M2541 - Time Travelling

Bosco Wang {happypotato}

2025-06-29

Problem Background

Problem Idea by `ethening`

Preparation by `happypotato`

Presented by `happypotato`

The Problem

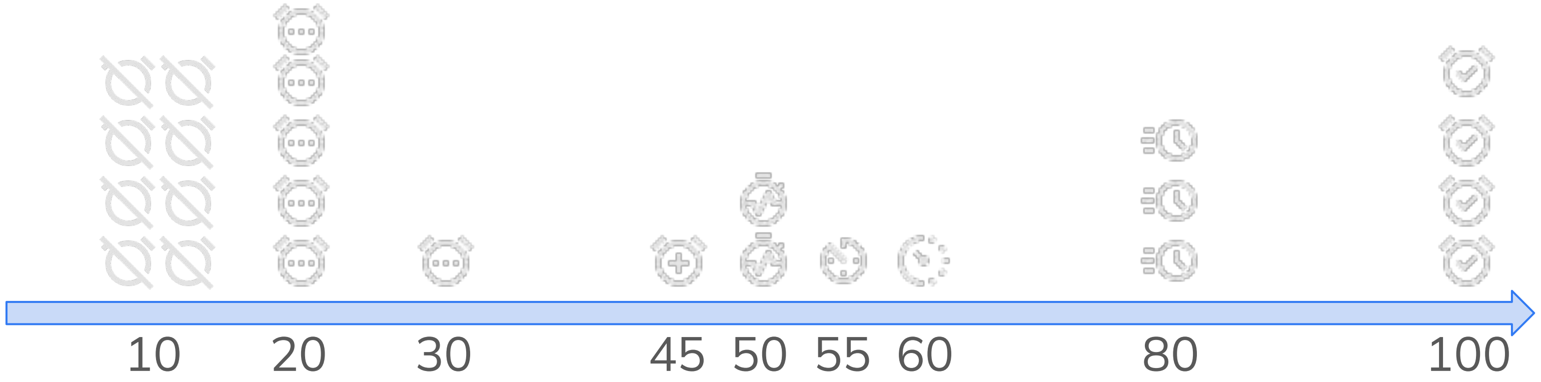
Support these operations:

- After command t, add edge (u, v)
- After command t, remove edge (u, v)
- After command t, query if nodes u and v are in the same connected component

Cases

- [Cases 1-2] N, Q <= 2000
- [Cases 3-4] $t_i = i - 1$, op ≠ 2
- [Cases 5-7] $t_i = i - 1$, remove operations must be on the last added edge
- [Cases 8-12] $t_i = i - 1$
- [Cases 13-16] op ≠ 2
- [Cases 17-20] No additional constraints

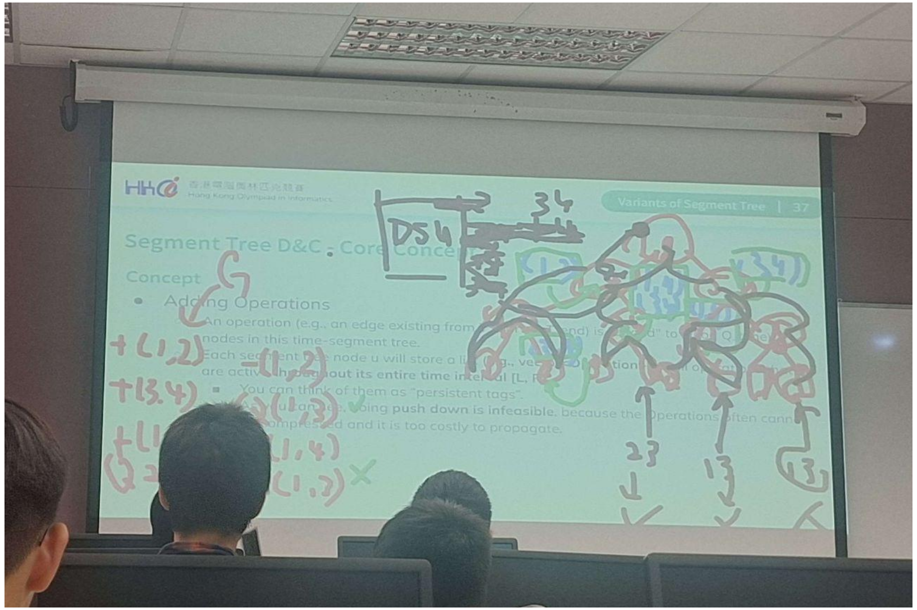
Statistics



First solved by **s20251** at 0:48

Before we begin…

Did you pay attention to the lecture yesterday?


Segment Tree D&C : Core Concept
Concept
Adding Operations
Variants of Segment Tree | 37
DSU
+(1,2)
+(3,4)

[Cases 1-2] N, Q ≤ 2000

The constraints allow for an O(QN log N) solution.
For each version, store a set of current edges. Adding and removing edges is easy here.
Then for each query, run any graph traversal algorithm on the resulting graph.

Time complexity: O(QN log N)

Expected Score: 10

[Cases 3-4] $t_i = i - 1$, no remove operations

There are no remove operations, and there is just one “branch” of graphs.

This is a standard Disjoint Set Union task. Refer to https://assets.hkoi.org/training2025/ds-ii.pdf for details on Disjoint Set Union.

As a recap, it supports the following in amortised O(α(N)) ≈ O(1) time:

- Union(u, v) - Connect nodes u and v; and
- Find(u) - Find the component node u belongs to, usually represented by a “head”
- To check if u and v are in the same connected component, do find(u) == find(v).

Expected Score: 20

[Cases 5-7] $t_i = i - 1$, remove operations must be on last added edge

Now we have remove operations on the last added edge. A normal DSU doesn’t work now (why?).

We can use DSU **with rollback**.

What is DSU with rollback?

Let’s recap on what DSU is. Here is a typical implementation of DSU:

```
struct DSU {
    vector<int> par, sz;
    DSU(int n): par(n, -1), sz(n, 0) {}
    int Find(int u) {
        if (par[u] == -1) return u;
        int head = Find(par[u]);
        // optimisation 1: path compression
        par[u] = head;
        return head;
    }
```

What is DSU with rollback?

Here is a typical implementation of DSU:

```
    void Union(int u, int v) {
        u = Find(u);
        v = Find(v);
        if (u != v) {
            // optimisation 2: union by size
            if (sz[u] < sz[v]) swap(u, v);
            par[v] = u; sz[u] += sz[v];
        }
    }
};
```

What is DSU with rollback?

```
// optimisation 1: path compression
par[u] = head;

// optimisation 2: union by size
if (sz[u] < sz[v]) swap(u, v);
```

When both optimisations are applied, the amortised time complexity is O(α(N))
When either optimisation is applied, the amortised time complexity is O(log N)
When neither optimisation is applied, the worst case can be O(N) (line graph).

What is DSU with rollback?

How do we support rollback?

In this specific case, we are supporting a *stack-like* rollback, undoing the most recent update.

What do we do? Simply maintain a stack on what values are updated in the previous `merge()` calls, then revert it during a single operation rollback.

For example, when calling `merge(1, 2)`, you update `par[2]` from `-1` to 1. Then, you store {2, `-1`} on the stack (of vectors), then when calling `rollback()`, you revert the changes.

What is DSU with rollback?

```
// optimisation 1: path compression
par[u] = head;

// optimisation 2: union by size
if (sz[u] < sz[v]) swap(u, v);
```

What happens when we try to support rollback?

If we use optimisation 1, rollback might take O(N) reverts.

If we use optimisation 2 only, rollback only takes O(1) reverts!!!

What is DSU with rollback?

So, what is DSU with rollback?

It is just a DSU, with the following modifications:

- Remove path compression optimisation
- Maintain a stack of vectors, and for each `merge()` call, push a vector storing what variables changed and its value **before changing**
- For each `rollback()` call, simply undo all the changes from the top of the stack.

[Cases 5-7] $t_i = i - 1$, remove operations must be on last added edge

Therefore, you may simply implement a working dsu with rollback.

Time Complexity: O(N log N)

Expected Score: 35

[Cases 8-12] $t_i = i - 1$

Now, it is no longer guaranteed that the rollbacked edge is the last edge that gets added. How can we handle that?

Segment Tree D&C!

Maintain the **timespan** for each edge, then build a segment tree on time.

Given that you know how to do DSU with rollback, you can simply perform an Euler tour on the segment tree to answer the queries.

Time Complexity: $O(N \log^2 N)$

Expected Score: 60

[Cases 8-12] $t_i = i - 1$

This is actually a standard problem (offline dynamic connectivity), and the solution was discussed in yesterday's *Variants of Segment Tree* lecture.

You can refer to that lecture notes for details on Segment Tree D&C.

[Cases 13-16] op ≠ 2

There are no remove queries, but we have multiple “branches” of graphs.

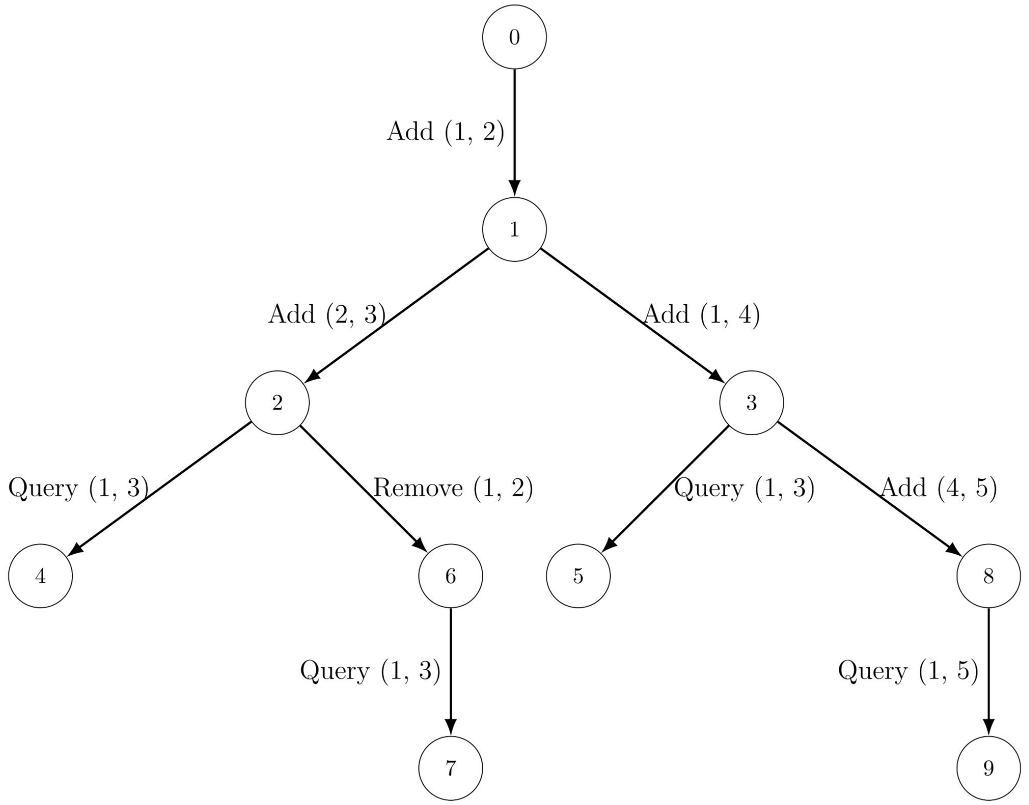
Usually, when dealing with multiple “versions”, we usually generate a “version tree”. It is a tree with each node being a “version”, and edges denoting the transition between versions.

