Lab 9 - Graph Traversal

CS2040S Data Structures & Algorithms

AY20/21 Semester 1

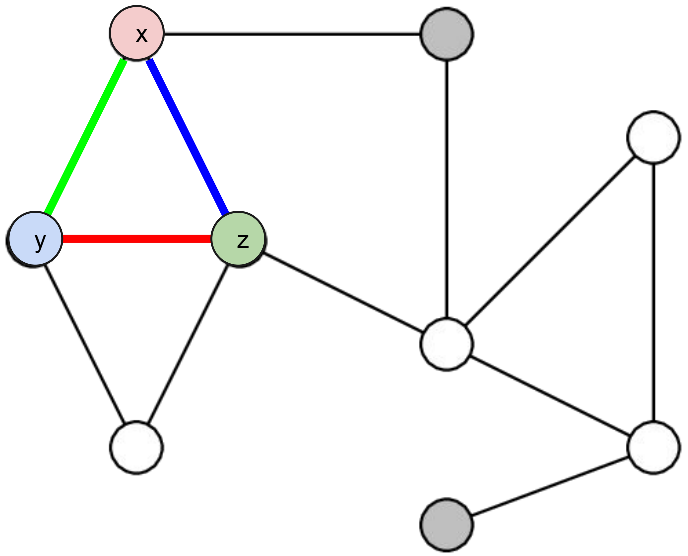
Week 11

One-Day Assignment 7 - Weak Vertices

- Given T (≤ 100) undirected unweighted graphs
    - Each graph is given in adjacency matrix form
        - N ≤ 20 vertices per graph
    - Print out all “weak” vertices in ascending order
        - Vertex not part of any triangle => “weak”

Triangles

- (x,y,z) is triangle in graph
  - ⇔ (x,y), (y,z), (x,z) exist
  - ⇔ We have the length-3 cycle
    - $x \to y \to z \to x$
- For any vertex x,
  - How to find if some triangle contains x?
  - What kind of graph DS operations do we want?
  - How fast can we answer that?



One-Day Assignment 7 - Weak Vertices

- Constraints:
  - Number of graphs: up to 100
  - Number of vertices per graph: up to 20

Can I try out every possible combination?

One-Day Assignment 7 - Weak Vertices

- Constraints:
  - Number of graphs: up to 100
  - Number of vertices per graph: up to 20
- V^3 algorithm to select all possible 3-vertex combinations
  - 20^3 = 8000
- Up to 100 graphs
  - 8000 x 100 = 800k <= 100 million
  - Runs in time!

One-Day Assignment 7 - Weak Vertices

- Which Graph DS to use?
    - You need to check for existence of edge many times
    - Adjacency Matrix
        - O(1) to check if edge exists between two given vertices
        - Question already provides the graph in this form

Solutions

1. For each graph:
   - Mark all vertices as weak
   - For vertex i in graph
     - For vertex `j` in graph
       - For vertex k in graph
         - Check if edge exists between `i` - `j`, `j` - k and k - `i`
         - If yes, mark vertex `i`, `j` and k as strong

One-Day Assignment 7 - Weak Vertices

- Note that it is easier to keep track of “Strong Vertices” rather than “Weak vertices”
- Finding a triangle guarantees that the three vertices are strong
- Having no triangle between the given 3 vertices does not guarantee that the vertices are weak
    - They may be part of other triangles

Graph Traversal

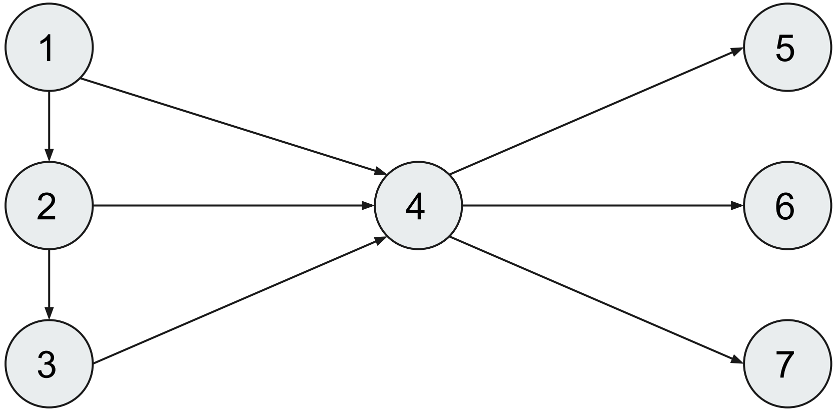
- Two main method of Graph Traversal for finite graphs
    - Breadth-First Search
    - Depth-First Search
- Pseudocode in lecture notes, we’ll cover implementation caveats

Graph Traversal - BFS

- Breadth-First Search (BFS)
    - Tends to employ a queue
    - Keep track of visited vertices
    - **When to mark a vertex as visited?**
        - Let’s try marking as visited only when a vertex is removed from the queue

Graph Traversal - BFS

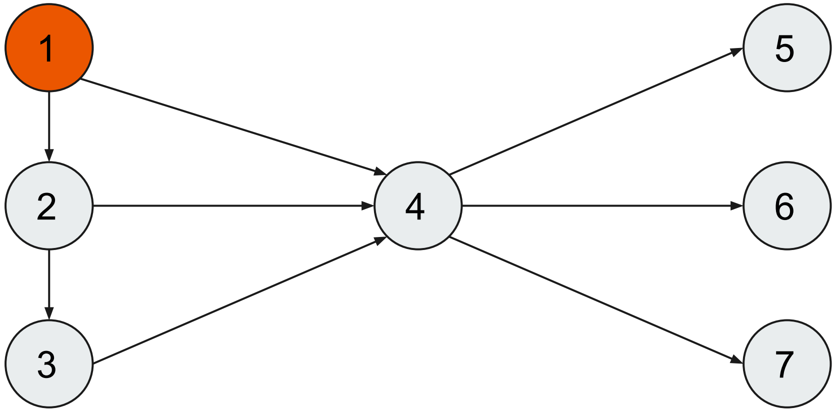
Begin BFS from 1



Queue
Node 2
Node 4

Graph Traversal - BFS

Current vertex



Queue
Node 4
Node 3
Node 4

Duplicate in the queue!

