island size. Keep an index into this list.

```
j = 0
for query_a, query_index in queries:
  while j < len(islands) and islands[j][0] >= query_a:
    add island j
    j += 1
  answer[query_index] = ...
```

4 Mushrooms

Mushrooms

Author: Joshua Andersson

Description

You have an array A, where $A[i] = 0$ initially. You are given a target array W. Make $A = W$ in few rounds. In each round, you select an operation for each mushroom to perform, and they all then perform their operations at the same time.

Mushrooms: operations

In an operation on element i, you can set $A[i]$ to one of

- $P[i]$
- $P[j] \ll 1$
- $P[j] \gg 1$
- $P[j] + 1$
- $P[i] \hat{} P[j]$
- $P[i] \& P[j]$
- $P[i] \mid P[j]$

$N = 512$, $0 \leq W[i] \leq 255$. Use at most 9 rounds for full score.

Mushrooms: 10 points

In each round, we add 1 to $A[i]$ if $A[i] < W[i]$, otherwise we pass.

Takes at most 255 rounds.

Mushrooms: 46 points

Addition is slow

We need to create large numbers quickly. Only two operations really allow us definitely increase the numbers: + and <<. What does << do? It x<< 1 is equivalent to x*2.

Using this, we can quickly grow large numbers:

- Start: 0
- Round 1: 1 (+ 1)
- Round 2: 2 (<< 1)
- Round 3: 4 (<< 1)
- Round 4: 8 (<< 1)
- Round 5: 16 (<< 1)
- Round 6: 32 (<< 1)
- Round 7: 64 (<< 1)
- Round 8: 128 (<< 1)
- Round 9: 256 (<< 1)

Mushrooms: 46 points

We can quickly build powers of 2. This suggests we should look at out number in binary. How can we build our number using a sequence of `*2` and `+1`?

We can use the same idea as in The Calculator (PO Skolkval 2025): let's build our number in reverse! We can thus do `-1` or `/2`: however, we can only do `-1` if our number is odd, and `/2` if it is even. Thus, at any point in time, our choices is forced! Build every number independently.

- 13=1101 (odd -> −1)
- 12=1100 (even -> /2)
- 6= 0110 (even -> /2)
- 3= 0011 (even -> /2)
- 3= 0011 (odd -> −1)
- 2= 0010 (even -> /2)
- 1= 0001 (even -> /2)
- 0= 0000 (odd -> −1)

We can then reverse this again to get our solution. Worst case is 255, taking 15 operations.

Mushrooms: honorable mention, ~50 points

Model as a shortest path from state $(A[1], A[2], ..., A[N])$ to $(W[1], W[2], ..., W[N])$. Too slow, so split array into blocks of size 2 or 3 and solve optimally.

