香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

M2641 - Subtree Dictionary

Ethen Yuen {ethening}

2026-06-28

Table of Contents

Background

Problem Idea by `ethening`
Preparation by `ethening`

The Problem

- We are given a rooted tree with N nodes, rooted at node 1.
- Each node i has a string S_i.
- Each query gives a node X and a string T.
- We need to count nodes u such that:
  - u is in the subtree of X
  - S_u appears as a substring of T
- If two nodes have the same string, they are counted separately.

Two Dimensions

This task combines two independent-looking difficulties.

Tree dimension

- Query only considers subtree of X.
- We need to maintain or count nodes in a subtree.

String dimension

- Need to know which S_i appear in T.
- There are many pattern strings.

The full solution will handle both dimensions at the same time.

Constraints

For all test cases:

- $2 \leq N \leq 10^5$.
- $1 \leq Q \leq 10^5$.
- $1 \leq P_i < i$ for all $2 \leq i \leq N$.
- $\sum_{i=1}^{N} |S_i| \leq 5 \times 10^5$.
- $\sum_{j=1}^{Q} |T_j| \leq 5 \times 10^5$.
- All strings are non-empty and contain only lowercase English letters.

Cases

Case	N, Q	$\sum \lvert S_i \rvert, \sum \lvert T_j \rvert$	Additional Properties
1–2	≤ 200	≤ 2000	None
3–4	$\leq 10^5$	$\leq 10^5$	A
5–6	$\leq 10^5$	≤ 10000	None
7–9	$N \leq 10^5,\ Q \leq 5000$	$\leq 5 \times 10^5$	None
10–11	$\leq 10^5$	$\leq 5 \times 10^5$	B
12–16	$\leq 10^5$	$\leq 5 \times 10^5$	C
17–20	$\leq 10^5$	$\leq 5 \times 10^5$	None

A: $|S_i| = 1$ for all i. B: $P_i = i - 1$ for all $i \geq 2$.

C: Each S_i appears in each T_j at most once.

Table of Contents

[Cases 1-2] Small Constraints

Here $N, Q \leq 200$ and $\sum |S_i|, \sum |T_j| \leq 2000$.

- Precompute the list of nodes in every subtree, or run DFS per query.
- For query (X, T):
  - enumerate all nodes u in subtree of X
  - check whether S_u appears in T using `find`
- Duplicate strings are naturally handled, because we count nodes.

Time Complexity: $O\left(NQ + \left(\sum |S_i|\right)\left(\sum |T_j|\right)\right)$

Expected Score: 10

Euler Tour

The standard trick for rooted tree subtree queries, to flatten it into a sequence:

- DFS the original tree.
- Let tin_u be the time we enter node u.
- Let tout_u be the last time inside subtree u.

Then:

$$u \text{ is in the subtree of } X \iff \text{tin}_X \leq \text{tin}_u \leq \text{tout}_X.$$

[Cases 3-4] Single-Character S_i

If every S_i is a single character, the string part becomes easy.

- For each character c, store all tin_u such that $S_u = c$.
- For query (X, T), mark which characters appear in T.
- For every marked character, binary search how many stored positions lie in:

$$[\text{tin}_X, \text{tout}_X].$$

Time Complexity: $O\left(N + \sum |T_j| + 26Q \log N\right)$

Expected Score: 20

[Cases 5-6] Small String Length

Suppose $\sum |S_i|$ and $\sum |T_j|$ are small.

- For each distinct string s, store sorted tin positions of nodes with $S_i = s$.
- For each query string T, enumerate all distinct substrings of T using rolling hash.
- If a substring equals some S_i, count its nodes inside the subtree interval by binary search.

Important detail:

- If multiple nodes have the same S_i, all of them are stored and counted.

Time Complexity: $O\left(N \log N + \sum |S_i| + \sum_j |T_j|^2 \log N\right)$

Expected Score: 30

Table of Contents

Many Pattern Strings

We need to match many strings $S_1, S_2, \ldots, S_N$ against many query strings.

Natural data structure:

Aho-Corasick automaton (AC automaton)

Build the automaton using all S_i. Let pos_i be the terminal AC state of S_i.

[Cases 7-9] Few Queries

With small Q, we can afford a less optimized matching step.

- Scan T in the AC automaton.
- At every visited state, explicitly climb fail links.
- Every terminal state found corresponds to a matched pattern string.
- Use timestamps to avoid counting the same terminal state twice in the same query.
- Count nodes in subtree X using sorted Euler positions.

Time Complexity: $O\left(\sum |S_i| + \sum_j |T_j| \cdot H \log N\right)$, where H is the maximum fail-link depth.

Expected Score: 45

Why Explicit Fail Climbing Fails

The previous method may walk many fail links at every character.

Consider patterns arranged like:

$$a,\ aa,\ aaa,\ \ldots,\ a^m$$

and a long query string:

$$T = aaaaa\ldots a.$$

Each visited AC state has a long fail chain, so the work can become quadratic.

The full solution avoids walking these chains explicitly.

Fail Tree

Consider the fail links of the AC automaton.

- Every state except the root has one fail parent.
- Reverse these fail links to form the **fail tree**.

Key observation:

$$S_i \text{ appears in } T$$

if and only if

$$\text{pos}_i \text{ is an ancestor of some state visited while scanning } T$$

in the fail tree.

Why Ancestors?

When scanning T, suppose we are at AC state v.

- The state v represents a suffix of the prefix of T scanned so far.
- Following fail links gives shorter suffixes.
- Therefore, terminal states on the fail chain of v are exactly pattern strings ending here.

In the fail tree, the fail chain is the path from v to the root. So matched terminal states are ancestors of visited states.

Table of Contents

BIT Trick on the Fail Tree

Suppose some terminal state p is currently active, meaning that it should be counted in a subtree query.

We want a query at state v to count p if:

$$p \text{ is an ancestor of } v$$

in the fail tree.