Graph Traversal - BFS

- Same vertex is enqueued into the queue again!
  - Slows down your program
  - Worse as the graph gets more connected
- Mark a vertex as visited as you enqueue it

Graph Traversal - DFS

- Depth-First Search (DFS)
    - Tends to employ a implicit stack via recursion
    - May result in running out of computer memory that OS provides
        - Can use iterative DFS using an explicit stack instead

Traversing Multiple CCs

- Single call of BFS/DFS may not cover whole graph
    - If graph disconnected, it will only visit one of the CCs
- Solution:
    - Try starting from every vertex

- If we retry from every single vertex
    - Re-visiting a CC wastes time
- Solution:
    - Keep a **global** visited array
    - Shared between all search calls

```
For start in [1...V]
    If not Visited[start]
        Search(start)
```

One-Day Assignment 9 - Islands

- Given a map of `r` rows and `c` columns ($1 \le r, c \le 50$), find the minimum number of islands possible
- Each cell can represent land (`L`), water (`W`) and cloud (`C`)
- Island is defined as region of land that is connected to every other by some path, only in 4 directions (up, down, left, right)

One-Day Assignment 9 - Islands

w	w	w	w	w
w	w	L	w	w
w	L	L	L	w
w	w	L	w	w
w	w	w	w	w

single Island

w	w	w	w	w
w	L	W	L	w
w	W	L	W	w
w	L	W	L	w
w	w	w	w	w

5 Islands

One-Day Assignment 9 - Islands

- How should we store the graph?
  - Each coordinate is a vertex
  - Each vertex has an edge to the 4 adjacent vertex
- **You can treat the map itself as the graph!**
  - Vertex (x, y) has an edge to vertex (a, b) if their coordinate value differ only by one
  - (3, 2) has an edge to (2, 2), (4, 2), (3, 1) and (3,3)

One-Day Assignment 9 - Islands

- Curveball: Cloud can be either Land or Water
- We want to find the minimum possible number of islands on the map
- What should the clouds be?

Assignment Guidelines

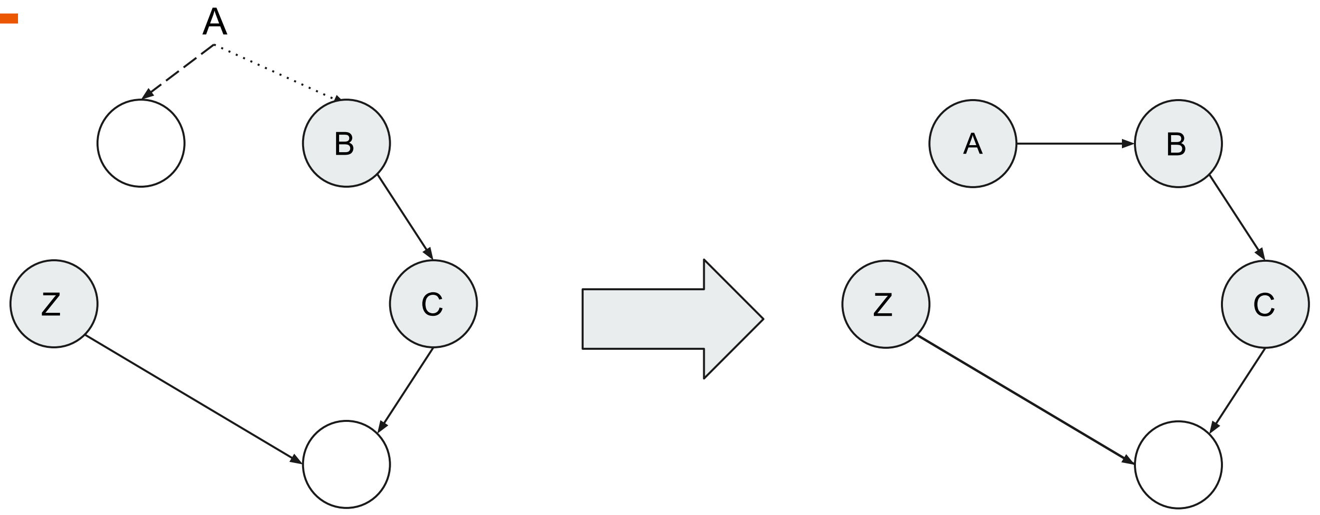
- Include your **name** and **student number** in comments at the top of your code.
- You are allowed (and encouraged) to discuss algorithms
    - List down all your collaborators in your source code
- **You are NOT allowed to:**
    - **Copy another person’s code**
    - **Look at another person’s code**
    - **Use another person’s code as a base for your own code**
- Plagiarism checks will be in place

Take-Home Lab 3 - Ladice

- N items, L drawers
  - Each item has 2 allowed positions A_i, B_i
- Try to insert each item
  - If successful, print LADICA
  - If discarded, print SMECE

1. Try position A_i.
2. Try position B_i.
3. Try repeatedly pushing the item already at A_i, to its other positions, until a free space is reached.
   If we loop, continue to next rule.
4. Try repeatedly pushing the item already at B_i, to its other positions, until a free space is reached.
   If we loop, continue to next rule.
5. Discard the item.

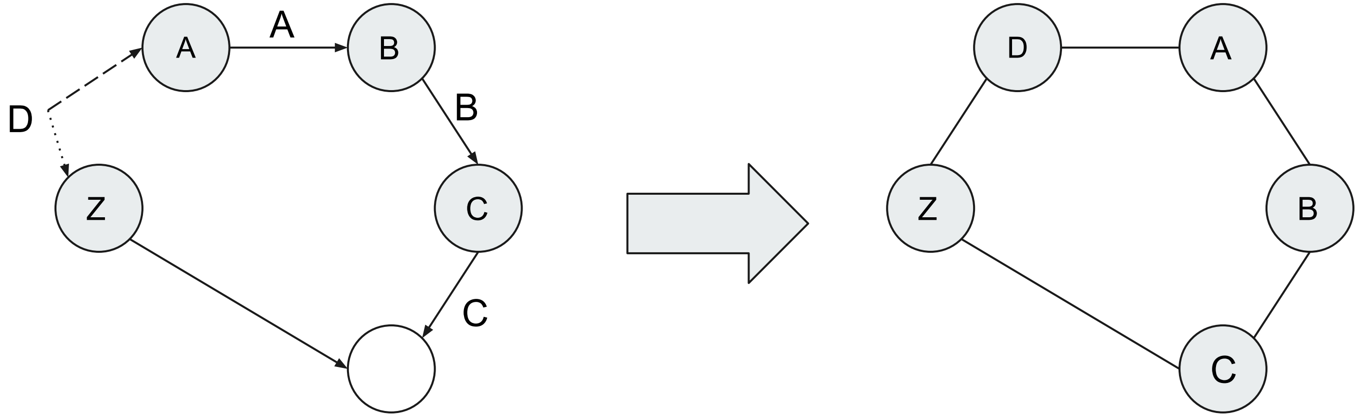
Pushing Stuff Back



Can we track where we can “push” items to make space?