Mushrooms: 70 points

Designate a single mushroom as the "grower". It will go through every power of 2 up to 128. Using the grower, build all numbers 0 to 255 in 9 rounds.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
4 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0
8 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 0
...
```

In one round, we can fix the rest. How? XOR! Suppose $A[i] = k$ and we want $W[i]$. We want to solve the equation

- A[i] ^ x = W[i]

Xor is its own inverse: A ^ x ^ x = A

- x = A[i] ^ W[i], a number between 0 and 255

Mushrooms: 70 points (ii)

If we have built numbers 0 to 255, we can fix every number in one round by XORing with an appropriate other number.

Mushrooms: 100 points

In 8 rounds, we can grow numbers 0 to 128. We will let every mushroom grow its lowest 6 bits (0-127), and the grower will become 128. In the final round, every A[i] ORs with the grower if A[i]>=128. This is guaranteed to fix every number but the grower. Will we be able to fix the grower? Not always as-is. We need to choose the grower cleverly.

Find some pair x,y where W[x]==W[y]. This pair must exist by the pigeonhole principle. Let v be the value.

- If $v \geq 128$: let x be the grower. Last round, we can fix x by A[x]=A[x] | A[y], and y by A[y]=A[y] | A[x]
- If $v \leq 128$: let x be the grower (will go up to 128). y will grow W[y]−1. Last round, set A[x]=A[y]+1 and A[x]=A[y]+1.

I also have a solution that only uses A[i]=A[i], A[i]+=1 (don't add another), A[i] = P[j] << 1 and A[i]^=P[j]. We can also solve It in 9 rounds with $N = 255$ and $0 \leq W[i] \leq 255$.

5 Grinding

Grinding

Author: Joshua Andersson

Description

There are N caves, each of which has a number of monsters. When you attack a cave, you fight the monsters one by one. Each monster has a strength and xp. If you defeat a monster, your strength increases by their xp. In order to attack cave i, you must first have defeated cave $i - 1$. What is the minimum number of attacks needed to gain at least B strength?

Grinding: DP/BFS for partial score

We can do a BFS with state (caves unlocked, attack). We then try every cave. $O(NBT)$: $O(NB)$ states, and each takes $O(T)$.

To optimize it, we can realize that we are doing repeated work: we can precompute how far we get in each cave for each possible B in $O(T)$ time. If we do slightly more precomputation, we get $O(BT + BN)$.

Grinding: full solution

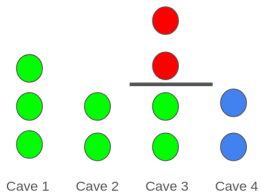
Main greedy idea: We want to choose attacks that give as much XP as possible most of the time. Sometimes we want to defeat the last available cave in order to unlock the next one.

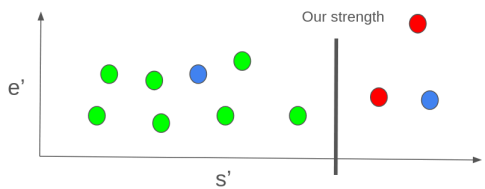
Let's fix c, the final cave that we want to reach. Now it is optimal to always choose the attack that gives the most XP, unless we haven't reached cave c yet and we can defeat the last available cave.

The full solution will be more or less this idea, but we need some tricks to make it fast enough.

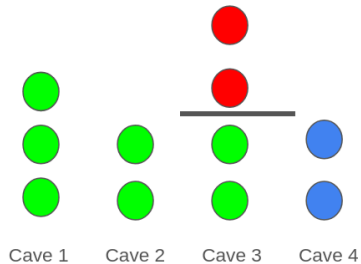
Grinding

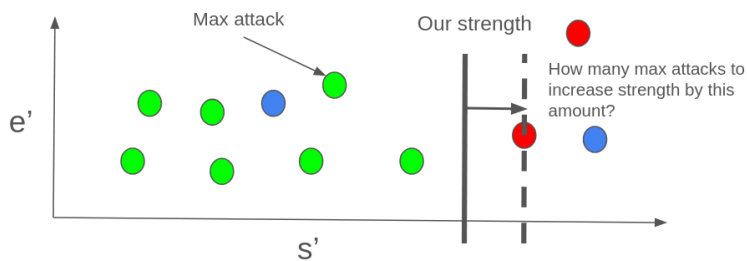
For each monster, instead of keeping track of s and e, we keep track of s' and e', where s' is the strength you need to defeat the monster *when entering the cave*, and e' is the experience you gain after defeating this monster and the ones before.



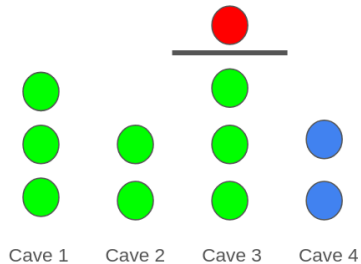


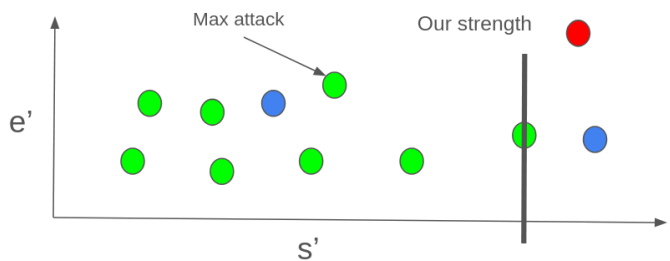
Grinding


Cave 1
Cave 2
Cave 3
Cave 4


Max attack
Our strength
How many max attacks to increase strength by this amount?
e'
s'

Grinding





Grinding

Keep the max attack value in a variable. Do some math to find how many max attacks are needed to unlock the next monster.