Let’s take a look at an example.

[Cases 13-16] op ≠ 2

Consider Sample 1:

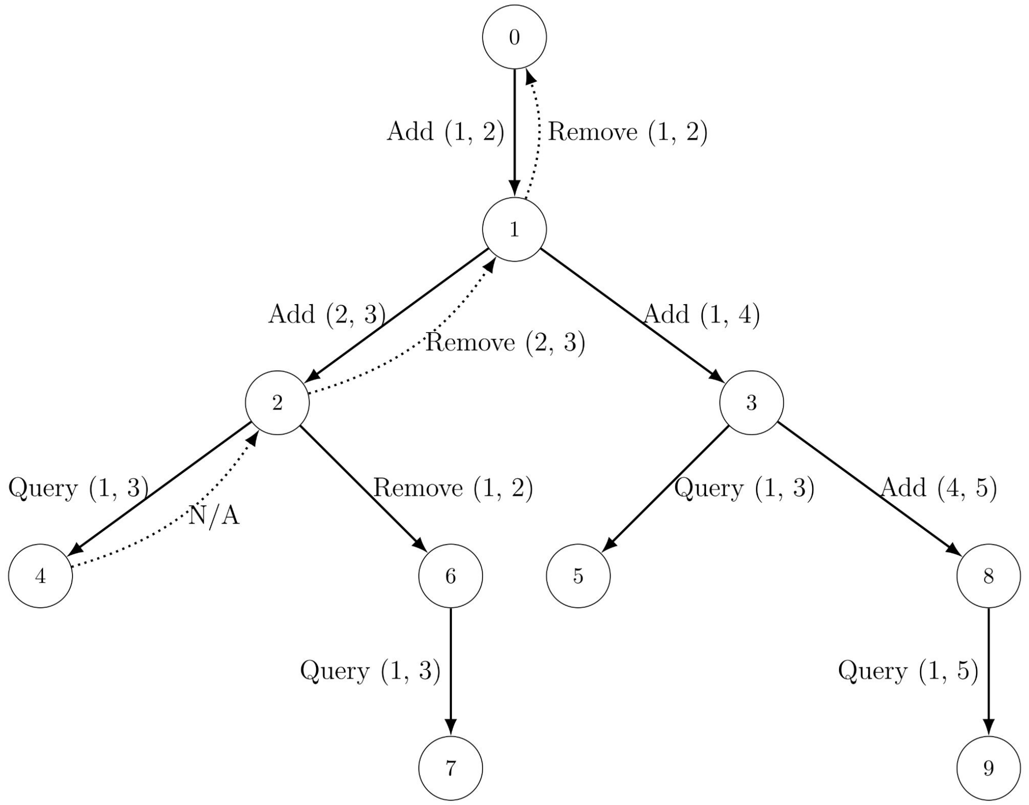
```
5 9
1 0 1 2
1 1 2 3
1 1 1 4
3 2 1 3
3 3 1 3
2 2 1 2
3 6 1 3
1 3 4 5
3 8 1 5
```



[Cases 13-16] op ≠ 2

Consider Sample 1:

```
5 9
1 0 1 2
1 1 2 3
1 1 1 4
3 2 1 3
3 3 1 3
2 2 1 2
3 6 1 3
1 3 4 5
3 8 1 5
```



[Cases 13-16] op ≠ 2

As you can see, by performing another Euler tour on the version tree and "undoing" the operation when moving back, you can reduce this to $t_i = i - 1$.

Note that this does not preserve the output order. More specifically, the order of queries you handle when traversing along the Euler tour might be different than the order during input, so you should add some query id to all queries.

Time Complexity: $O(N \log N)$

Expected Score: 80

[Cases 17-20] No additional constraints

Simply combine the solution for $t_i = i - 1$ and op $\neq 2$.
First, perform an Euler tour on the version tree. Then the problem is reduced to $t_i = i - 1$ again, and we can use the standard offline dynamic connectivity solution.

Time Complexity: $O(N \log^2 N)$

Expected Score: 100

M2542 Rainbow Cards

Hsieh Chong Ho {QwertyPi}

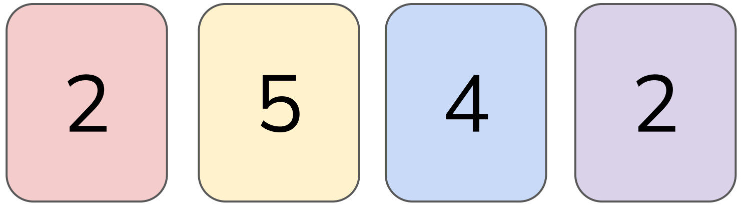
2025-06-29

Problem Background

Problem Idea by `QwertyPi`

Preparation by `gasbug` and `QwertyPi`

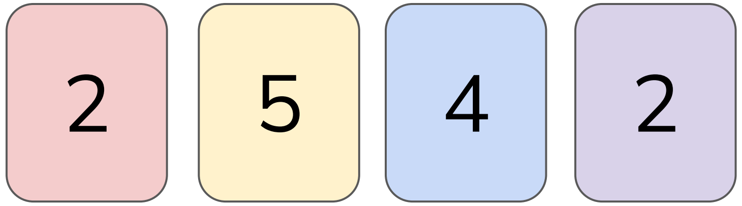
Presented by `QwertyPi`



The Problem

Each card contains two sides.

Each side contains a colour and a number.



Draw N cards one-by-one.

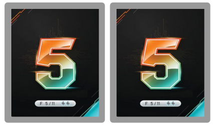
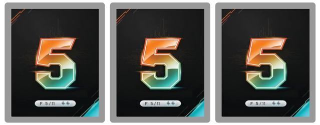
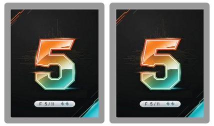
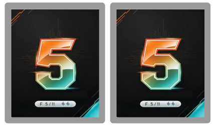
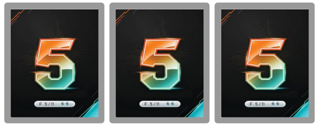
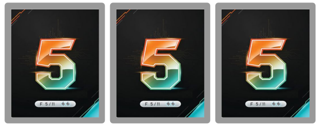
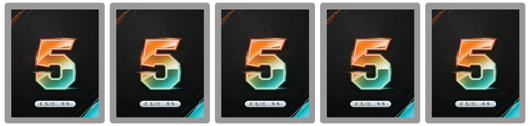
After you have drawn **each** of the N cards:

- Put down the drawn cards down on a table, with **colours shown distinct.**
- If this is possible, output the **maximum sum of all the values shown.**
- Otherwise, output -1.

Cases

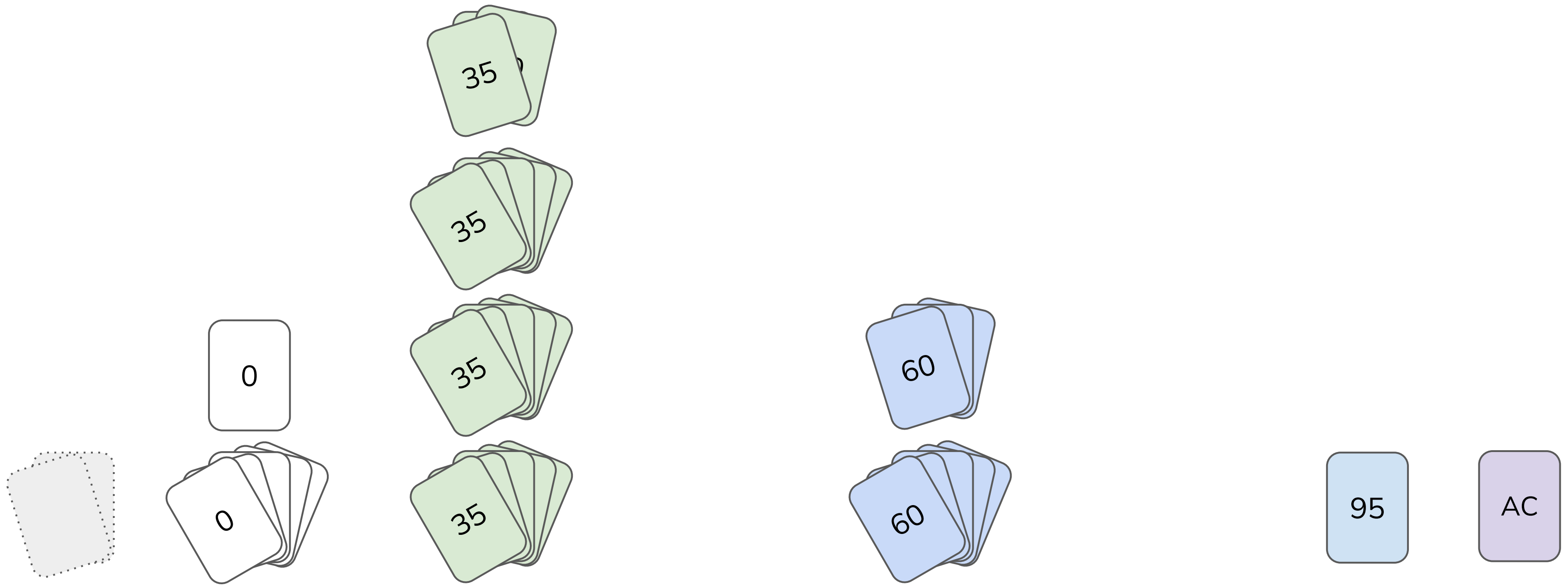
Chain*	$A[i] = i, B[i] = i + 1$ for $1 \leq i \leq N$
Tree*	$1 \leq A[i] < B[i] = i + 1$ for $1 \leq i \leq N$
Graph	No additional constraints

Cases

	$N \le 500$	$N \le 5000$	$N \le 2 \times 10^5$
Chain*			
Tree*			
Graph			

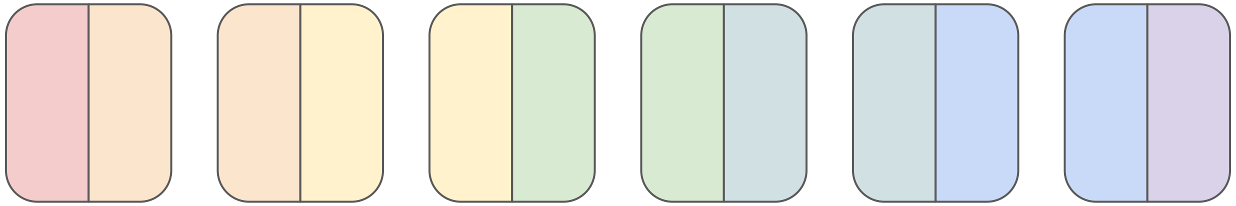
Statistics

Name	Rainbow Cards		
Author	Daniel Hsieh	**Tag**	Easy



Case 1 ~ 2 (Chain*, N ≤ 500, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)

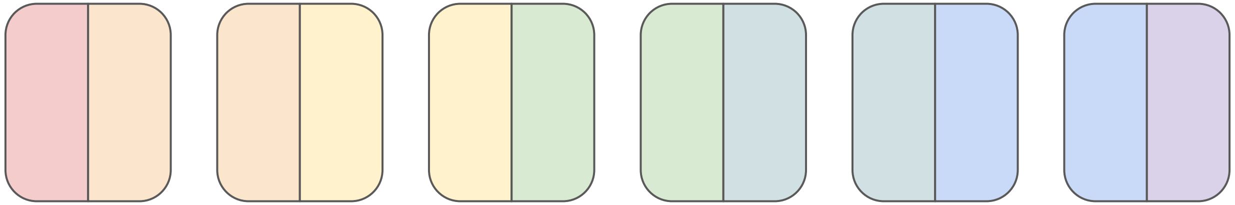
Card i (1 ≤ i ≤ N): One face with colour i, one face with colour i + 1.