Not full yet, still have empty drawer
Can still push stuff

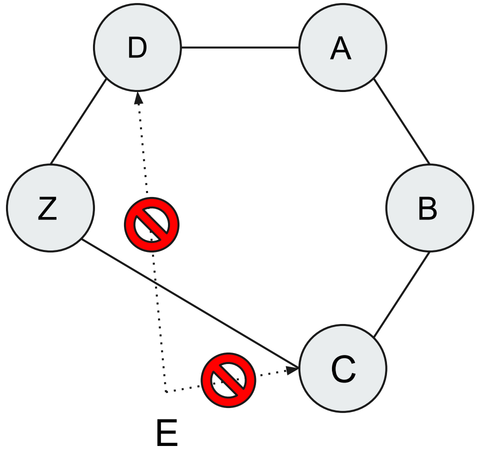
Pushing Stuff Back, Redux



Not full yet, still have empty drawer
Can still push stuff

Full, we can't push forward/backward
any of these items.

Pushing Stuff But Failing

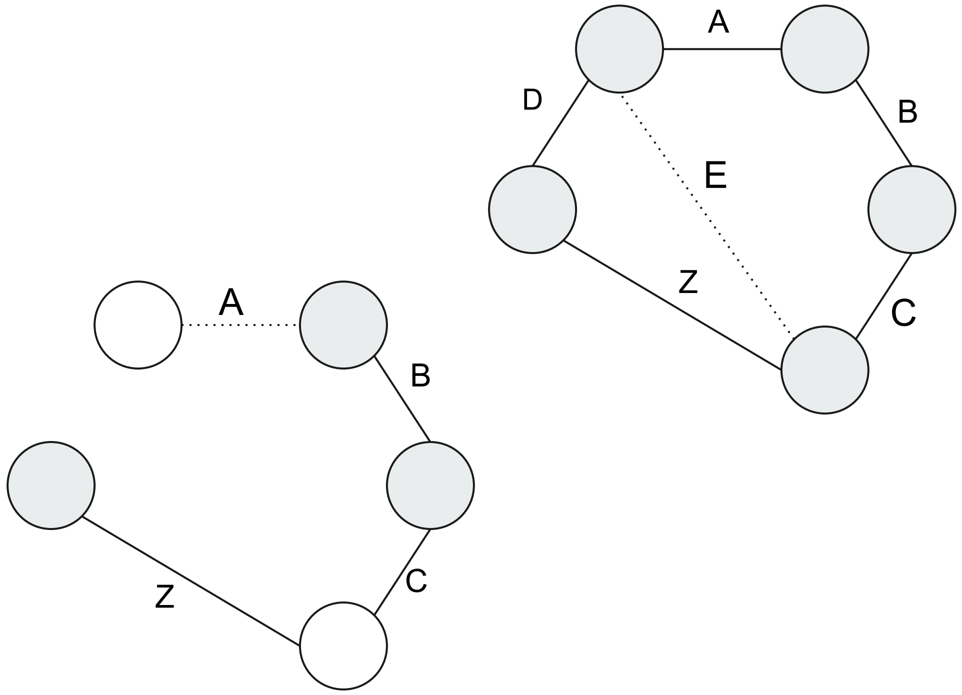


When is a “bunch” of drawers **full**?
(i.e. cannot push to make space?)

Here, drawers holding {A,B,C,D,Z} are full.

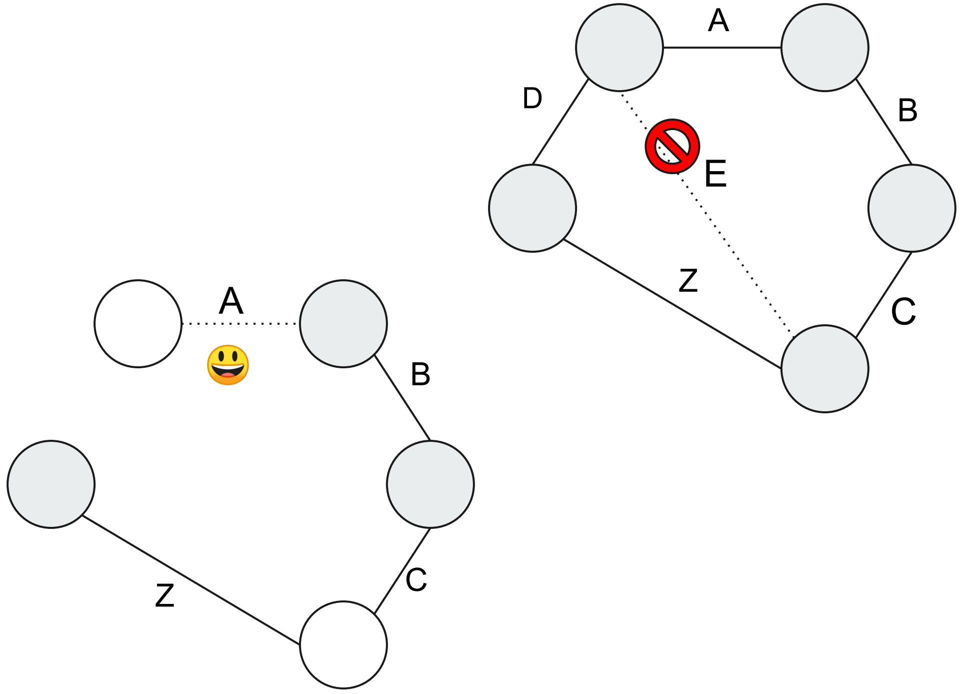
NEW: In hindsight, these look like a graph...

Key Insight: Actual Positions Don’t Matter



- Item positions don’t actually matter!
  - As long as there is free space
  - Possibly after pushing back a chain of items.
- Items only have **2 valid positions**
  - Can “slide” back and forth between these 2 drawers
  - We want to push entire chains of items at once
- Model as graph
  - Vertices: Drawers
  - Edges: Items

Key Insight: Model as Graph



- Model as graph
  - Vertices: Drawers
  - Edges: Items
- Connected component ("bunch")
  - Still have space if:
    - K vertices (drawers)
    - K-1 edges (items)
    - **Tree!**
  - No more space if:
    - K vertices (drawers)
    - K edges (items)
    - **Contains 1 cycle!**

Graph and UFDS

- Equivalent ideas:
    - UFDS node
        - Graph vertex
    - UFDS disjoint set
        - Graph connected component

- Find(a)
    - Find **representative** for a’s connected component
- Union(a,b)
    - AddEdge(a,b)

TAKEAWAY

UFDS can be used for the **dynamic connectivity problem** for graphs.