If we fix the final cave, and repeat this process, the solution is still quadratic. But we do the same simulation every time. So instead, every time we unlock a monster, find the what the answer would be if this was the final unlocked monster (do some math). Some monster has to be the final one, and getting to it as fast as possible is the optimal way to getting a strength of B as fast as possible.

The solution runs in $O(N)$.

6 Circlemerge

Circlemerge

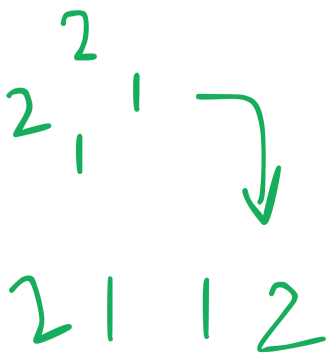
Author: Emil Sandberg, Implemented by: Harry Zhang and Joshua Andersson

Description

Given N integers in a circle. You are allowed to merge two adjacent numbers by adding them together. What is the minimal number of merges such that all numbers in the circle are the same?

Circlemerge: First number is never touched

If the first number is never touched, we can view it as a line.



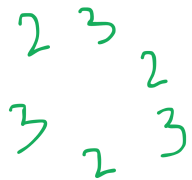
Problem turns into, to count the number of independent intervals of a_1 are there?

Because of the first subtask, the answer will just be $N - \frac{\sum a_i}{a_1}$ (we will remove everything but the remaining $\frac{\sum a_i}{a_1}$ numbers).

Circlemerge: $N \leq 100$

Continuing on the previous idea, we know that in an optimal solution, the final number will consist of some consecutive interval of the original array.

We can try all $O(N^2)$ possible intervals, and turn it into a line that we try to solve. For each interval, we know the final sum, so we check if that sum is valid. Then we take the minimum over all such possible answers.



Each check is done in $O(N)$, resulting in an $O(N^3)$ solution.

Circlemerge: $N \leq 3000, a_i \leq 200$

Let $S = \sum a_i$ be the sum over all integers in the original circle.

We know that the final number will be divisor of S.

Since $S \leq 6 \times 10^5$, the number of divisors of S is less than 128. We find all these divisors in $O(S)$.

For each divisor d of S, we want to check if the final number in the circle is d.

The check can be done in $O(N^2)$, by cutting the circle up into a line in all possible $O(N)$ ways, we can check if it is possible to cover any line with intervals of sum d.

This results in a $O(S + 128N^2)$ solution.

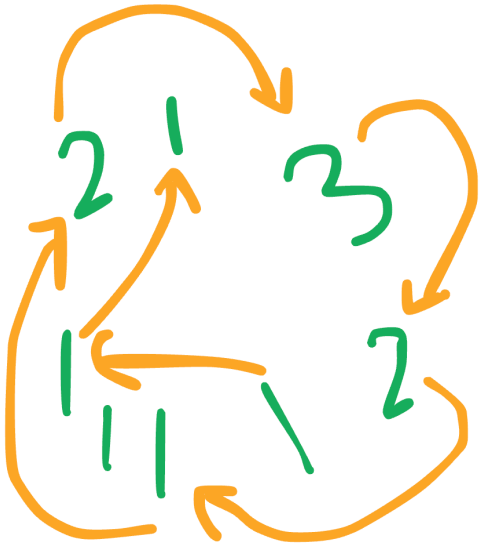
Circlemerge: $a_i \leq 200$

Since $S \leq 4 \times 10^7$, the number of divisors of S is less than 448. We find all these divisors in $O\left(\sqrt{S}\right)$, by checking if j or $\frac{S}{j}$ is a divisor for all $j = 1, ..., \sqrt{S}$.

What remains is to speed up the check for each d.

Fix some divisor d. We can for each element i in the circle, find the next element to the right j such that the sum of elements between i and j is exactly d.

Circlemerge: $a_i \leq 200$ (ii)



Circlemerge: $a_i \leq 200$ (iii)

With this model, we only need to check if there exists any cycle in this graph. This can be done in $O(N)$ with a DFS, or using Union Find (Because of some special properties with this graph.)

The results in a solution that runs in $O\left(\sqrt{S} + 448N\right)$.

Circlemerge: Wait what, union find?

Consider the edges as bidirectional. We will use Union Find if there exists a cycle.

Because we are only looking at divisors d, if we start on a node and travel along edges, it has to end at the same node itself IFF this node was in the cycle.

In this case, Union Find will actually be faster than DFS (by some constant factor).

Circlemerge: Full solution

In general, we have that $S \leq 10^{18}$, which could have up to 103680 divisors. There are methods that can find all of these divisors quickly, but we can use one nice fact:

We are only interested in factors d such that $\frac{S}{d} \leq N$. Since if d is the remaining numbers in the circle, there will be $\frac{S}{d}$ such numbers. But the number of elements in the circle can never exceed N.