We can consider the Euler tour of the fail tree:

$$p \text{ is ancestor of } v \iff \text{tin}_p^F \leq \text{tin}_v^F \leq \text{tout}_p^F.$$

BIT Trick on the Fail Tree

Remember, we want to count the p that matters at a AC state v. To do that, we can activate one terminal state p by:

$$\text{add } +1 \text{ to interval } [\text{tin}_p^F, \text{tout}_p^F].$$

Then for any AC state v:

$$f(v) = \text{point query at tin}_v^F$$

gives the number of active terminal states that are ancestors of v.

This can be implemented by BIT range-add and point-query.

Overcounting

If we scan T and simply sum $f(v)$ over all visited states v, we count occurrences.

But the task asks:

Does S_i appear at least once?

Example:

$$S_i = a, \quad T = aaaa.$$

The string a appears many times, but node i should be counted once.

Union of Root Paths

Let the distinct visited AC states be:

$$v_1, v_2, \ldots, v_k.$$

A terminal state is matched if it lies in the union of paths:

$$\text{root} \to v_1, \quad \text{root} \to v_2, \quad \ldots, \quad \text{root} \to v_k.$$

Therefore we need to count active terminal states in this union of root paths.

LCA Deduplication Formula

Sort $v_1, v_2, \ldots, v_k$ by DFS order in the fail tree.

Let $f(v)$ be the number of active terminal ancestors of v.

Then:

$$\text{answer} = \sum_{i=1}^{k} f(v_i) - \sum_{i=2}^{k} f(\text{lca}(v_{i-1}, v_i)).$$

This counts every active terminal state in the union of root paths exactly once.

Why LCA Deduplication Works

Fix one active terminal state p.

It contributes to $f(v_i)$ exactly when p is an ancestor of v_i.

After sorting by fail-tree DFS order:

- all visited states inside subtree p form one contiguous block
- if this block has size c, then p is counted c times in $\sum f(v_i)$
- among adjacent pairs, exactly $c - 1$ LCAs are still inside subtree p.

Therefore p contributes $c - (c - 1) = 1$ if it appears, and 0 otherwise.

Virtual Tree View

A **virtual tree** is the compressed tree containing the selected nodes $v_1, v_2, \ldots, v_k$ and the LCAs needed to connect them.

If selected nodes are sorted by DFS order, the virtual tree edges are determined by adjacent LCAs.

The LCA formula is the same idea without explicitly building the virtual tree:

$$f(v_1) + \sum_{i=2}^{k} \left(f(v_i) - f(\mathsf{lca}(v_{i-1}, v_i))\right).$$

Each term adds only the new part of the root path contributed by v_i.

[Cases 10-11] Chain

If $P_i = i - 1$, the original tree is a chain.

The subtree of node X is just the suffix: $X, X + 1, \ldots, N$.

We can process queries by decreasing X.

- Activate $S_N, S_{N-1}, \ldots$ one by one.
- When reaching X, active strings are exactly the subtree of X.
- Answer using the fail-tree BIT and LCA formula.

Time Complexity: $O\left(\sum |S_i| + (N + \sum |T_j|) \log \sum |S_i|\right)$

Expected Score: 55

[Cases 10-11] Chain

Cases 10–11 remove the original-tree maintenance.

But it still requires:

- AC automaton
- Fail-tree ancestor view
- BIT range-add / point-query
- LCA deduplication for repeated matches

Table of Contents

[Cases 12-16] No Repeated Appearance

Suppose every S_i appears in every T_j at most once.

Then repeated-match deduplication is unnecessary.

We still need to handle the original-tree subtree restriction.

Instead of maintaining a subtree directly, we convert it into two prefix queries on the Euler order.

Prefix Transformation

Define $F(k, T)$:

$$F(k, T) = \#\{u : \text{tin}_u \leq k,\ S_u \text{ appears in } T\}.$$

Since a subtree is an interval in Euler order:

$$\text{answer}(X, T) = F(\text{tout}_X, T) - F(\text{tin}_X - 1, T).$$

So each query becomes two offline events.

Sweeping Euler Prefixes

Sweep $k = 0, 1, \ldots, N$ in original-tree DFS order.

- When node u enters the prefix, activate S_u.
- Activating S_u means activating terminal state pos_u.
- On the fail tree, this is a range add:

$$[\text{tin}^F_{\text{pos}_u}, \text{tout}^F_{\text{pos}_u}].$$

- At prefix k, the BIT represents exactly nodes with $\text{tin}_u \le k$.

Query Procedure for Cases 12-16

To evaluate one prefix event for string T:

1. Scan T in the AC automaton.
2. For every visited state v, query $f(v)$ from the fail-tree BIT.
3. Add these values:

$$\text{answer} = \sum_{\text{visited } v} f(v).$$

Since each S_i appears in T at most once, this does not overcount.

Time Complexity: $O\left(\sum |S_i| + (N + \sum |T_j|) \log \sum |S_i|\right)$

Expected Score: 80

What Is Missing

The remaining issue is that the same S_i may appear many times in one query string T.

Therefore the full task needs a way to count the union of fail-tree root paths, not the sum over all occurrences.

This is exactly what the LCA deduplication formula computes.

Full Query Procedure

To evaluate one prefix event for string T in the full solution:

1. Scan T in the AC automaton.
2. Collect each distinct visited state once.
3. Sort them by fail-tree DFS order.
4. Apply:

$$\sum f(v_i) - \sum_{i=2}^{k} f(\mathsf{lca}(v_{i-1}, v_i)).$$

The final query answer is still:

$$F(\mathsf{tout}_X, T) - F(\mathsf{tin}_X - 1, T).$$

[Cases 17-20] Full Solution

Combine all components:

- AC automaton over all S_i.
- Fail tree, DFS order, and LCA preprocessing.
- Euler tour on the original tree.
- Convert each subtree query into two prefix events.
- Sweep Euler prefixes and activate terminal states in the fail-tree BIT.
- For each event, sort distinct visited AC states and apply the LCA formula.

Time Complexity: $O\left(\sum |S_i| + (N + \sum |T_j|) \log \sum |S_i|\right)$

Expected Score: 100

Complexity

Let:

$$L_S = \sum_{i=1}^{N} |S_i|, \quad L_T = \sum_{j=1}^{Q} |T_j|.$$

- AC automaton: $O(L_S)$ states and transitions.
- Original-tree Euler tour and fail-tree preprocessing: $O(N + L_S)$.
- Prefix sweep activates each node once.
- Each activation updates the fail-tree BIT in $O(\log L_S)$.
- Query-event processing costs $O(L_T \log L_S)$ in total.