- Connectivity => Test if (a,b) are **currently** connected.
- Dynamic => Can add edges on the fly
    - UFDS doesn’t support remove!

Cycle Detection

- How do we know when a “bunch of drawers” is full?
    - Connected component has |vertices| = |edges|
    - We completed a cycle!
- When does AddEdge/Union(u,v) make a cycle?
    - When u and v are already connected.
    - i.e. Already have Find(u) == Find(v)
- Slight modification of UFDS code

Union

```
void union(int a, int b) {
        int x = find(a);
        int y = find(b);
        if(x == y) return;

        if(rank[y] < rank[x]) {
                parent[y] = x;
        } else {
                if(rank[x] == rank[y]) rank[y]++;
                parent[x] = y;
        }
}
```

```
boolean union(int a, int b) {
        int x = find(a);
        int y = find(b);
        if(x == y) return true;

        if(rank[y] < rank[x]) {
                parent[y] = x;
        } else {
                if(rank[x] == rank[y]) rank[y]++;
                parent[x] = y;
        }
        return false;
}
```

Union(a,b) returns true if a,b were already connected.
(i.e. union-ing failed)

Ladice

TAKEAWAY

UFDS can be used for the **dynamic connectivity problem** for graphs.

- Connectivity => Test if (a,b) are **currently** connected.
- Dynamic => Can add edges on the fly
    - UFDS doesn't support remove!

```
1. Make UFDS of N drawers/vertices
2. IsFull = new boolean[N]
3. For each item in input, read positions [a,b].
    a. If IsFull[Find(a)] && IsFull[Find(b)], reject.
    b. WillBeFull = IsFull[Find(a)] || IsFull[Find(b)]
    c. If Union(a,b) || WillBeFull
        ● If Union returns true, we completed a cycle in the CC containing a & b.
        ● Otherwise, they were separate. If one was full, combined CC is full.
        i. IsFull[Find(a)] = True
```

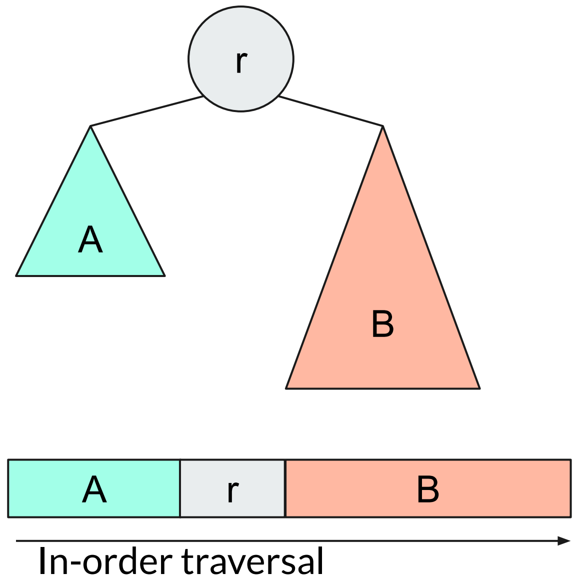
Take-Home Lab 3 - Factor-Free Tree

- A **factor-free** tree is a binary tree, with natural numbers at each node
  - Any node value is coprime to all of its ancestor node values
- Given an **in-order** traversal of this binary tree
  - Reconstruct a valid factor-free tree
  - Does not have to be the original, any valid is OK

Factor-free trees are made just for this problem, they are not a 2040S DS.

Tree Construction

- Property of in-order traversal:
    - [left subtree] (root) [right subtree]
- Maybe we can borrow a page from the perfect binary tree construction.
    - Find an *appropriate* root (position *r*)
    - Split and recurse on left/right halves
        - Produce left/right subtrees

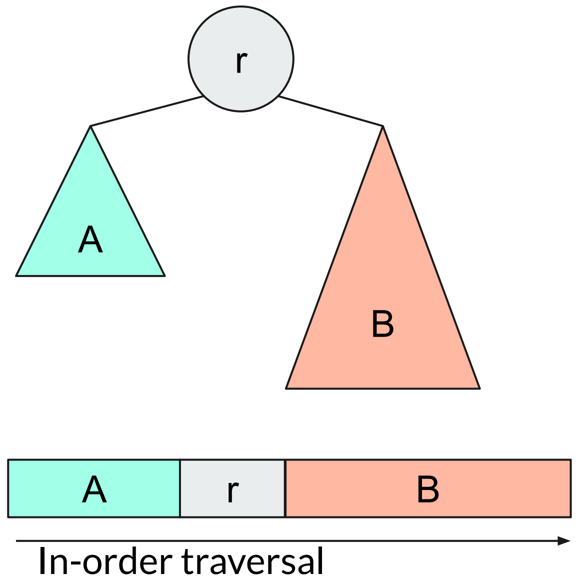


Tree Construction

- Can we find **the original** root (if any)?
- Can we find **a possible** root?
    - Root has to be coprime with all its descendants

IDEA: Black-box the CanBeRoot check.

- Pretend CanBeRoot is given to us.
    - “Oracle”/“Black-box”
    - We don’t know/care how it works for now
- Can query it for the answer in 🍕
    - Hopefully 🍕 is fast
        - (e.g. constant or log).



Guessing the Root?

- If possible, guaranteed to have at least 1 tree
  - The original tree
- What happens if at some subarray [L,R]
  - We choose a different root than the original tree?
  - Is it possible to fail later? (i.e. we made a bad choice)

Safe!