Thus we can find all relevant divisors in $O(N)$, by looping over $\frac{S}{d}$ from 1 to N.

Circlemerge: Full solution (ii)

There are up to 6000 divisors of $S \leq 10^{18}$ such that $\frac{S}{d} \leq 2 \times 10^5$.

If we run an $O(N)$ check for each of these divisors, it will be too slow. O($6000N$) is a lot!

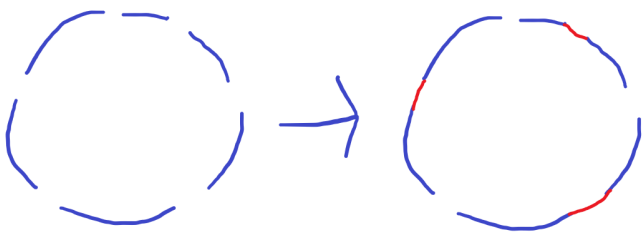
What can we do now? Can we skip checking any of these divisors?

Circlemerge: Full solution (iii)

Let $f(p)$ be 1 if there is a way to merge the circle such that there are p intervals, otherwise 0. The problem is then to find the largest p where $f(p) = 1$.

First, if $f(10) = 1$, then we don't need to check $1, 2, 3, 4, 5, 6, 7, 8, 9$: we want to find the largest p, and these are all smaller.

Second, if we can split the circle into 6 parts, we can split it into 3 parts by gluing together the parts.



In general, if $f(p)$ holds, then $f(d)$ holds for all divisors of p. Taking the contrapositive, if we $f(d)$ doesn't hold, then $f(d * k)$ doesn't hold for all k.

Circlemerge: Full solution (iv)

For a given divisor d, no matter if $f(d) = 0$ or $f(d) = 1$, we can always disregard the result of some other divisors with high probability.

By shuffling the relevant divisors and prune out other divisors based on rules from the previous slide, we get a fast enough solution. With shuffling, we only have to check at most 600 divisors in the worst cases with high probability.

The analysis of why shuffling is fast can be argued based on the DAG generated by all the factors of S.

Circlemerge: Full solution (v)

The analysis of why shuffling is fast can be argued based on the DAG generated by all the factors of S.

If we only have 1 unique prime in S, the DAG will be a single chain of length $l = O(\log S)$. By doing random shuffle, we will find the lowest desired divisor in $O(\log l) = O(\log \log S)$ checks.

With several unique primes in S, we can bound the number of checks naively by the minimum chain cover of the divisor DAG, which is equivalent to the longest anti-chain of the DAG by Dilworth's theorem. By picking a random divisor out of all divisors, can be seen as picking a random chain out of the minimum chain cover, and over that chain pick a uniformly random divisor.

This bound is still a bit high, but what we can realize is that when we query a divisor d, the divisors we can prune out are all on ALL possible chains that goes through this d.

Because of how the DAG is based on divisors of an integer, the DAG will be very interlaced with the chains/paths.

Thus, randomly choosing divisors and checking for those, should on average prune a lot of the other divisors that exist.

7 Tina Scoreboard

Tina Scoreboard

Description

We run a resolver on a contest similar to PO. Every problem is worth 1 point, and there are no subtasks. If you solve a problem, you get 20 minutes of penalty for every incorrect submission to the problem. The order that submissions are revealed is given. Given that people know their own results during the freeze, determine for every person the point in time when they definitely know their final rank.

Tina scoreboard: some insights

Every person's score is (solves, penalty). How does a person P know when their final rank is determined? For every other team, compute the best and worst possible score. If the final rank of P would change depending on whether the other team gets the lower or upper, P cannot be sure of their final rank. If there is no such other team, P has determined their final rank.

Tina scoreboard: $O(N^2F^2)$

Write functions lb(P) and ub(P) that computes the upper and lower bound of a team.