Overall:

$$O\left(L_S + (N + L_T) \log L_S\right).$$

Alternative: DSU on Tree

There is another valid way to maintain the active set.

- Run DSU on tree on the original rooted tree.
- When processing node X, keep exactly the strings in subtree X active.
- Use the same fail-tree BIT and the same LCA deduplication formula for queries attached to X.

This ensures that when you process a query, the active set is exactly about all the nodes in the subtree that you care about. This implementation requires one more log factor and it's a bit more complicated to code.

Takeaways

This task tests familiarity with several standard tools, rather than requiring one very creative observation. Please make sure you are skilled with the following:

- Euler tour converts subtrees to intervals.
- Aho-Corasick automaton handles multi-pattern matching.
- Fail tree turns substring matching into ancestor queries.
- Offline prefix sweep handles subtree restrictions.
- LCA on the fail tree deduplicates repeated matches.

It also rewards modular code, because several data structures have to work together cleanly.

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

M2642 Rollercoaster Tycoon

Daniel Hsieh {QwertyPi}

2026-06-28

Background

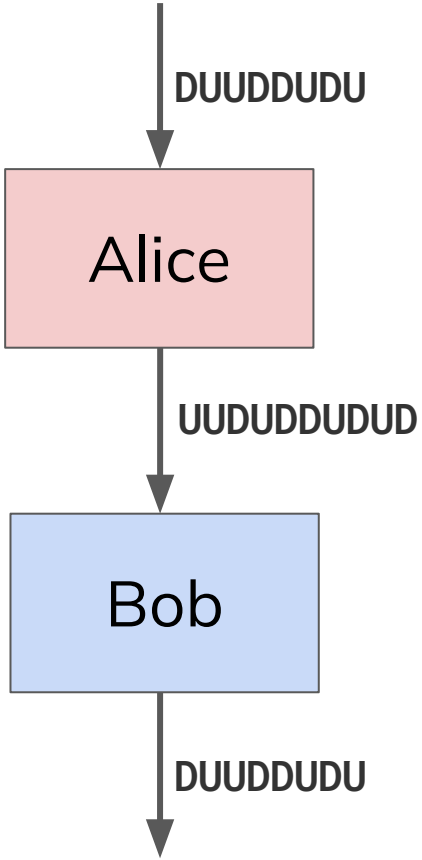
- Problem Idea by `firewater`
- Preparation by `QwertyPi`

The Problem

- Given length-N string to Alice
  - **Equal numbers of U and D**

- Alice send a length-M string to Bob
  - **Equal numbers of U and D**
  - **For all prefixes, number of U ≥ number of D**

- Bob: Decrypt the original string
- Communication Task!



The Cases

#	N
01	18
02	
03	36
04	
05	60
06	90
07	120
08	180
09	240
10	300

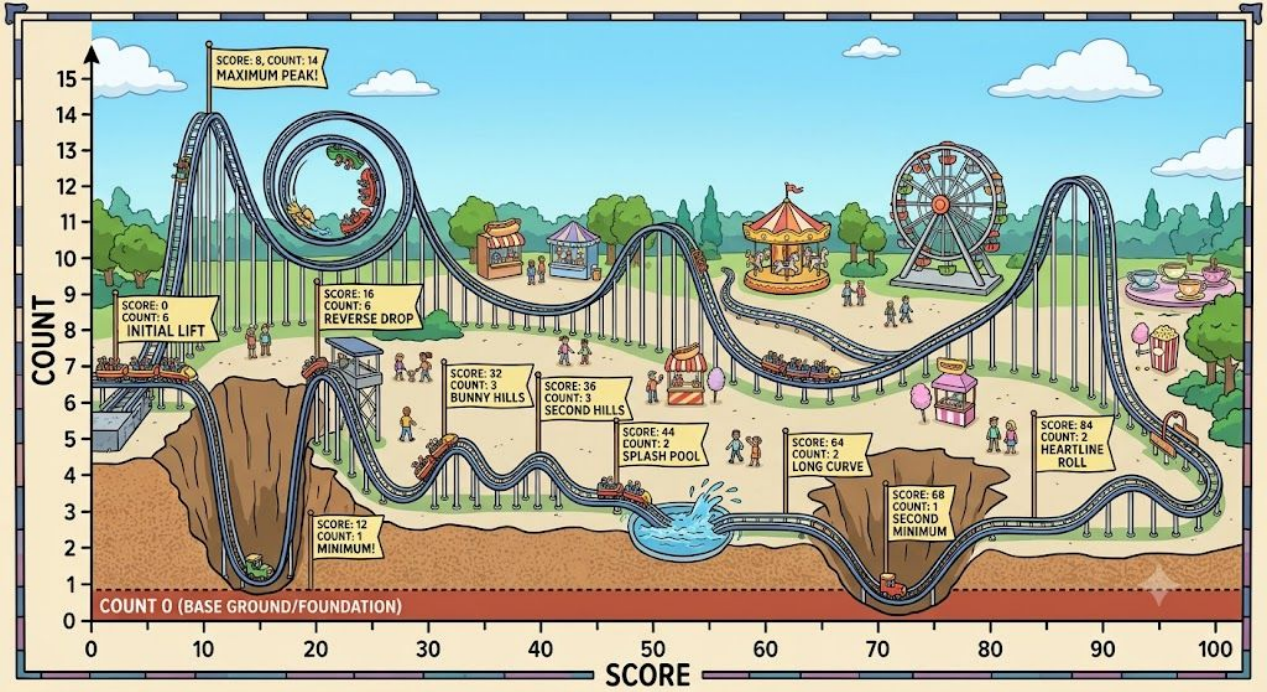
#	N
11	450
12	750
13	1500
14	3000
15	6000
16	12000
17	18000
18	24000
19	30000
20	45000

#	N
21	60000
22	90000
23	120000
24	240000
25	360000

For all cases:

$M = N + 20$

Statistics



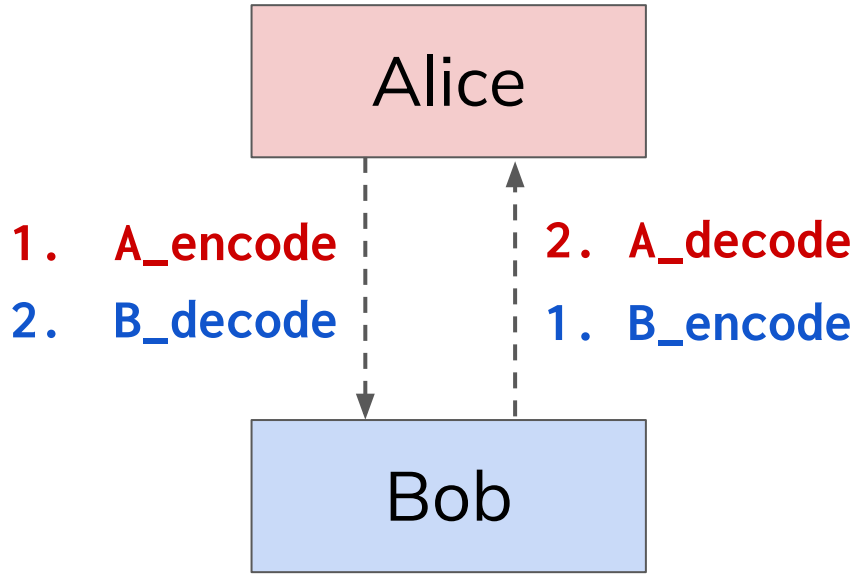
First solved by nobody!

Dream

This task is so easy: Just create a mapping between strings and integers!

```
int A_encode(int N, string A);
string A_decode(int X, int M);

int B_encode(int M, string B);
string B_decode(int Y, int N);
```



Score: 100

~~Dream~~ Reality

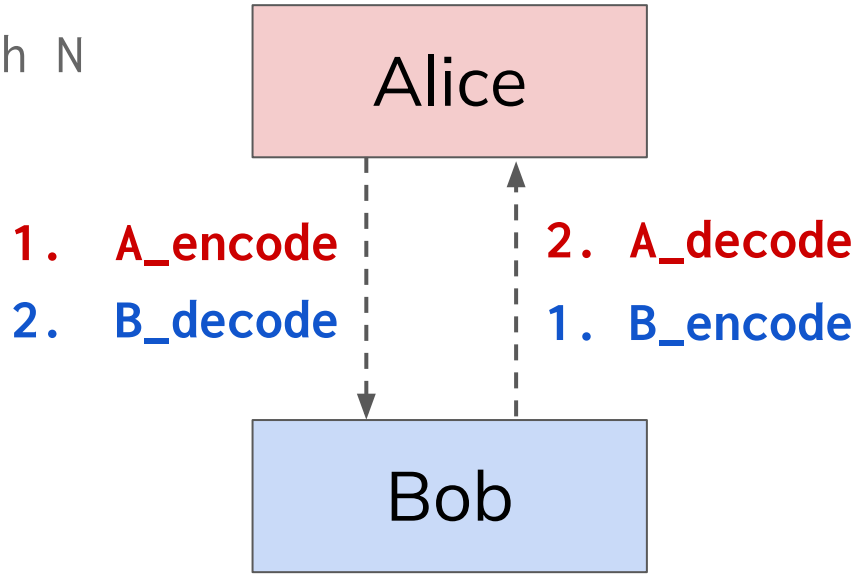
Conclusion: Do not sleep during mini comp!

This task is so easy: Just create a mapping between strings and integers!

```
dp[N][H] := # of paths of length N
  with current height H
```

Simply compute lexicographical order with DP, ~~bigint and FFT~~

~~Score~~ Runtime: 100 **hours**
Actual Score: Max 48



Simple Solutions

Solution 1. Replace each U by UD and D by DU, then add U at front and D at end. Requirement: $M \geq 2N + 2$.

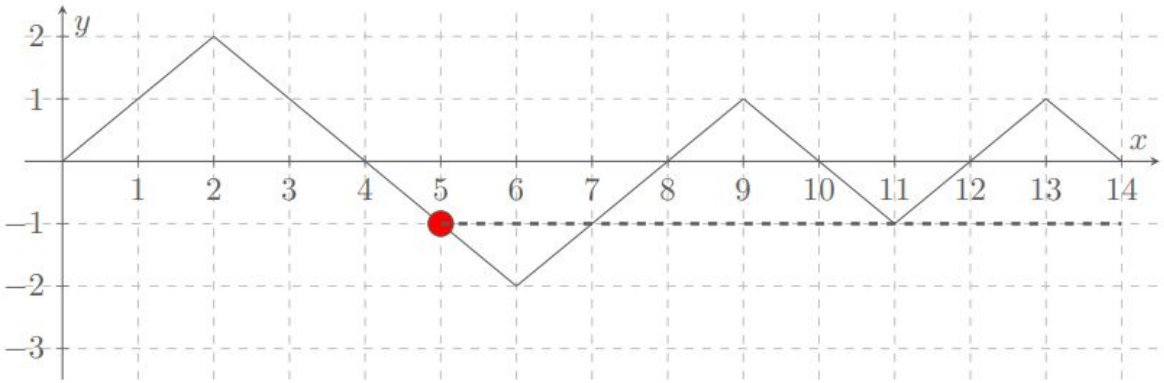
Expected Score: 8

Solution 2. If we are “lucky”, then maybe the polyline does not do too “deep below” the horizontal. Random shuffle (with fix seed), add a tons of U at front and a tons of D at the back. Requirement: Luck.