- If we failed, that means at some subarray [L, R]
  - We cannot find a possible subtree root
- We try following a path in the original tree
  - All original subtrees/subarrays fully covering [L,R]
- At the lowest subtree still fully covering [L,R]
  - Original choice splits [L,R], as children do not cover [L,R]
  - Original choice must be inside [L,R]
    - Coprime to rest of [L,R] and possibly more
  - We still have that choice. Contradiction!
- **Upshot: Guessing a possible root will NOT doom us later.**

3230: This is an **exchange argument** proving **greedy optimality.**

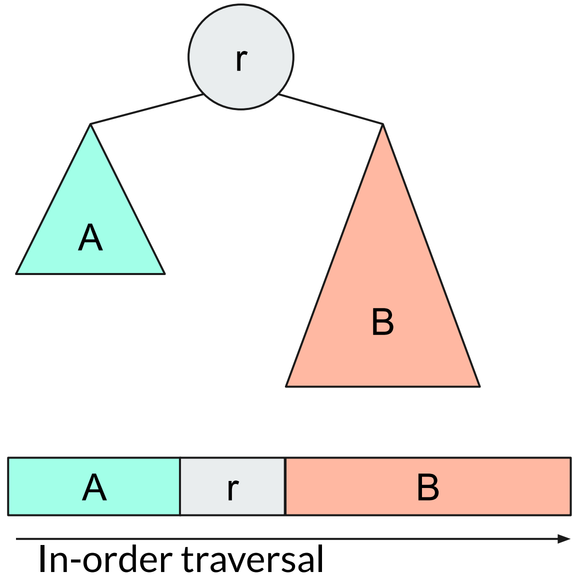
Tree Construction

K = Number of elements in subarray
 = R - L + 1
D = Distance of chosen root, from start
T(K) = D*🍕 + T(D) + T(K-D)

```
// Try to tree-ify subarray [L,R]
// Return true iff possible.
Treeify(L, R, previousRoot)
     if( R < L )
          Subarray is empty, return True.
     For each position x in [L,R]
          if CanBeRoot(x, L, R)     🍕 repeated D times
               Parent[x] = previousRoot
               Return Treeify(L, x-1, x) && Treeify(x+1, R, x)
                              T(D)                 T(K-D)
     // If we reach here, we failed to find a root.
     Return False.
```

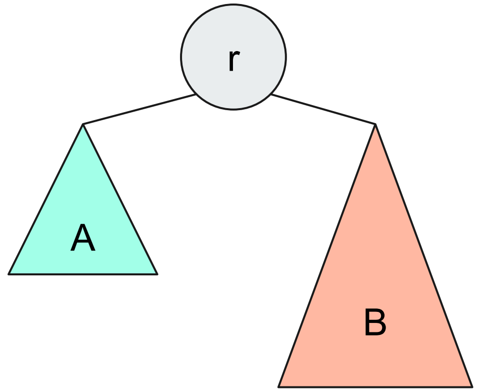
Tree Construction

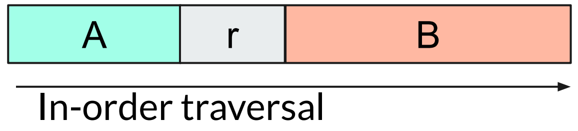
- What's our recurrence relation?
    - T(K) = 🍕 * D + T(D) + T(K-D)
    - O(1) work + *uneven* split => O(🍕 n) total
        - Very good!
    - O(n) work + even split => O(🍕 n log n) total
        - Not too bad
    - O(n) work + *uneven* split => O(🍕 n^2) total
        - How did we end up here?
        - Can we avoid this case?

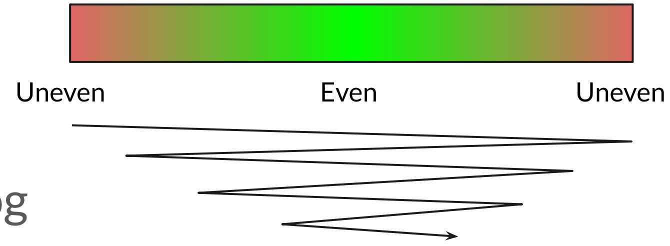


Tree Construction

- We want to avoid two bad things at once:
  - Slow work
    - Root search visits this last.
  - Uneven split
    - Root chosen near end.
- At most 1 still OK.
- IDEA: **Zigzag, starting from both ends**
  - Handle the uneven split **early**
    - Fast work + uneven split ⇒ O(🍕n)
  - Handle the even split **late**
    - Linear work + even split ⇒ O(🍕n log n)







(Mostly) Formal Proof by Induction

Let k be **length of <u>sub</u>array**.
Let d be **minimum distance to either end**.

$$1 \le d \le k/2$$

Assume $T(k) \le 2🍕\ k \log_2 k$, for smaller values of k.

$$\begin{aligned}
T(k) &\le \min\_\{1 \le d \le k/2\} && 2d🍕 + T(d) && + T(k-d) \\
&\le \min\_\{1 \le d \le k/2\} && 2d🍕 + 2🍕\ d \log_2 d && + 2🍕\ (k-d) \log_2 (k-d) \\
&= \min\_\{1 \le d \le k/2\} && 2🍕 [d + d \log_2 d && + (k-d) \log_2 (k-d)] \\
&\le \min\_\{1 \le d \le k/2\} && 2🍕 [d + d \log_2 (k/2) && + (k-d) \log_2 (k-d)] \\
&= \min\_\{1 \le d \le k/2\} && 2🍕 [d + d (\log_2 k - 1) && + (k-d) \log_2 (k-d)] \\
&= \min\_\{1 \le d \le k/2\} && 2🍕 [d \log_2 k && + (k-d) \log_2 (k-d)] \\
&\le \min\_\{1 \le d \le k/2\} && 2🍕 [d \log_2 k && + (k-d) \log_2 k] \\
&= \min\_\{1 \le d \le k/2\} && 2🍕 k \log_2 k \\
&= 2🍕 k \log_2 k
\end{aligned}$$

Expand inductive hypothesis.

$d \le k/2$, so $\log_2 d \le \log_2 k/2$

$k-d \le k$, so $\log_2 (k-d) \le \log_2 k$

Opening the ~~🍕~~ Box

- How do we quickly find if an element is a possib root?
    - If we can make 🍕 fast, we are done!
- Each time we recurse on some subarray [l,r]
    - “Rootness” of element may change!
- Depends on what other items in subarray
    - Which are not coprime
        - i.e. GCD(notRoot, other) != 1

Can 15 be a root?