```
for submissions in frozen:
  process submission
  for person in people:
    any_covers = False
    for other in people:
        if person==other: continue
        lower = lb(other)
        upper = ub(other)
        # check if the other person covers me

    if !any_covers:
      person is done
```

Tina scoreboard: $O(N^2F)$

The lower and upper bound doesn't change that often. Keep track of lb(P) and ub(P) for every person, and only change it for the affected person after processing a submission.

Tina scoreboard: $O(NF\log(F))$

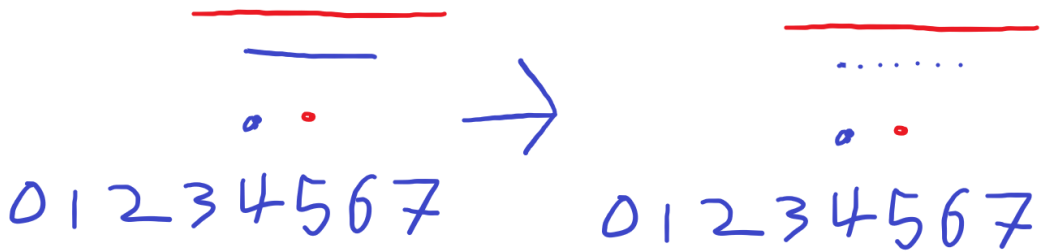
It's annoying that scores are 2D. Let compress(s, p)=$s * \text{BIG} - p$ where BIG is sufficiently large ($20F$ suffices). These compressed values compare the same as the original pairs. Thus, every pair lb, ub is now an interval. With this, our problem is

Reduced problem

We have N points and intervals. We will gradually shrink intervals, and we want to determine for each point which shrink causes it not to be covered by any other interval (not counting their own).

Tina scoreboard: $O(NF\log(F))$ example

In the following image, each color is a team. The dot is their final score, and the interval is their uncertainty interval. The submission of the blue team is revealed to be WA, thus reducing their interval to a point. With this, red is not covered by the uncertainty interval of any other team and is thus done. The reason for this is that even if red is covered, it is only covered by the uncertainty interval of itself.



Tina scoreboard: $O(NF\log(F))$

Ooga booga solution: build a lazy sparse segment, where the indices are compressed values. For every interval, add +1 to every number in it. Then,

```
for submission in frozen:
  process submission
  for person in people:
    add -1 to my interval (I'm not covered by myself)
    check the sum at my final score: if 0 I'm not covered, otherwise I am
    add +1 to my interval (add back)
```

Easier to code solution: binary search the answer for every person.

Tina scoreboard: map

For full points, we can process submissions in reverse. Then, our problem is simply that we have points and intervals, and intervals grow. For each point, we want to find the grow that first covers it. Store a map of people's final score, and then do a binary search to iterate over people in the interval, removing them if covered (take care to not allow your own interval growing to kill yourself).

Tina scoreboard: lazy segment tree

For every position in compressed space, we want to compute the first interval that covers it. Then, for every team we can simply print the value at their final score. We can simply use a lazy sparse max segment tree. Caveat: we don't count our own interval. So we compute first and second earliest interval covering each point, so we can make sure we don't count ourself.

Tina scoreboard: parallell binary search

We can binary search to solve an individual person. Each binary search evaluation takes $O(N)$ time, for $O(N^2 \log(N))$ total.

We can optimize this by doing all binary searches in one big batch. This technique is known as parallell binary search: https://robert1003.github.io/2020/02/05/parallel-binary-search.html.