Expected Score: 16 ~ 24

Path Counting

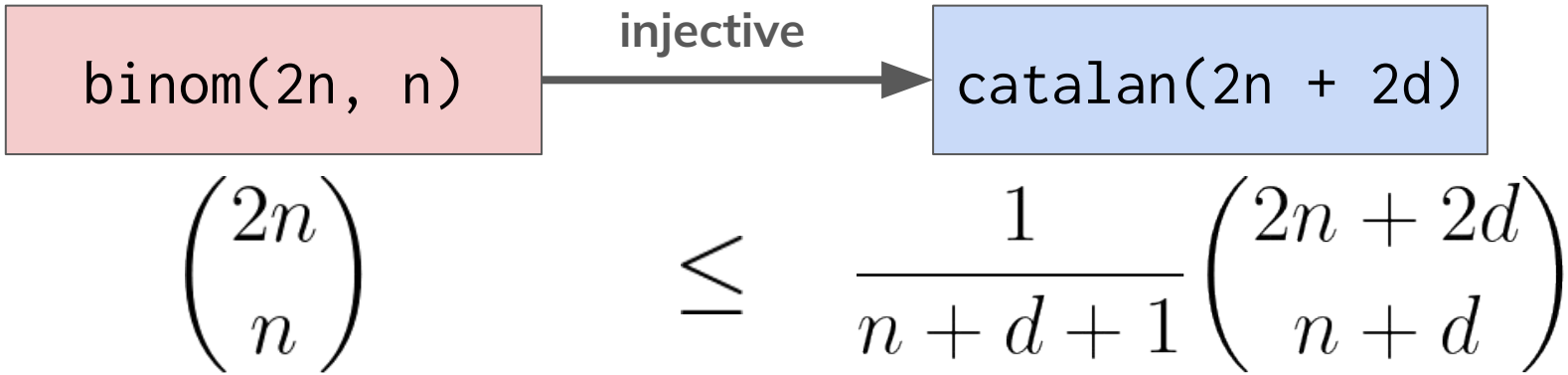
- Let S_N be the set of all length-N (even) paths where you starts at (0, 0), each time go (+1, +1) or (+1, -1), and ends at (N, 0).
- Let T_N be a subset of S_N, where additionally **Y ≥ 0 at all times**.
- Target: Find an **injective** function that maps $\mathbf{S_N}$ to $\mathbf{T_{N+D}}$:
  - That is, no two paths map to the same path



D = 20
for this task

From Binomial to Catalan

Let `N = 2n, D = 20 = 2d`:



$$\binom{2n}{n} \leq \frac{1}{n+d+1}\binom{2n+2d}{n+d}$$

Explicit Formula for `binom(n, r)`

Math in OI (II) 2026

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} \tag{1}$$

We show that $n! = \binom{n}{r} r!(n-r)!$.

What are we counting?

The number of permutations of $1 \dots n$.

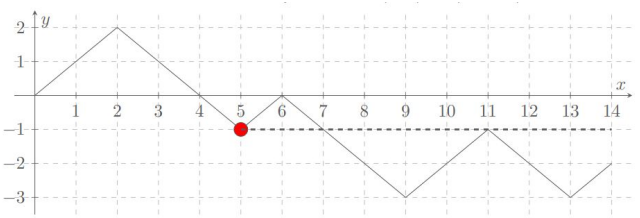
- LHS: By definition, it is $n!$.
- RHS: Choose r elements out of n elements to be the first r elements.
  - For the first r elements, there are $r!$ ways to permute them.
  - For the next $n-r$ elements, there are $(n-r)!$ ways to permute them.

  By multiplication principle, there are $\binom{n}{r} r!(n-r)!$ permutations.

Explicit Formula For `catalan(n)`

Math in OI (II) 2026

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$



Reformulate the Catalan model as follows:

- You start from $(0, 0)$ and want to reach $(2n, 0)$.
- From (x, y) you can move "up" to $(x + 1, y + 1)$ or "down" to $(x + 1, y - 1)$.
- C_n is the number of "good" paths – paths that do not go below the x-axis.

Ideas:

- We count, instead, the number of bad paths.
- Then $C_n = \binom{2n}{n}$ – (number of bad paths).
- Therefore, for the formula to hold, we expect there to be $\binom{2n}{n+1}$ bad paths.

Asymptotic Growth Rate

TL; DR: If the length of your rollercoaster extends by 2, then number of possibilities (approximately) quadruples.

Binomial

$$\frac{\binom{2n+2}{n+1}}{\binom{2n}{n}} = \frac{(2n+1)(2n+2)}{(n+1)^2} = \frac{4n+2}{n+1} \approx 4$$

Catalan

$$\frac{C_{2n+2}}{C_{2n}} = \frac{(2n+1)(2n+2)(n+1)}{(n+1)^2(n+2)} = \frac{4n+2}{n+2} \approx 4$$