2	7	**15**	8	9	5	❌
2	7	**15**				✓
	7	**15**	8	9	5	❌
		15	8			✓

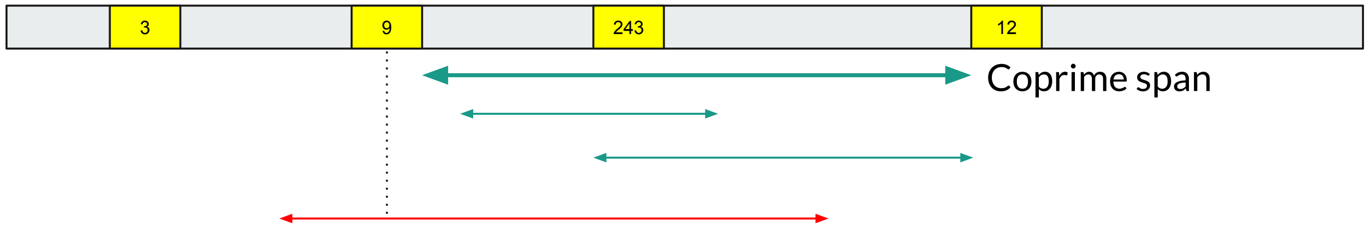
Coprimality

- How to test if two numbers are coprime?
  - GCD, O(log M) for numbers at most M.
- Given N numbers, how to bulk-test pairwise coprimality?
  - GCD(a,b) != 1 iff some prime p is a common divisor
  - Maybe something to do with prime factorization?
- Let's examine special cases:
  - Only 1 prime divisor

Prime Powers

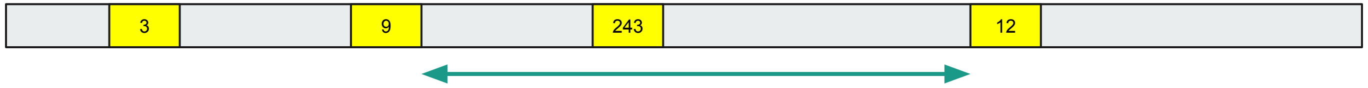
- Take for example, **$243 = 3^4$**
- What can “block”/prevent 243 from becoming root?
  - GCD(243, X) != 1
  - X must be multiple of 3!
- **IDEA: Track all the multiples of 3 in the whole array.**
  - 243 is only affected by multiples of 3.

Coprime Spans



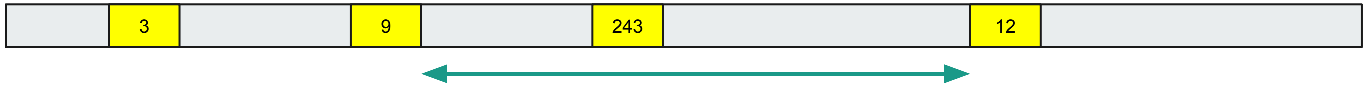
- As long as subarray doesn't cover any other multiple of 3
  - 243 can be a root
- Call the **largest** subarray around 243, without other multiples of 3:
  - <u>**Coprime span**</u> of 243
- 243 can be root in [L,R] ⇔ [L,R] fits within **coprime span**
  - If we **precompute all coprime spans,** 🍕 is O(1)!

Coprime Span Computation



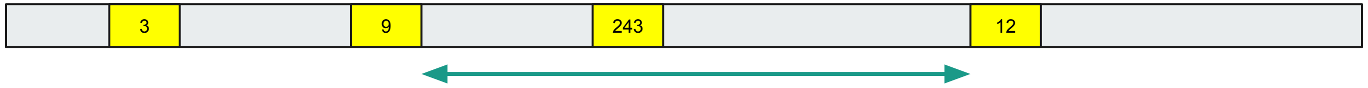
- Method 1: Start at 243.
  - Try expanding left/right, one position at a time
    - Until we hit a multiple of 3.
  - May cover significant part of whole (sub)array
    - Slow! (O(N))
- We don’t really care about “empty space”. Can we “skip” ahead?

Coprime Span Computation



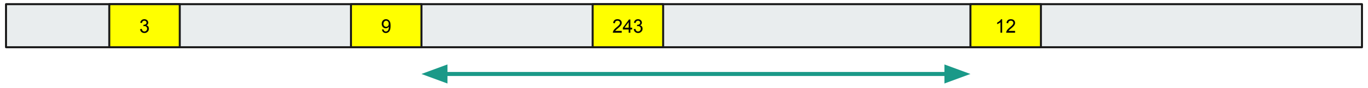
- We don’t really care about “empty space”. Can we “skip” ahead?
  - Skip to rightmost multiple of 3 **right before 243**
    - Largest index, **less than 243’s index**
  - Skip to leftmost multiple of 3 **right after 243**
    - Smallest index, **more than 243’s index**
- Looks like TreeSet.lower/higher()!

Coprime Span Computation: Binary Search



- Method 2: Precompute positions of multiples of 3.
  - When reading input, if multiple of 3, add position to TreeSet.
    - At most N positions tracked, O(N log N) time to add.
  - To compute 243's coprime span:
    - Check TreeSet.lower/higher(position of 243)
    - If no lower, span extends to start (index 0)
    - If no higher, span extends to end (index N-1)
  - O(log N) to compute 243's span!

Coprime Span Computation: Binary Search

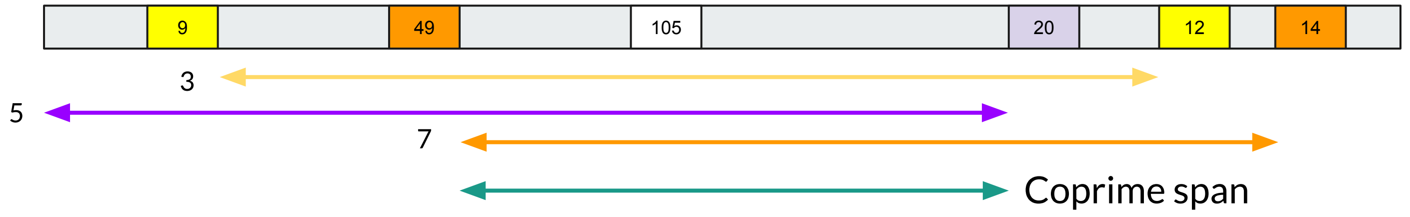


- We don't really use the "dynamic" nature of TreeSet
  - Only build whole thing, then query
- Method 2a: Precompute positions of multiples of 3.
  - When reading input, if multiple of 3, add position to ArrayList.
    - O(N) time to add.
  - To compute 243's coprime span:
    - Collections.binarySearch(list, position of 243)
  - O(log N) to compute 243's span!

Multiple Prime Factors

- 243 is really special case (3^k, prime power)
  - Many more primes than just 3
- Most numbers have multiple distinct prime factors.
- E.g. $105 = 3 \times 5 \times 7$
  - Each number $K \leq$ (max value M)
    - Has at most ($\log_2 K \leq \log_2 M$) prime factors
    - Dividing off a prime $\leq$ Dividing by 2
  - Have to worry about 3-multiples, 5-multiples, 7-multiples
  - Coprime span cannot cover any of them!