We cannot show the same colour twice. When can we achieve that?

Case 1 ~ 2 (Chain*, N ≤ 500, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)

Card i (1 ≤ i ≤ N): One face with colour i, one face with colour i + 1.



We cannot show the same colour twice. When can we achieve that?

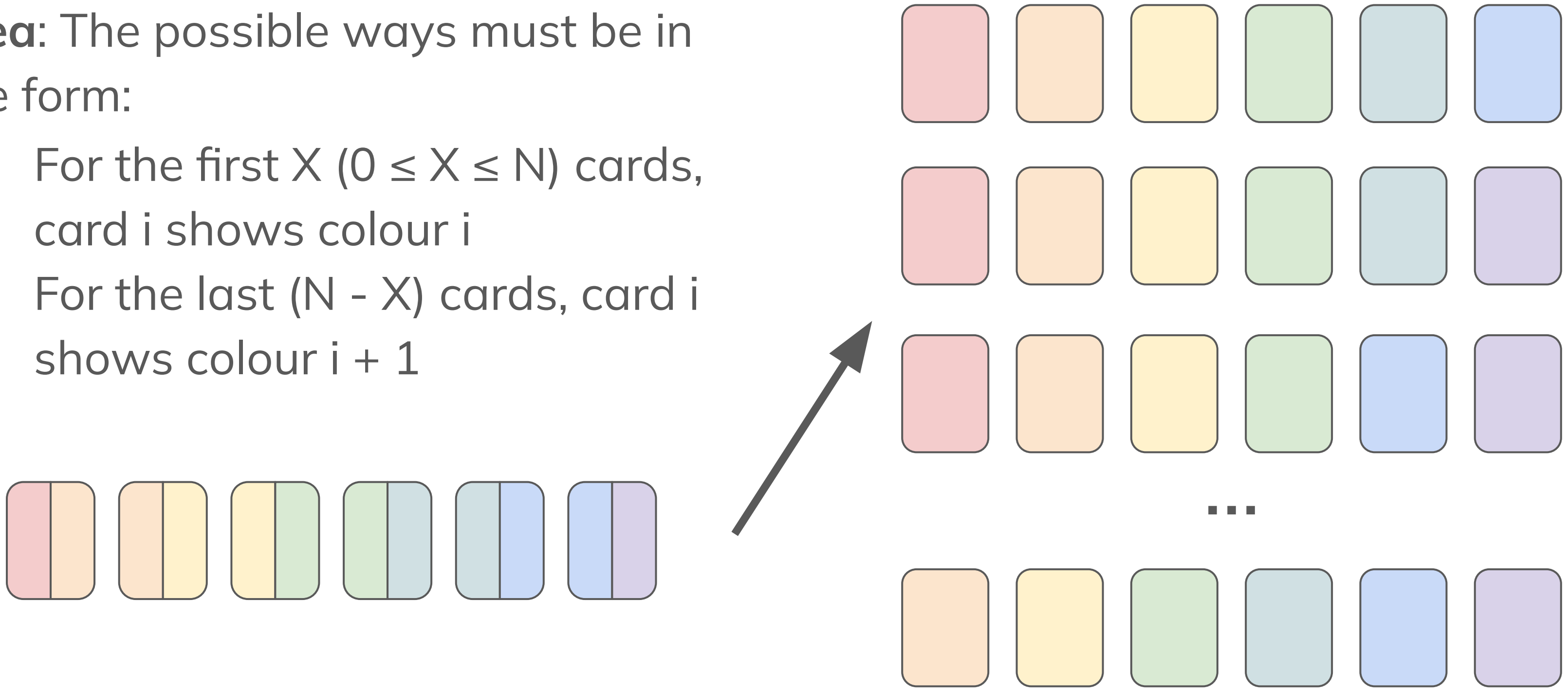
Idea: The possible ways must be in the form:

- For the first X (0 ≤ X ≤ N) cards, card i shows colour i
- For the last (N - X) cards, card i shows colour i + 1

Case 1 ~ 2 (Chain*, N ≤ 500, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)

Idea: The possible ways must be in the form:

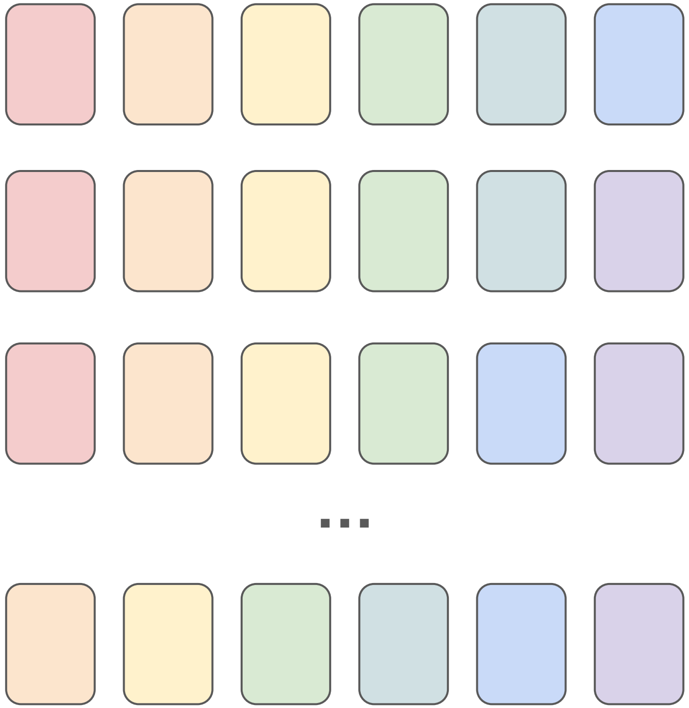
- For the first X (0 ≤ X ≤ N) cards, card i shows colour i
- For the last (N - X) cards, card i shows colour i + 1



Case 1 ~ 2 (Chain*, N ≤ 500, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)

For each prefix of K cards, we can loop through all the (K+1) different cases.

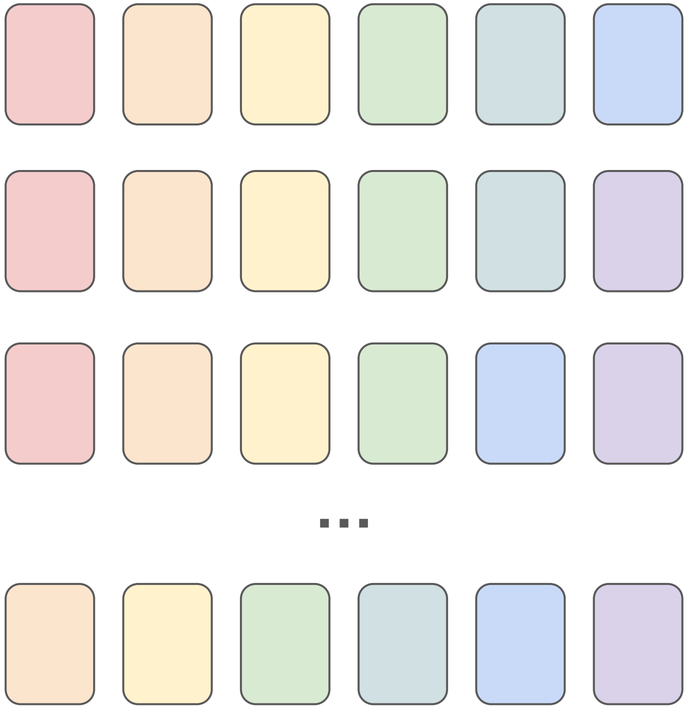
Time Complexity: $O(N^3)$
Expected Score: 10



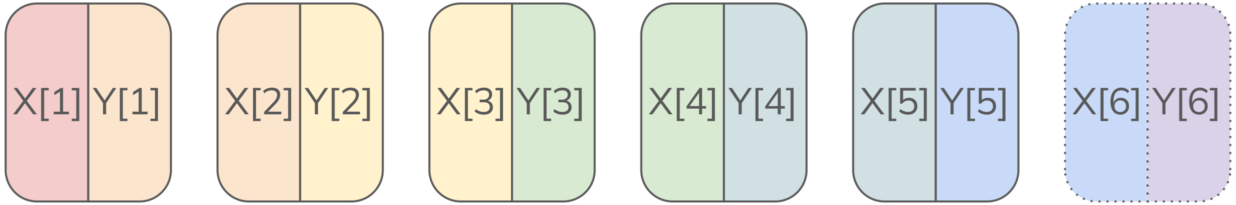
Case 3 ~ 5 (Chain*, $N \le 5000$, $A[i] = i$ and $B[i] = i + 1$ for $1 \le i \le N$)

Obviously, you can optimise the previous solution by partial sum.

Time Complexity: $O(N^2)$
Expected Score: 25



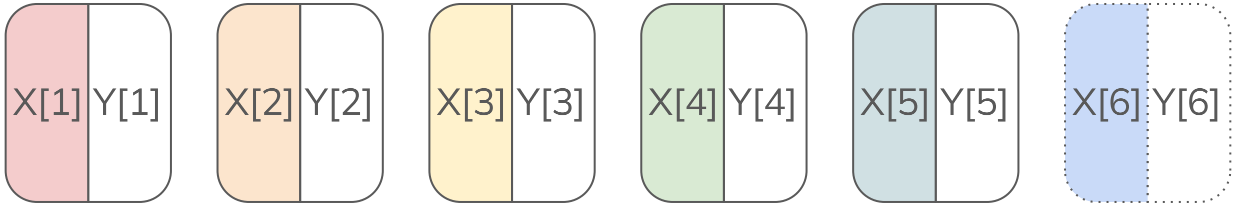
Case 6 ~ 7 (Chain*, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)



Let's say ans[K] is the answer for the first K cards.

What is the effect if we add card K?