Theoretical Lower Bound

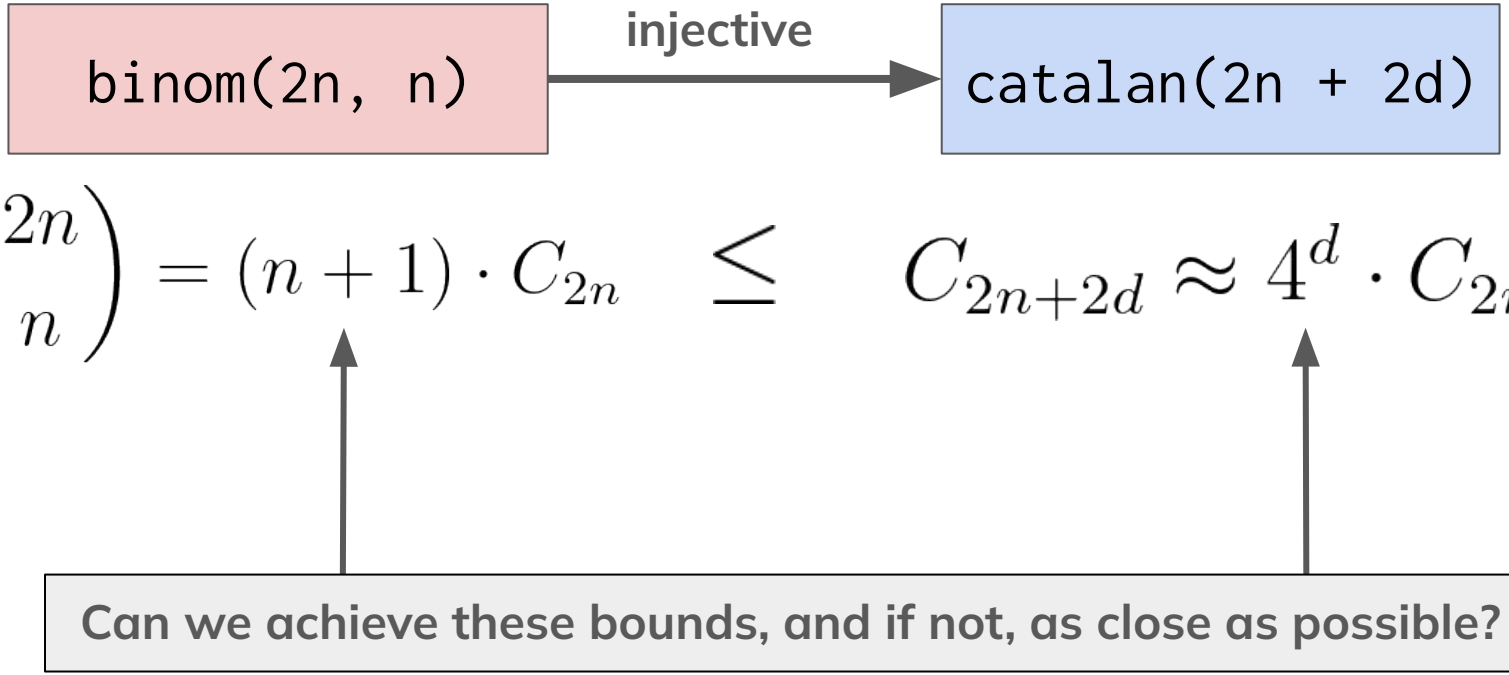
$$\binom{2n}{n} = (n+1) \cdot C_{2n} \quad \leq \quad C_{2n+2d} \approx 4^d \cdot C_{2n}$$

Here comes the theoretical best:

$$d \approx \log_4 n$$

For d = 10, this corresponds to n ≈ 10^6.
Fact: the preparer only come up with n ≈ 1.8 × 10^5 solution :(

Theoretical Lower Bound



Catalan Encoding

Target

A

```
pair(int, good_path_n) A_encode(any_path_n A);
any_path_n A_decode(int X, good_path_n P);
```

n+1

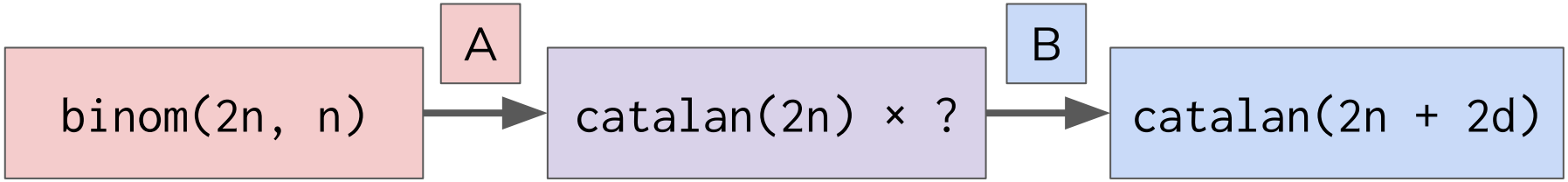
B

```
pair(int, good_path_n) B_encode(good_path_m B);
good_path_n B_decode(int X, good_path_n P);
```

4^d

From now on, we separate the task into A and B two parts.

This This Solution



Solutions

A1. 2n

A2. n+1

B1. 2^{d-1}

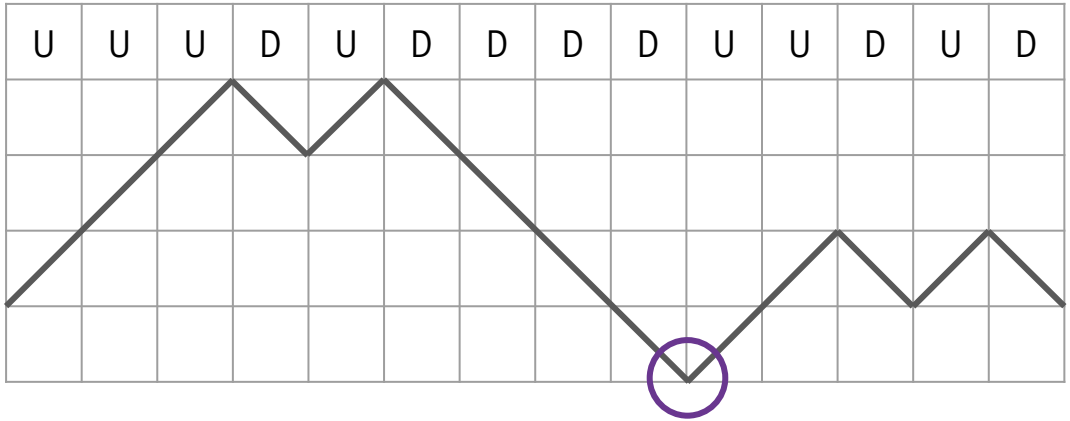
B2. catalan(2d)

B3. binomial(2d, d)

Side A: Encode Binomial by (Catalan, Number)

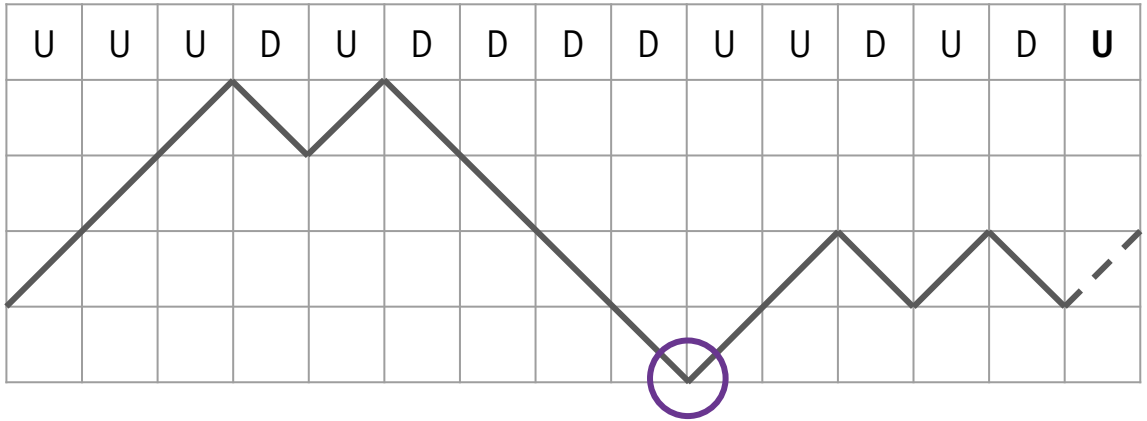
- Given a path of length 2n from (0, 0) to (2n, 0).
- Send an integer and a good path of length 2n.