Multiple Prime Factors



- Solution:
  - Find per-prime spans, and **take intersection**
  - Left = Max(Left$_3$, Left$_5$, Left$_7$)
  - Right = Min(Right$_3$, Right$_5$, Right$_7$)
- Total O(N log N log M)
  - N numbers × O(log N) query × O(log M) primes

Lots of Primes

- For each prime, compute list of multiples’ positions.
- How many primes do we need to worry about?
  - Maximum value is M
  - Prime Number Theorem: About M/ln M primes under M.

Precomputation

- Keep separate ArrayList for each prime ≤ M
  - HashMap<Integer, List<Integer>> primePositions;
- For each input value X [N inputs]
  - Factorize X. **[BLACK BOX, O(~~🌭~~) time]**
  - For each distinct prime P in factorization [≤ $\log_2$ M primes]
    - primesPositions.get(P).add(current position) O(1) get & add
- Overall O(N (🌭 + log M))

Primes ~~🌭~~

- How to prime factorize single number?
  - Trial division: O(sqrt(M)) per factor × O(log M) factors
  - Total precomputation:
    - O(N(🌭+log M)) = O(N sqrt(M) log M), may be too slow!
- How to prime factorize many numbers (value at most M)?
  - IDEA: For any number K ≥ 1
    - Shrink it by dividing by **smallest prime factor**
    - O(1) per factor × O(log M) factors
    - Total precomputation: O(N log M), good!
- Now: **Precompute** smallest prime factors for a lot of numbers?

Sieve of Eratosthenes

- Array of booleans IsPrime[2...M], initially all true
  - For each number X in [2...M] ascending:
    - If IsPrime[X] is still true, X is prime.
    - Strike off all multiples of X.
- **IDEA:** Track reason why a composite number was struck off
  - The first strike-off of X, is by its **smallest** prime factor

Sieve of Eratosthenes

```
isComposite = new boolean [M+1] // All false

For i in [2...M]
      If not isComposite[i]
            // i is prime
            For j in [i...M/i]
                  isComposite[i*j] = True
```

```
smallestPF = new int[M+1] // All 0

For i in [2...M]
      If smallestPF[i] == 0
            // i is prime
            smallestPF[i] = i
            For j in [i...M/i]
                  If smallestPF[i*j] == 0
                        smallestPF[i*j] = i
```

Sieving up to M takes O(M log log M) time.
This allows O(log M) factorization per number, in bulk.

Putting It All Together

1. Create map of primes to position lists.	O(1)	
2. Run sieve up to maximum M.	O(M log log M)	
3. Read in input. For each input X	N iterations:	Overall O(N log M)
a. Store X in array	O(1)	
b. For each prime in factorization	O(log M) iterations	
i. Add position to prime's list	O(1)	
4. For each X in array	N iterations:	Overall O(N log N log M)
a. Left = 0, Right = N-1	O(1)	
b. For each prime in factorization	O(log M) iterations	
i. Query left/right for this prime.	O(log N) queries	
c. Save coprime span.	O(1)	
5. Treeify(0, N-1, -1)	O(N log N)	

Total time complexity:
O(M log log M + N log N log M)

Speedups

```
1. Create map of primes to position lists.
2. Run sieve up to maximum M.
3. Read in input. For each input X
    a. Store X in array
    b. For each prime in factorization
        i. Add position to prime's list
4. For each X in array
    a. Left = 0, Right = N-1
    b. For each prime in factorization
        i. Query left/right for this prime.
    c. Save coprime span.
5. Treeify(0, N-1, -1)
```

Faster sieving methods.

We compute factorization twice per number.

- The first time, we add it to a list.
- The second time, we want the item right before/after it in the list
  - Via binary search

Combine/Fuse the two loops

- Replace binary search with array lookup
  - Adjacent element at current time
- O(log N) => O(1)

Total time complexity:

O(M ~~log log M~~ + N ~~log N~~ log M)

Minor Speedups

- Faster than Fast IO
  - Custom per-digit reading of input integers
  - Manual input/output buffer management
- Replace HashMap with arrays where feasible
  - Our keys are often integers
    - Small (at most a few million)
    - Dense (Prime density is still $1/\ln M \approx 1/16$)
  - Even best-case hashmap access is slower than array access
- Avoid producing temporary arrays to store factorization
  - Compute in loop directly, from smallestPF table

Take-Home Lab 3 - Baby Names [Optional]

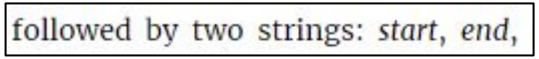
- Update and query a database of baby names, with gender suitability
- Q <= <u>**500,000**</u> queries

0. Exit program.
1. Given name and gender suitability, add suggestion
2. Given name and gender suitability, remove suggestion
3. Given [start] and [end], and suitability
    - Print number of names satisfying { [start] <= name < [end] }

FOCUS ON PREVIOUS PROBLEMS/OTHER MODS FIRST
This problem is much harder, but offers decent practice in implementing DSes.

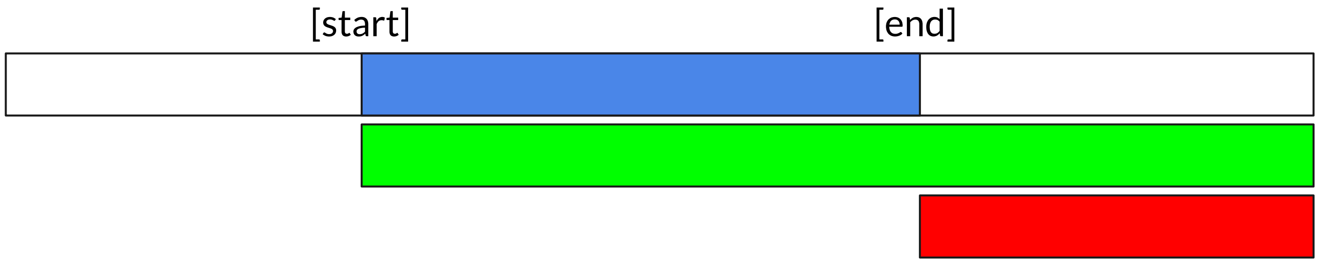
Query

- Kattis sample input gives you 1-letter queries ‘A’ and ‘Z’
    - **<u>Not true in general!</u>**
        - Only guarantee is at most 30 letters.
    - We can ask: In {“Andy”, “Brian”, “Charlie”}
        - How many names are (“Ada” <= name < “Anna”)?
        - Answer: 1 (“Andy”)