Case 6 ~ 7 (Chain*, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)

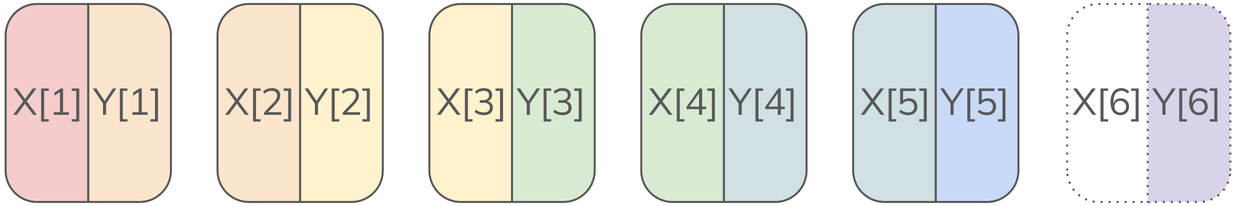


Let's say `ans[K]` is the answer for the first K cards.

What is the effect if we add card K?

- **Case I: First side picked, then every other cards are fixed**
  - Max sum: `X[1] + X[2] + … + X[K]`

Case 6 ~ 7 (Chain*, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)

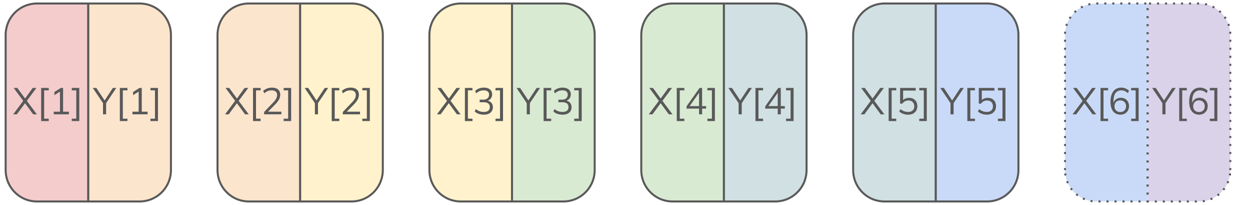


Let's say `ans[K]` is the answer for the first K cards.

What is the effect if we add card K?

- Case I: First side picked, then every other cards are fixed
  - Max sum: `X[1] + X[2] + … + X[K]`
- **Case II: Second side picked, then first (K - 1) cards form a sub-problem**
  - Max sum: `ans[K - 1] + Y[K]`

Case 6 ~ 7 (Chain*, A[i] = i and B[i] = i + 1 for 1 ≤ i ≤ N)



Therefore, we can solve the chain case with dynamic programming.

Transition formula:

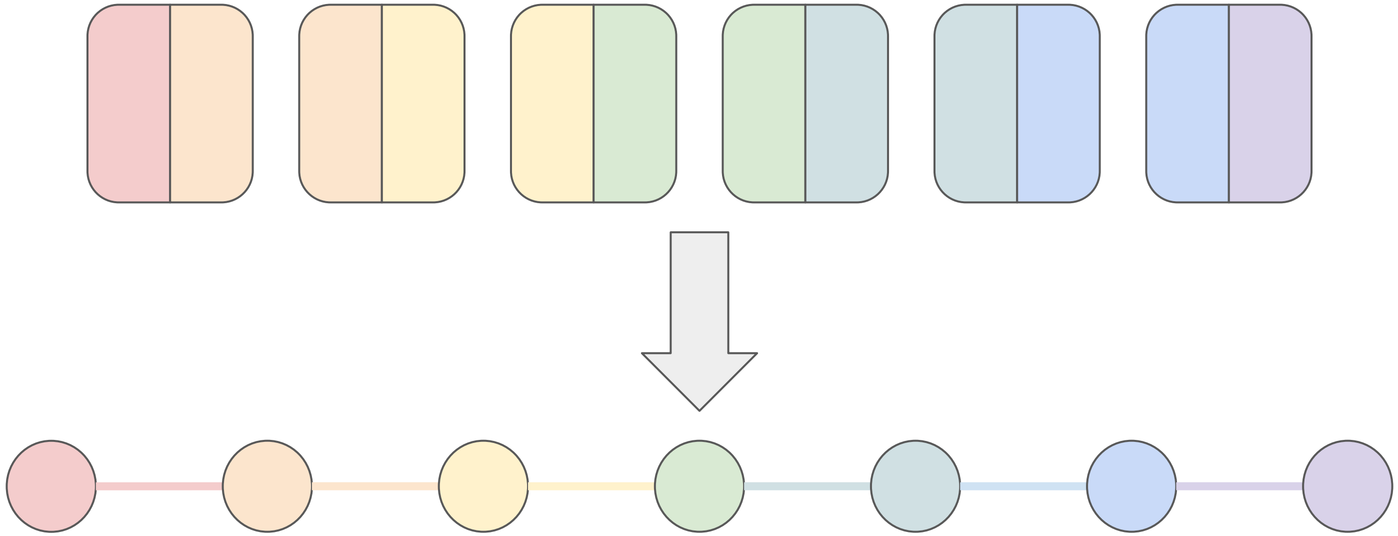
```
    ans[K] = max(X[1] + X[2] + … + X[K], ans[K - 1] + Y[K])
```

Time Complexity: O(N) (with partial sum)

Expected Score: 35

Case 8 ~ 9 (Tree*, $N \le 5000$, $1 \le A[i] < B[i] = i + 1$ for $1 \le i \le N$)

Actually, we can **treat each colour as a node** and **each card as an edge**. For each edge we assign it to one of its end, such that no two edges are assigned to the same node.

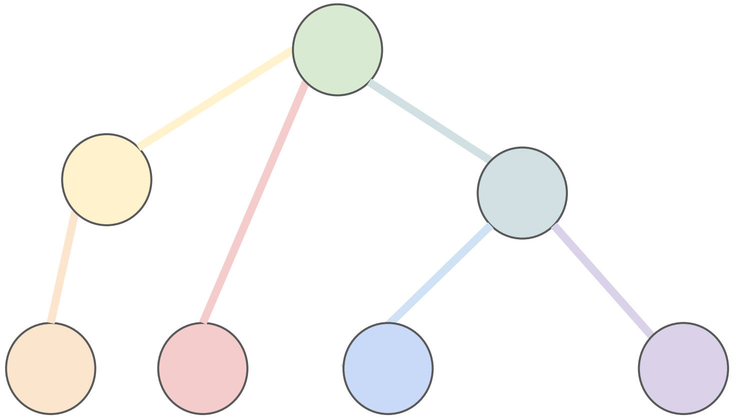


Case 8 ~ 9 (Tree*, $N \le 5000$, $1 \le A[i] < B[i] = i + 1$ for $1 \le i \le N$)

Define `dp[v][0/1]` to be the answer for the subtree rooted at node v, and whether or not node v has been used.

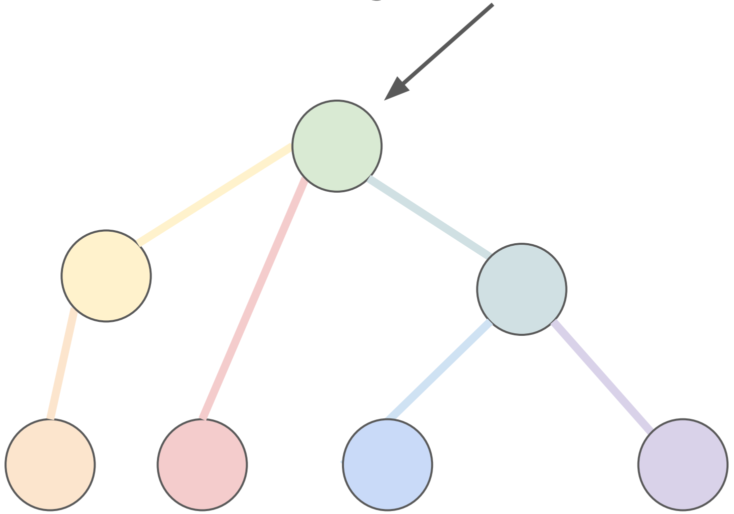
You may do a standard tree dp.
Transition formula left as exercise.

Time Complexity: $O(N^2)$
Expected Score: 35 (Cumulative: 45)



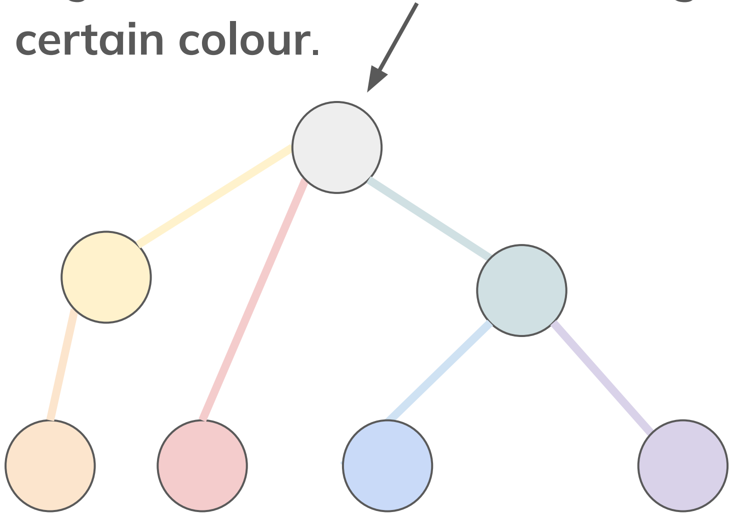
Case 10 ~ 12 (Tree*, $1 \le A[i] < B[i] = i + 1$ for $1 \le i \le N$)

Observation 1. If there are N colours (which are connected) and N - 1 card, then there must be exactly one colour isn't assigned to a card.



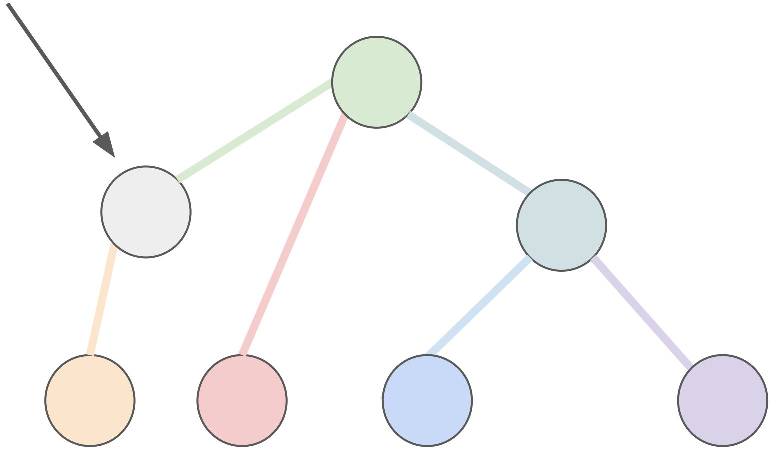
Case 10 ~ 12 (Tree*, $1 \leq A[i] < B[i] = i + 1$ for $1 \leq i \leq N$)

Observation 1*. If there are N colours (which are connected) and N - 1 card, then exactly one colour isn't assigned to a card, **and the configuration is fixed given that you don't choose a certain colour**.



Case 10 ~ 12 (Tree*, $1 \le A[i] < B[i] = i + 1$ for $1 \le i \le N$)

Observation 1*. If there are N colours (which are connected) and N - 1 card, then exactly one colour isn't assigned to a card, **and the configuration is fixed given that you don't choose a certain colour**.