A1. (2n) Simply observe that there exists a **cyclic shift** of the original path that is good. Why? Simply pick the lowest point to start the cyclic shift!



Side A: Encode Binomial by (Catalan, Number)

A2. (n + 1) Append a U at the end. Claim: there exists an **unique** cyclic shift that starts with U, then a good path (that never goes below y = 1), then delete the first U. Send the label of the U you have deleted.



Side B: Encode (Catalan, Number) by longer Catalan

- Given an integer and a good path of length 2n.
- Send a good path of length 2n + 2d.

B1. (2^{d-1}) Simply use the 2N + 2 idea (U -> UD, D -> DU) to encode the integer and append to the end of the good path.

B2. (catalan(2n)) Actually note that we are just encoding a good path of length 2d which is very small - so we can simply dp (or even brute force) to encode good path of length 2d in integer optimally!

Side B: Encode (Catalan, Number) by longer Catalan

`B3. (binomial(2n, n))` Note that we can insert the original good path of length 2n into good path of length 2d anywhere, if we can “recognise” its position.

In fact, for n >> d, by putting the good path of length 2d **at the beginning**, or **after an U**, its position always be recognised!

Note that the insertion position is always the first position where the next position with the same height is farther than 2n (or the beginning), since `U[catalan-2n]` does not drop below y = 1 for a long time!

Performances

Score	B1	B2	B3
A1	44	64	92
A2	48	76	100

Takeaways

- Always figure out what the theoretical best is, so you have some basic idea on, at least, the **asymptotic** behaviour of the solution

- Try to implement in functions and **separate** the encoding / decoding in **several steps** - otherwise it would be painful to figure out what happens when the output is some random stuff

- Although you can implement something like bigint and FFT to water some marks in this task, always think if there are **easier ways** to get the points before committing a 200-line solution!

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

M2643 Positive Or Negative

Nicholas Ko {__jk__}
2026-06-28

Background

- Problem idea by `__jk__`
- Preparation by `__jk__`, `__declspec`, `hywong1`, `firewater` and various AIs

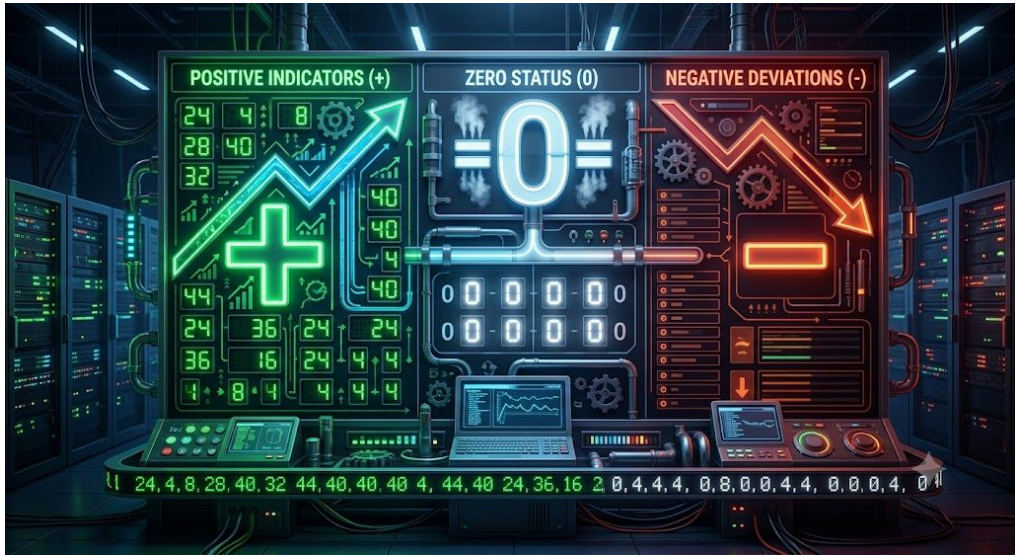
- Probably one of the most toxic minicomp problems so far
- It is a **much** harder version of NOI 2017 整數.
  - There are two different solutions to the NOI problem. This problem requires both ideas (and obviously something more too).

The Problem

- Given a tree of N nodes rooted at node 1
- Each edge i has a comfort level C[i] = X[i] × pow(2, Y[j]) with X[i], Y[i] specified in input
- Q queries: consider the simple path e[1], ..., e[k] from node U to V, is ∑(pow(2, i) * C[e[i]]) larger / equal / smaller than 0?

Statistics

- ~~The problem setter thought this problem was funny when proposing, turns out it backfired horribly during preparation :((~~



Test Case 1 - 6: ∑N, ∑Q ≤ 2000

- For each query, we can naively trace through the simple path from U to V
- How do we keep track of the sum?
- We can use two maps to store the sum of positive/negative terms
  - Each C[e[i]] * 2^i term has at most log |X[i]| bits for a total of O(N log |X|) bits
- Compare the two sums to find the sign of `sum_pos - sum_neg`
- For cases 1-3, an array can be used instead of a map
- Time Complexity: O(QN log(|X[i]|) log(N log(|X[i]|)))
- Score: 24

Test Case 4, 7, 11, 16, 17: Only need to determine equal to 0

- We denote, for each node, two sums: pathup and pathdown
- pathup[v] is the sum from node v to node 1 (the root)
- pathdown[v] is the sum from node 1 to node v
- Then for a query (u, v) and l = LCA(u, v), the sum can be decomposed into:

  pathup[u] - pow(2, dep(l)) × pathup[l]
  + pow(2, (dep(v) - 2 × dep(l))) × (pathdown[v] - pathdown[l])