followed by two strings: *start*, *end*,

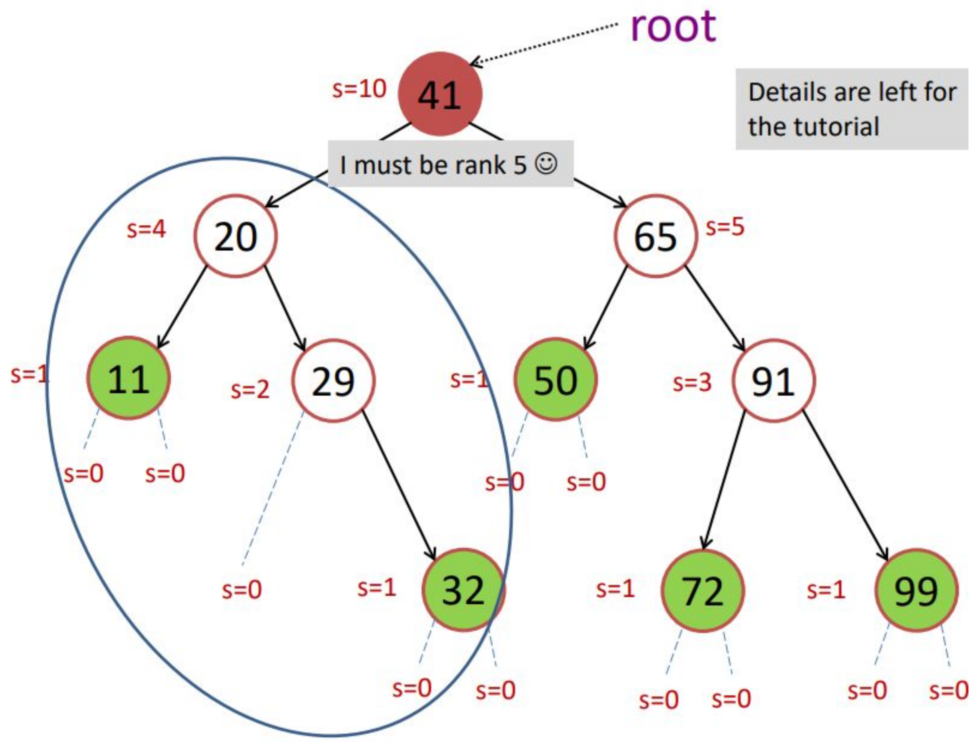
Counting

- We want to answer queries of the form
    - Size of { [start] <= name < [end] }
- We can rewrite these as:
    - Size of { [start] <= name } - Size of { [end] <= name }



Binary Search Trees: Size (s)

Details are left for the tutorial



Since this image grew much bigger than last time, this is probably an important hint.

Rank

- Rank of value X (borrowing from BSTs)
  - Size of { names <= [X] }
- What we want
  - Size of { [start] <= name }
- Just count in reverse order

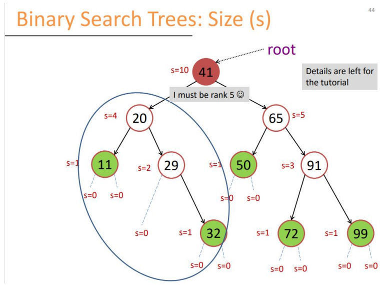
Counting

- If [start] is "EFG", already went into "E" node
  - How do we **count** things like "EGA", but not "EEA"?
  - Count **whole of everything** after EF:
    - EG(...), EH(...), …

      Don't recurse, just add summary values.
  - Part of EF(...) should count
    - How to count part?

      Recurse with remainder of string.
  - Do not count anything before EF:
    - EE(...), ED(...), ...

      Don't recurse, just ignore entirely.



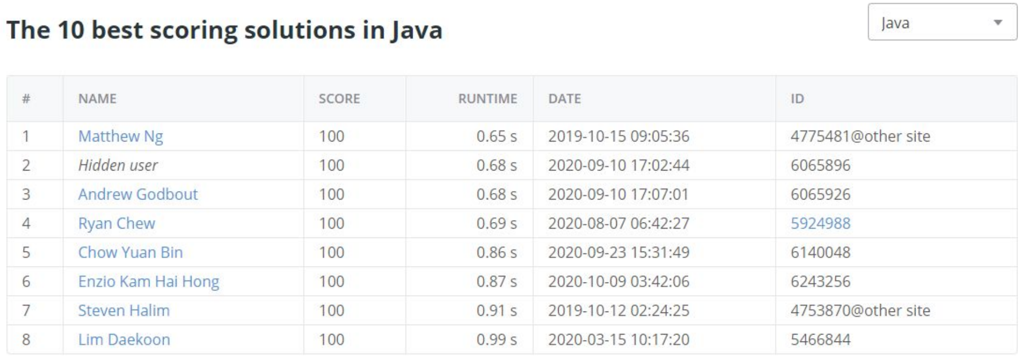
T(L) = O(26) + T(L-1)

Performance Caveats!

Fenwick Tree
RSQ solutions

Trie Model solution

AVL



The 10 best scoring solutions in Java

Java

#	NAME	SCORE	RUNTIME	DATE	ID
1	Matthew Ng	100	0.65 s	2019-10-15 09:05:36	4775481@other site
2	*Hidden user*	100	0.68 s	2020-09-10 17:02:44	6065896
3	Andrew Godbout	100	0.68 s	2020-09-10 17:07:01	6065926
4	Ryan Chew	100	0.69 s	2020-08-07 06:42:27	5924988
5	Chow Yuan Bin	100	0.86 s	2020-09-23 15:31:49	6140048
6	Enzio Kam Hai Hong	100	0.87 s	2020-10-09 03:42:06	6243256
7	Steven Halim	100	0.91 s	2019-10-12 02:24:25	4753870@other site
8	Lim Daekoon	100	0.99 s	2020-03-15 10:17:20	5466844

- Most likely to just scrape past 0.99s
- Question is highly specific to particular trie implementation
- 5×10^5 queries, storing 2×10^5 names
    - A lot of things to read in/write out **(hint hint)**

Trie

- Java API does not contain a trie class
- Some suggestions:
    - Avoid implementing trie “optimizations” for longer strings
        - E.g. path compression, qp-trie, etc.
        - Names are at most 30 characters
    - Take advantage of small radix
        - Names consist of only uppercase characters A-Z