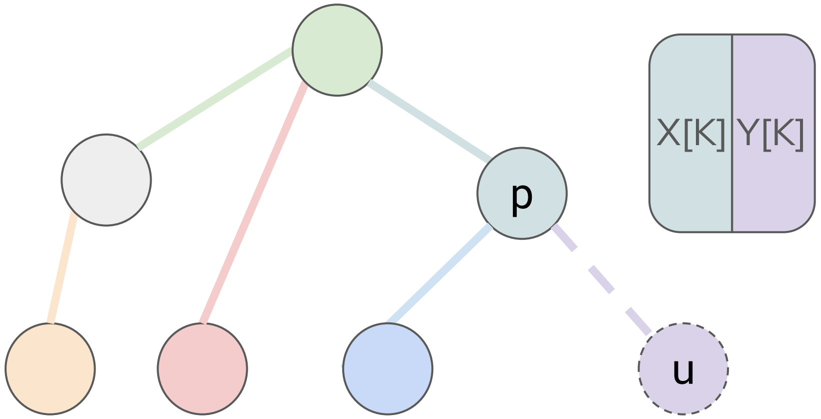


Case 10 ~ 12 (Tree*, $1 \leq A[i] < B[i] = i + 1$ for $1 \leq i \leq N$)

Key Idea: We can maintain the answer for each node, consider it not chosen.

Let's say ans[v] is the answer for **not** choosing node v.

How can we update ans after adding card K (i.e. an edge)?

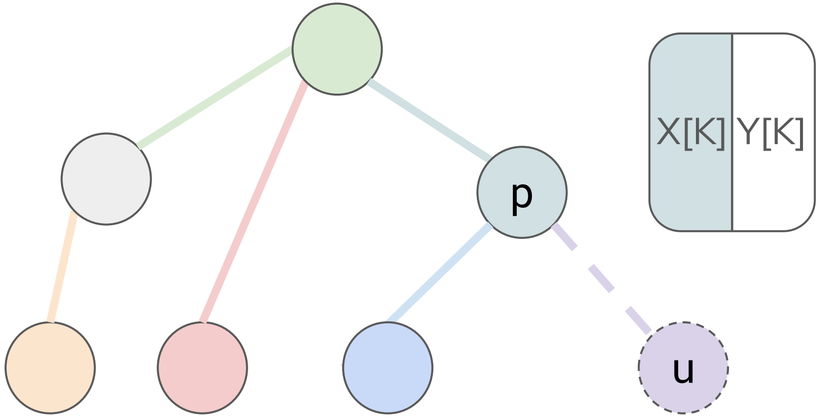


Case 10 ~ 12 (Tree*, $1 \le A[i] < B[i] = i + 1$ for $1 \le i \le N$)

Consider both cases where you pick node p and node u respectively:

Case I. If we pick node p, then node u must be the only unchosen.

```
ans’[u] = ans[p] + X[K]
```

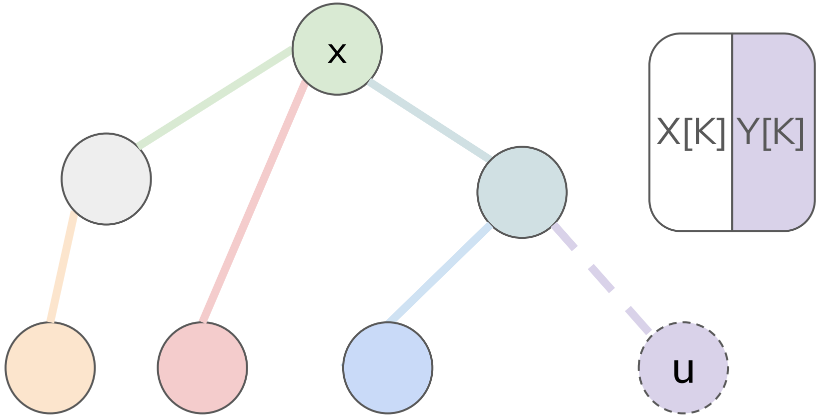


Case 10 ~ 12 (Tree*, $1 \leq A[i] < B[i] = i + 1$ for $1 \leq i \leq N$)

Consider both cases where you pick node p and node u respectively:

Case II. If we pick node u, then any other nodes can be unchosen. For all $x \neq u$,

```
ans'[x] = ans[x] + Y[K]
```



Case 10 ~ 12 (Tree*, $1 \le A[i] < B[i] = i + 1$ for $1 \le i \le N$)

Case I. If we pick node p, then node u must be the only unchosen.

`ans’[u] = ans[p] + X[K]`

Case II. If we pick node u, then any other nodes can be unchosen. For all $x \neq u$,

`ans’[x] = ans[x] + Y[K]`

Therefore, we can simply maintain a global `offset` value, increase `offset` by `Y[K]` and `ans[u]` by `X[K] - Y[K]`, while maintaining the maximum.

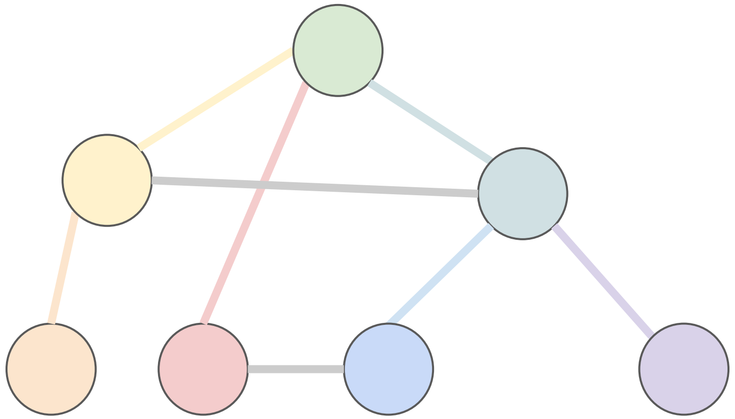
Time Complexity: O(N)

Expected Score: 60

Case 13 ~ 15 (Graph, N ≤ 5000)

There are some more observations to aid you in the general case.

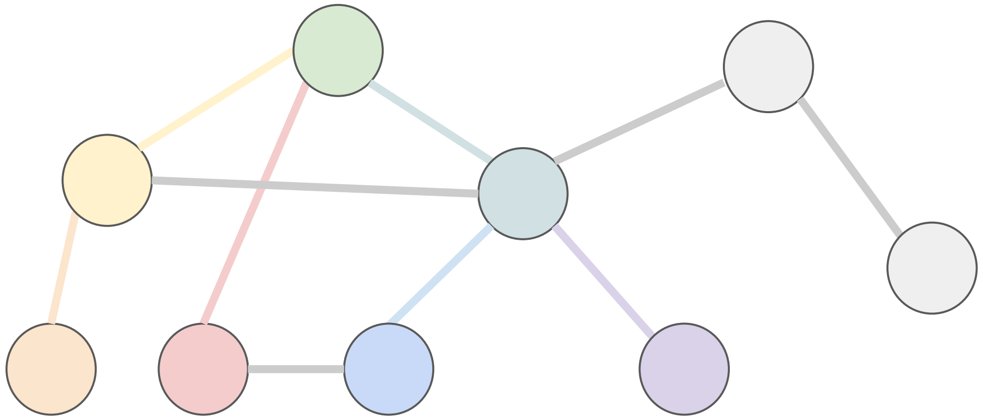
Observation 2. Consider a connected component. If there is more edges than nodes, there is no valid assignment for now, and every subsequent updates.



Case 13 ~ 15 (Graph, N ≤ 5000)

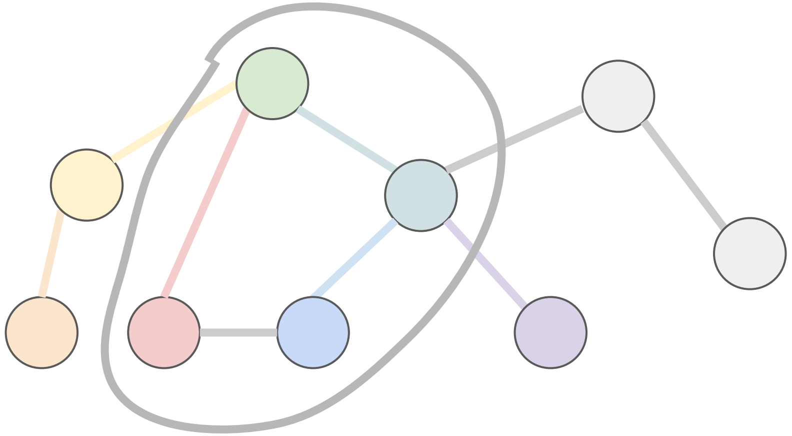
There are some more observations to aid you in the general case.

Observation 2. Consider a connected component. If there is more edges than nodes, there is no valid assignment for now, and **every subsequent updates**.



Case 13 ~ 15 (Graph, N ≤ 5000)

Observation 3. Consider a connected component. If the number of nodes equals the number of edges, then we can calculate an answer by trying the two ways to assign the **cycle**, then solve the “**branches**” using solution to tree.



Jellyfish Graph

Case 13 ~ 15 (Graph, N ≤ 5000)

If you naively re-compute for the answer of graph after each added cards, then you can solve this in O(N) per card added with tree dp.

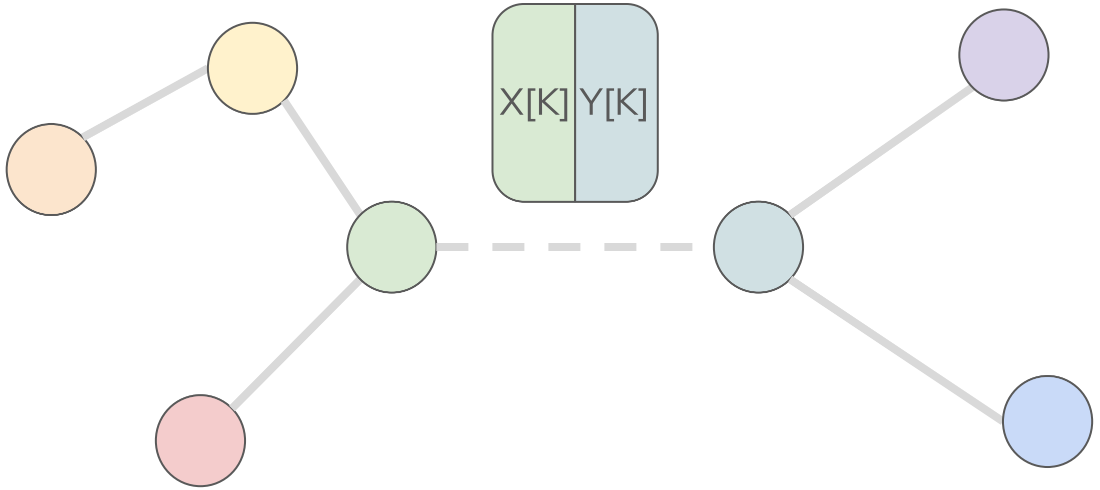
However, the implementation is slightly more complicated (than full solution).

Time Complexity: $O(N^2)$
Expected Score: 50 (Cumulative: 75)

Case 16 ~ 20 (Graph)

How to solve for the general case efficiently?