Test Case 4, 7, 11, 16, 17: Only need to determine equal to 0

- Since we only care if the sum is 0, we can maintain a hashed version of each pathup, pathdown and compute the hashed sum
  - pathup, pathdown can be computed during the DFS run, as both values of a node can be expressed in terms of that of its parent
- If the hashed sum is 0 we report 0 and otherwise any sign
- Time Complexity: O(Qlog(N))
- Score: 20 (Cumulative: 40)

Test Case 7 - 10: U = 1 for all queries

- We only need to compute the sign of pathdown[v] for each node
- To compute that, we can perform a DFS and maintain a bigint value of the sum
- We use Trygub Numbers (named after Anton Trygub) – a binary representation but each digit can be one of {-1, 0, 1} instead of only {0, 1}
  - The purpose is to get a better amortised complexity for additions and subtraction by sacrificing the uniqueness of binary representations.
- The bigint sum can be maintained using a (sparse) segment tree with Trygub numbers, with each leaf node representing the value of a digit
- The sign of the whole sum is equal to the sign of the highest non-zero bit

Test Case 7 - 10: U = 1 for all queries

- How do we actually operate on the segment tree?
- We first decompose the update into log(|X[i]|) single-digit updates
- To update the i-th digit to value x, we search for the first digit $j \geq i$ such that the j-th digit has a different sign than x
- We then set every digit in [i, j) to 0 and increment/decrement digit j by 1
- Since each digit can only be set to 0 when it is assigned a non-zero value, the time complexity is bounded by #operations
- Time Complexity: amortized O(M log M (compute) + Q (queries)), where M = O(N log(|X[i]|)) is the number of single-digit updates
- Score: 16 (Cumulative: 52)

Test Case 11 - 15: A[i] = i, B[i] = i + 1 for all 1 ≤ i < N

- Only one of the pathup/pathdown component will be non-zero
- WLOG assume U < V, and we only need to consider pathdown
- The query sum is (pathdown[V] - pathdown[U]) / pow(2, U - 1)
- The sign is the same as the numerator as denominator is always positive
- Issue 1: Unlike the previous subtask, we will also have to store the values of pathdown
- Issue 2: How do we actually retrieve the sign of pathdown[V] - pathdown[U]?

Test Case 11 - 15: A[i] = i, B[i] = i + 1 for all 1 ≤ i < N

- Issue 1 can be solved by introducing persistence into the segment tree
- Instead of overwriting values, we create new nodes for the updated digits
- By a similar amortized argument, the number of nodes introduced from carry will be bounded by O(M log M) for a total precomputation time of O(M log M (log (log M)))

Test Case 11 - 15: A[i] = i, B[i] = i + 1 for all 1 ≤ i < N

- How about issue 2?
- Denote T[U], T[V] to be the Trygub values at nodes U, V respectively, and a dp value with dp[i] = $(\sum 2^j (T[V][j] - T[U][j])) / 2^i$ for all $j \geq i$
- The answer to the query is then the sign of dp[0]
- The dp values have the recurrence dp[i-1] = (T[V][i-1] - T[U][i-1]) + 2dp[i]
- Notice if $|dp[i]| \geq 2$, then the sign of dp[i-1] is the same as dp[i] and $|dp[i-1]| \geq 2$ will also hold
- Therefore we only need to find the smallest i such that $|dp[i]| < 2$, and compute dp[i-1]

Test Case 11 - 15: A[i] = i, B[i] = i + 1 for all 1 ≤ i < N

- How does this actually help us?
- dp[i] taking value in {-1, 0, 1} is equivalent to $\sum 2^j$ (T[V][j] - T[U][j]), j ≥ i, taking value in $\{-2^i, 0, 2^i\}$
- We can maintain a (hashed) subtree sum at each node of the segment tree, and perform segment tree descent to find the first digit i that satisfies this
- Score: 20 (Cumulative: 68)

Test Case 2, 8, 16 - 19: $|X[i]| \leq 1$

- Our amortized complexity analysis doesn't hold anymore, as each non-zero digit introduced at a node can now be set to 0 in multiple copies
  - Amortised data structures will fall apart when it meets persistence
- We can instead perform lazy propagation and maintain two additional tags at each node: is_all_1 and is_all_-1
- When we perform a digit update, we lazily set the carry 0s
- The subtree sum can still be maintained under lazy as the lazied subtree must have sum 0

Test Case 2, 8, 16 - 19: |X[i]| ≤ 1

- Without the special constraints, our query will also have to include 4 Trygub terms now instead of 2
- We derive the dp values similar to the previous subtask, but this time we descent for the first |dp[i]| ≤ 3 instead: use hashing to check whether the value is between -3 and 3 inclusive
- Score: 24 (Cumulative: 76)

Full Solution

- The previous solution is in theory correct but too slow with log(|X[i]|) updates when popcount(|X[i]|) is large
- Instead of storing the value of an individual digit at the leaf, we can store chunks of 64 digits (i.e. each leaf of the segment tree stores 64 digits, so we have 6 fewer layers)
- This way we can perform O(1) updates instead of O(log(|X[i]|)) updates, and the saved log term is enough for the solution to AC
- Score: [60, 100] depending on constant factors

Optimisations

Chances are that you likely will get constant factor-ed when submitting for the first time (and in contest this is unfortunately your only time), here are a few optimisations:

- Precompute powers of 2 to retrieve them in O(1) time
- Since the only range set update we are doing is setting to 0, we “destroy” the node (i.e. map it to the empty node) instead of creating new ones
- If you are using vectors, use std::reserve to preallocate memory
- Early break more when searching on segment trees

Takeaways

- Especially when you know your code is going to be long and messy, try to modularise your code for easier debugging
- Test cases are not necessarily sorted in ascending difficulty order (as is the case for subtasks), water points strategically
- Occasionally there can be implementation heavy tasks in IOI / NOI, and I hope this gave you an experience to plan your time ahead accordingly
- Wish you good luck, patience, and perseverance when upsolving :)