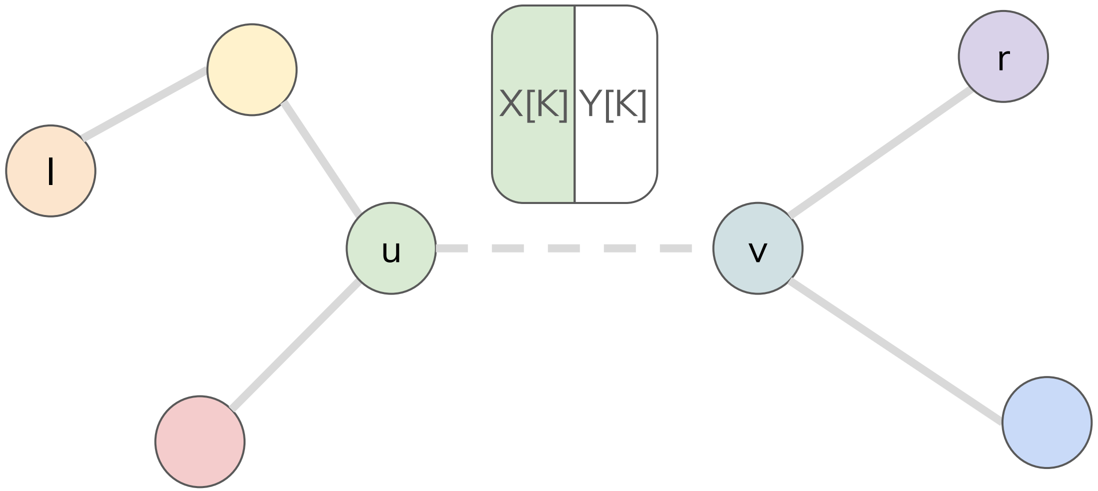
Actually, you can reuse the tree idea!



Case 16 ~ 20 (Graph)

Case I. If node u is picked, then left subtree configuration is forced:

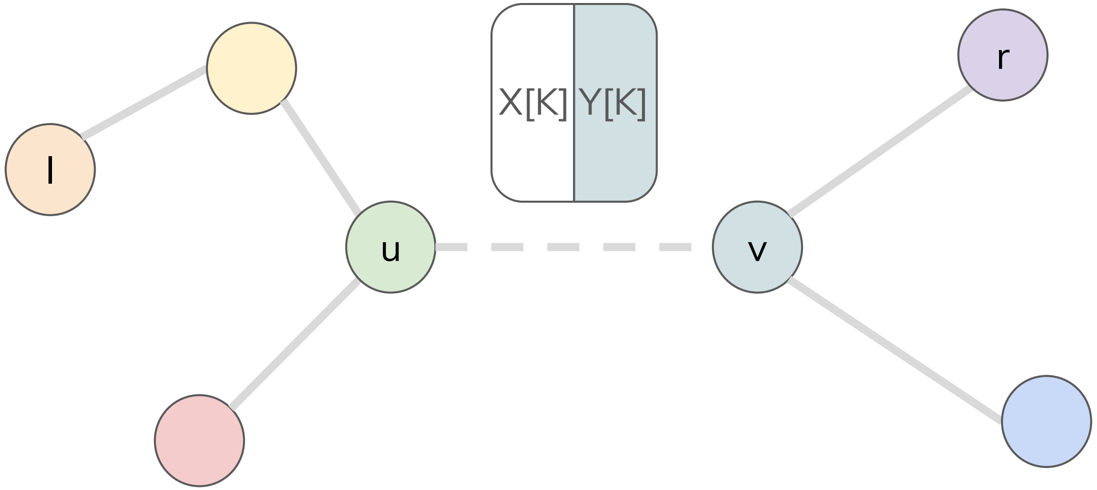
```
ans’[r] = (ans[u] + X[K]) + ans[r]
```



Case 16 ~ 20 (Graph)

Case II. Symmetrically, if node v is picked, then:

```
ans’[l] = (ans[v] + Y[K]) + ans[l]
```



Case 16 ~ 20 (Graph)

Case I. If node u is picked, then left subtree configuration is forced:

```
ans'[r] = (ans[u] + X[K]) + ans[r]
```

Case II. Symmetrically, if node v is picked, then:

```
ans'[l] = (ans[v] + Y[K]) + ans[l]
```

Note that the value added to left / right subtrees are constant, we can actually maintain this with a disjoint-set union.

Case 16 ~ 20 (Graph)

Obviously there are more to handle, but those implementation details are left as exercise for the readers :)

With some more case-handling on the tree structure when merging, you can get full score from this task.

Time Complexity: $O(N\alpha(N))$
Expected Score: 100

M2543 - Bar Chart

Isaac Chan {snowysecret}

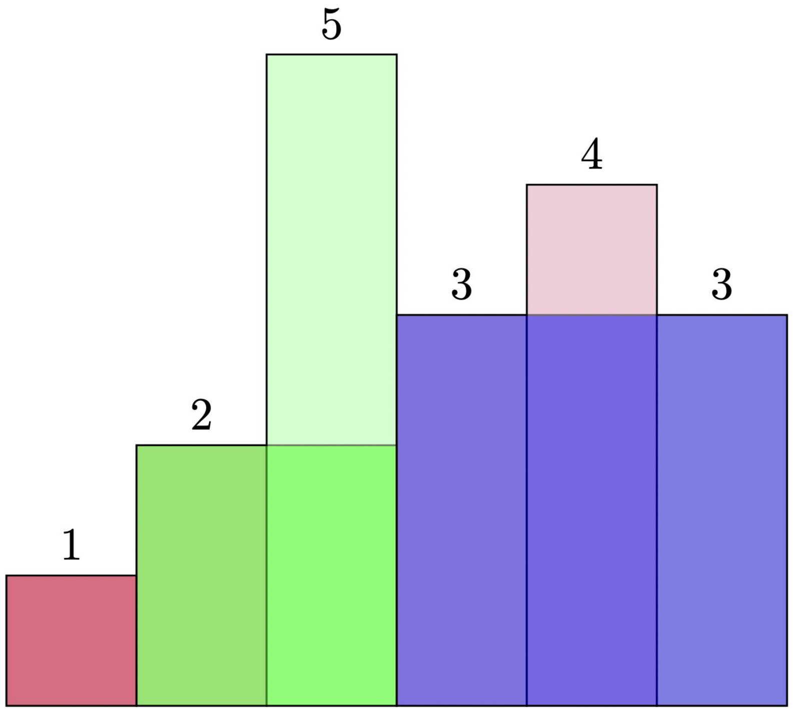
2025-06-29

Background

Problem idea by `snowysecret`

Preparation by `ethening, snowysecret`

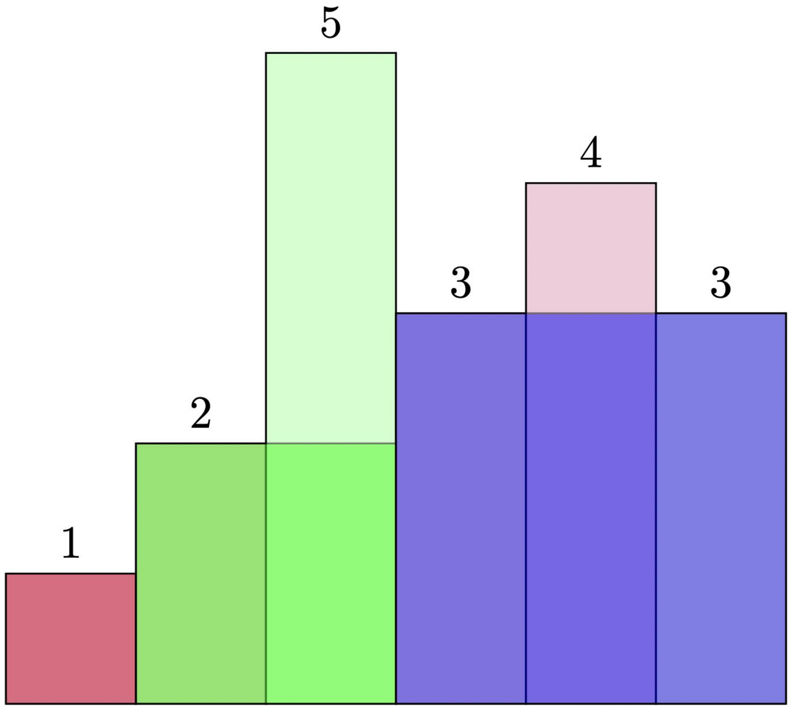
Presented by `snowysecret`



The Problem

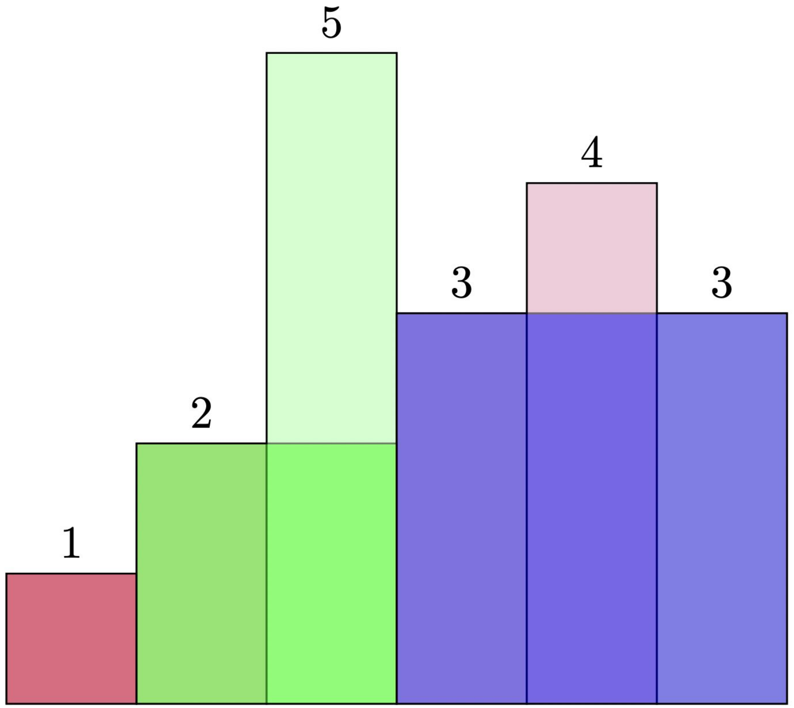
Given a bar chart with N columns, where the i-th column has height H[i], determine the maximum possible total area of all squares that can be placed within the bar chart, such that:

- Each square's bottom edge must rest on the bottom line of the bar chart.
- The square must not exceed the boundaries of the bar chart.
- Squares must not overlap one another.



Example

Optimal Solution: $1^2 + 2^2 + 3^2 = 14$

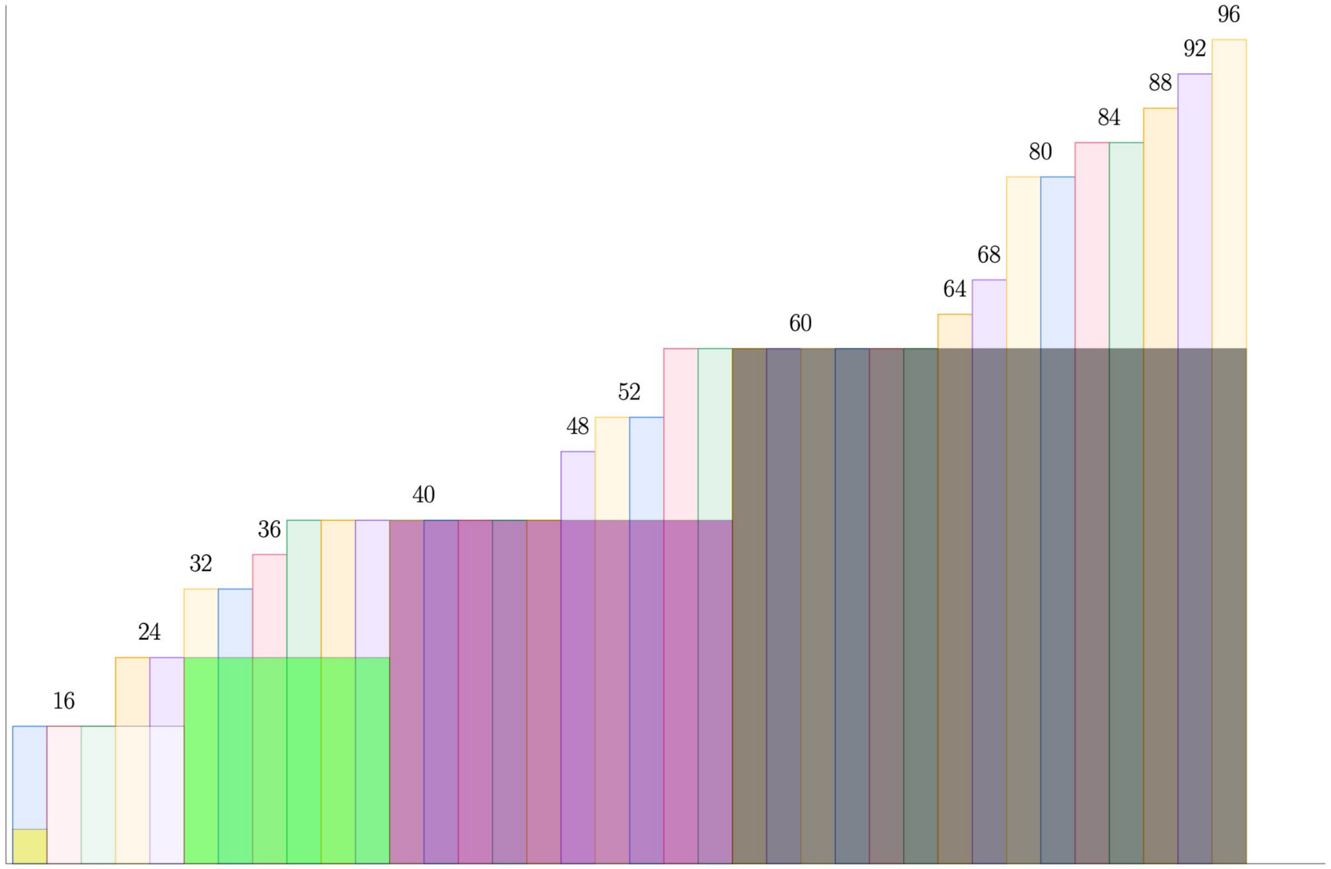


Cases

Many opportunities to obtain partial scores. Most notably, we have:

- [Cases 1-4] $N \le 10$, $H[i] \le 2$
- [Cases 5-6] $N \le 500$
- [Cases 7-10] $H[i] \le 100$
- [Cases 11-15] $H[1] \le H[2] \le \dots \le H[N]$
- [Cases 16-25] $10^5 \le N \le 10^6$, gradually increasing

Statistics



[Cases 1-4] N ≤ 10, H[i] ≤ 2

Here, the constraints are extremely small; we can use one of two approaches:

1) Greedy Algorithm
2) Brute Force Algorithm

For 1):

- Notice that for columns with H[i] = 1 the choice is obvious - just put a 1x1 square using this single column.
- For M consecutive columns with H[i] = 2, we can always pick 2 columns at a time, giving floor(M/2) 2x2 squares.

[Cases 1-4] $N \leq 10$, $H[i] \leq 2$

Here, the constraints are extremely small; we can use one of two approaches:

1) Greedy Algorithm
2) Brute Force Algorithm

For 2):

- We can brute force the "sequence of square lengths" $L_1, L_2, \ldots, L_K$.
- Notice that $L_1 + L_2 + \ldots + L_K = N$ for sure.
- After brute forcing each possible sequence $(L_1, L_2, \ldots, L_K)$, it remains to check whether we can actually place the squares without exceeding the limits of the bar chart.

Expected Score: 16

[Cases 5-6] N ≤ 500

For cases 5-6, N ≤ 500 and all H[i] ≤ N.
We can no longer rely on greedy / brute force techniques.

For the remainder of this question, we will need to use **Dynamic Programming**.

- What would the DP state be?

[Cases 5-6] N ≤ 500

For cases 5-6, N ≤ 500 and all H[i] ≤ N.
We can no longer rely on greedy / brute force techniques.

For the remainder of this question, we will need to use **Dynamic Programming**.

- What would the DP state be?
  Let's say
  DP[i] = {maximum area sum of squares only covering columns 1, 2, ..., i}

[Cases 5-6] $N \leq 500$

- What would the DP state be?
  Let's say
  DP[i] = {maximum area sum of squares only covering columns 1, 2, ..., i}

  The transition for DP[i] would be as follows:
- Check the square that contains column i.
- If such a square uses columns [j + 1, i], then the maximum area sum for the first i columns equals:
  The maximum area sum for the first j columns **+ $(i-j)^2$**

[Cases 5-6] N ≤ 500

The transition for DP[i] would be as follows:

- Check the square that contains column i.
- If such a square uses columns [j + 1, i], then the maximum area sum for the first i columns equals:
  The maximum area sum for the first j columns + $(i-j)^2$

This gives rise to the following transition formula:

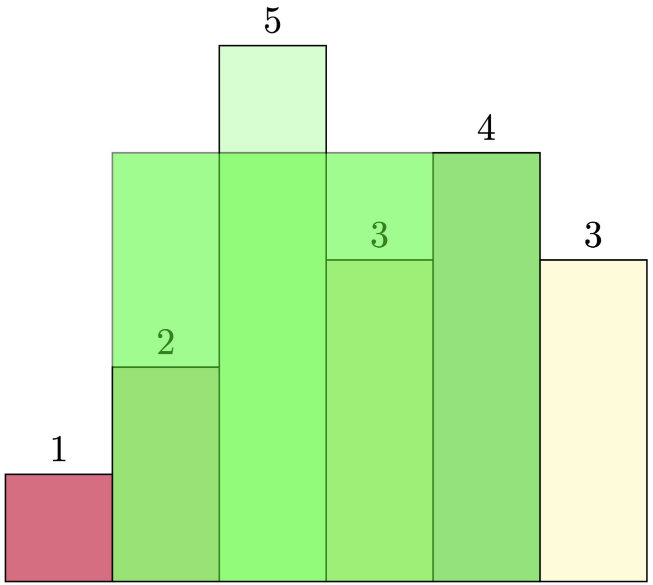
$$DP[i] = \max_{0 \le j < i} (DP[j] + (i-j)^2)$$

… is that it?

[Cases 5-6] N ≤ 500

$DP[i] = \max_{0 \le j < i} (DP[j] + (i-j)^2)$

- Note that this transition formula does not take into account the actual heights of the columns themselves, H[i].
- We might allow cases like the one on the right, allowing the square to not fit in the boundaries of the bar chart.



[Cases 5-6] $N \le 500$

We modify the formula, such that it ensures that the **minimum of column heights** is **at least the length of the square**.

$$DP[i] = \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} \left(DP[j] + (i-j)^2\right)$$

Naive implementation of this DP is $O(N^2)$, if we iterate $j = i-1, i-2, \ldots, 0$ and maintain the suffix minimum as we iterate.

Expected Score: 24

[Cases 7-10] H[i] ≤ 100

Look at the DP transition again:

$$DP[i] = \max_{0\leq j<i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \geq i-j} (DP[j] + (i-j)^2)$$

What happens when i-j > 100?

[Cases 7-10] H[i] ≤ 100

Look at the DP transition again:

$$DP[i] = \max_{0 \le j < i,\ \min(H[j+1], H[j+2], ..., H[i]) \ge i-j} (DP[j] + (i-j)^2)$$

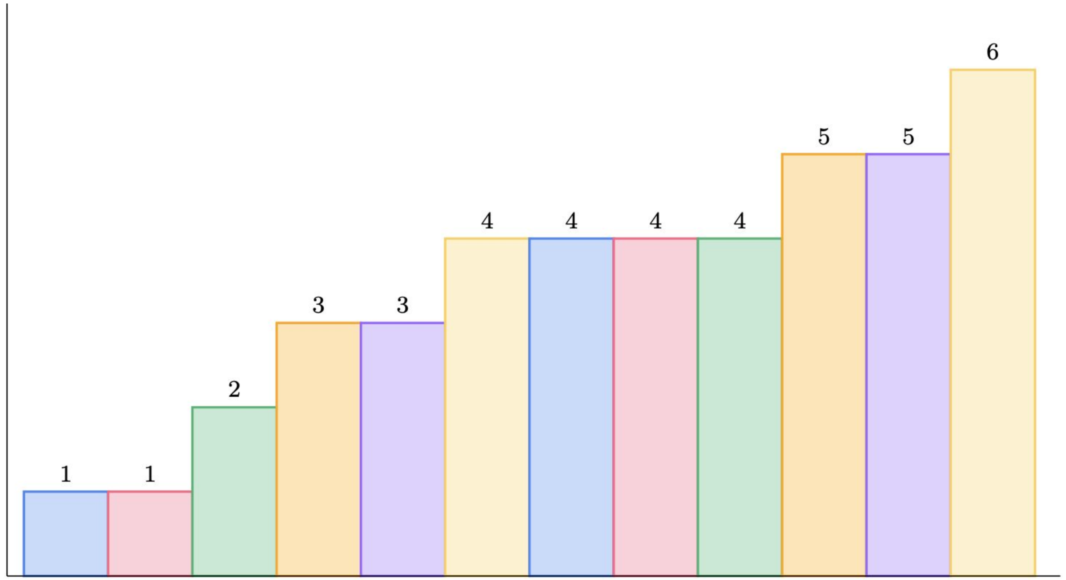
What happens when i-j > 100?

- Since H[i] ≤ 100, for sure min(H[j+1], H[j+2], ..., H[i]) < i-j
- Hence we don't need to consider any j such that j < i-100
- Hence we only need to consider j such that i-100 ≤ j < i (100 options!)
- Break after 100 previous indices, time complexity is O(100N)

Expected Score: 40

[Cases 11-15] H[1] ≤ H[2] ≤ … ≤ H[N]

What does this special condition mean?



[Cases 11-15] H[1] ≤ H[2] ≤ … ≤ H[N]

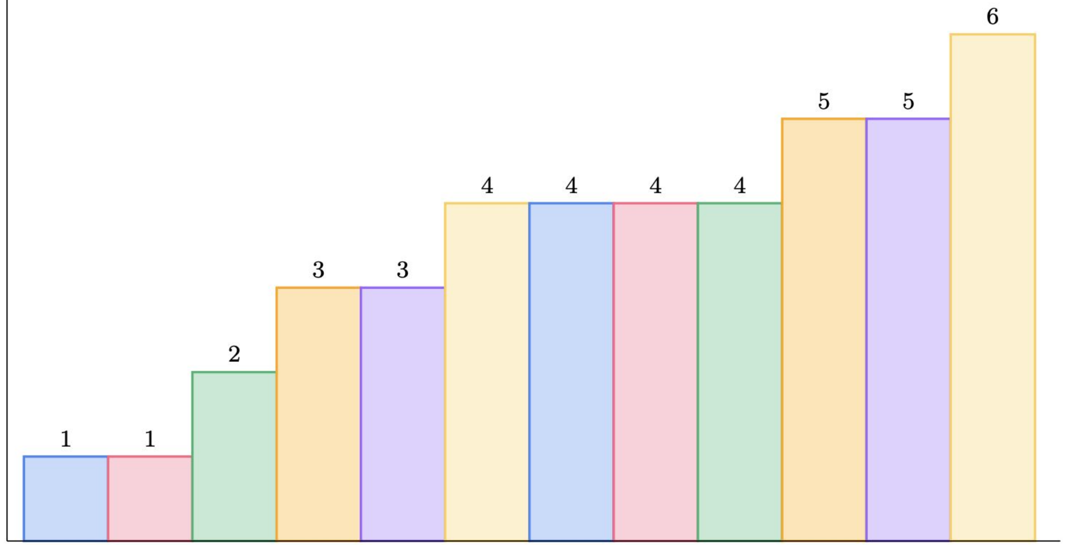
The constraints seem to suggest a greedy argument. We might want to:

- Use greedy to optimize the DP transition
- Just use a greedy algorithm, maybe?

[Cases 11-15] H[1] ≤ H[2] ≤ … ≤ H[N]

What does this special condition mean?

Hint: Begin by considering the square that covers the last column.



[Cases 11-15] H[1] ≤ H[2] ≤ … ≤ H[N]

Greedy strategy:

- We can consider the columns in reverse order (from N down to 1).
- Try to put as many columns inside the same square as possible, until it is impossible.
- This way, we can maximize the total area.
- You may try to think of why this is true.

[Cases 11-15] H[1] ≤ H[2] ≤ … ≤ H[N]

Therefore, we can just follow the aforementioned greedy algorithm.

The implementation is very short (example by contestant):

```
int curr = 0, ans = 0;
for (int i=n-1; i>=0; i--){
    if (h[i] < curr + 1){
        ans += curr * curr;
        curr = 1;
    } else {
        curr++;
    }
}
ans += curr * curr;
cout << ans << '\n';
return 0;
```

Expected Score: 60

[Cases 16-25] No additional constraints

No more additional constraints; we now need to solve the full task.
Test cases are given in gradually increasing N, from 10^5 to 10^6.

We will first look at how to solve the full task, then move on with some optimizations.

Revisiting DP

Look at the DP transition again:

$$DP[i] = \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} (DP[j] + (i-j)^2)$$

Can this DP be optimized?

Revisiting DP

Look at the DP transition again:

$$DP[i] = \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} (DP[j] + i^2 - 2ij + j^2)$$

Can this DP be optimized?

Revisiting DP

Look at the DP transition again:

$$DP[i] = i^2 + \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} (DP[j] - 2ij + j^2)$$

Can this DP be optimized?

Revisiting DP

Look at the DP transition again:

$$DP[i] = i^2 + \max_{0 \le j < i,\ \min(H[j+1], H[j+2], ..., H[i]) \ge i-j} (-2i)j + (DP[j] + j^2)$$

Can this DP be optimized?

Revisiting DP

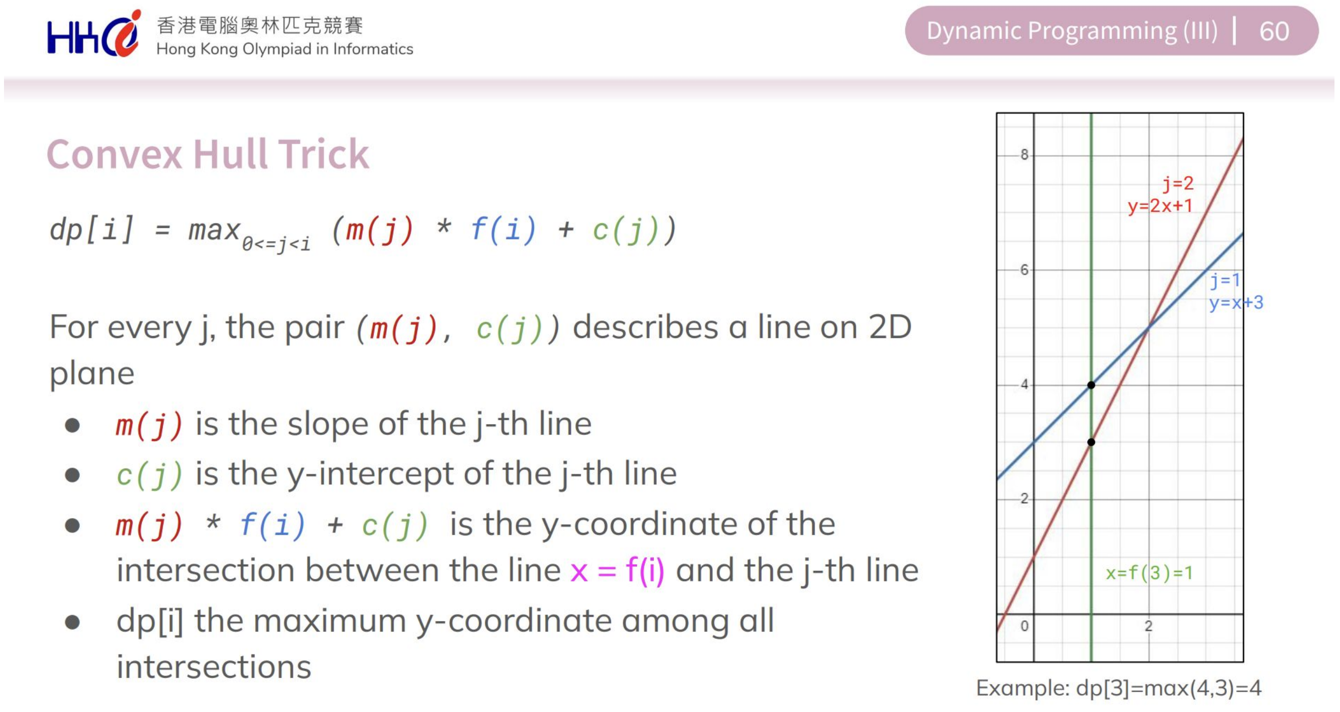
Look at the DP transition again:

$$DP[i] = i^2 + \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} j(-2i) + (DP[j] + j^2)$$

Can this DP be optimized?

You guessed it...

Review of Convex Hull Trick (CHT)

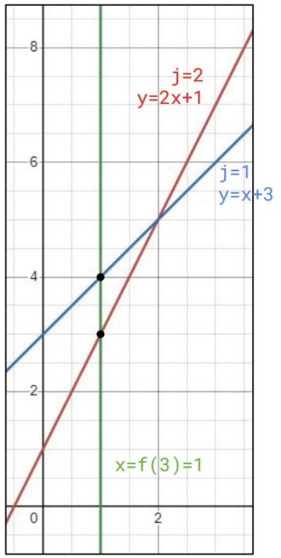


Convex Hull Trick

$$dp[i] = max_{0<=j<i}\ (m(j) * f(i) + c(j))$$

For every j, the pair $(m(j),\ c(j))$ describes a line on 2D plane

- $m(j)$ is the slope of the j-th line
- $c(j)$ is the y-intercept of the j-th line
- $m(j) * f(i) + c(j)$ is the y-coordinate of the intersection between the line x = f(i) and the j-th line
- dp[i] the maximum y-coordinate among all intersections



Example: dp[3]=max(4,3)=4

Implementations of CHT

We usually encounter two implementations of CHT:

- “Line container” implementation: actually stores a set of lines.
    - O(log N) insert
    - O(log N) evaluate at single point
- “Monotone deque” implementation: requires some monotonicity on the functions m and f.
    - O(1) insert
    - O(1) evaluate at single point
    - But doesn’t work in all cases

The Big Idea

Apply Convex Hull Trick to the following transition formula.

$$DP[i] = i^2 + \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} j(-2i) + (DP[j] + j^2)$$

Is that it?

The Big Idea

Apply Convex Hull Trick to the following transition formula.

$$DP[i] = i^2 + \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} j(-2i) + (DP[j] + j^2)$$

Is that it? **Probably not as simple as you think.**

Nuance:

- In normal CHT, the max is taken for $0 \le j < i$.
- In this task, the max is taken for $0 \le j < i$ **such that $\min(H[j+1], H[j+2], \ldots, H[i]) \ge i - j$**. Not all j are taken as candidates.
- We *might* need to implement some form of CHT with line deletions.

All Candidates Form a Range

Claim: All candidate j's for a single i form a range ending at i-1.

$$DP[i] = i^2 + \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} j(-2i) + (DP[j] + j^2)$$

Proof: Note that, as j decreases,

1) min(H[j+1], H[j+2], ..., H[i]) decreases
2) i - j increases

so if min(H[j+1], H[j+2], ..., H[i]) ≥ i - j does not hold for a certain j,
then min(H[j'+1], H[j'+2], ..., H[i]) ≥ i - j' must not hold **for all j' < j**.
Therefore all candidate j must form a range ending at i-1.

CHT + Range Query

Since for each i, we are basically querying a consecutive range of j's as candidates, we might think of doing some form of **CHT + Range Query**. This way we don't need to deal with line deletions.

Solution 1: $O(N \log^2 N)$

- Build a Segment Tree, where each node stores the candidates [L, R] using a Line Container.
- For a certain i, if we want to check all the candidates, say [lim_i, i - 1], then this is equivalent to a range query [lim_i, i - 1] on the Line Container Segment Tree.
- Visit O(log N) nodes, but each node requires a Line Container “evaluate at single point” query. So that’s $O(\log^2 N)$ per index $\Rightarrow$ $O(N \log^2 N)$ in total.

Expected Score: 72-84

Beyond O(N $\log^2$ N)

- In general, it seems unavoidable that for a range query, we will need O(log N) segment tree node visits.
- Let’s instead think of how to make every “evaluate at single point” query O(1) time, in other words:

  For some range of candidates [L, R], can we efficiently find the optimal candidate in O(1) time?

Monotonicity of Functions

$$DP[i] = i^2 + \max_{0 \le j < i,\ \min(H[j+1], H[j+2], \ldots, H[i]) \ge i-j} j(-2i) + (DP[j] + j^2)$$

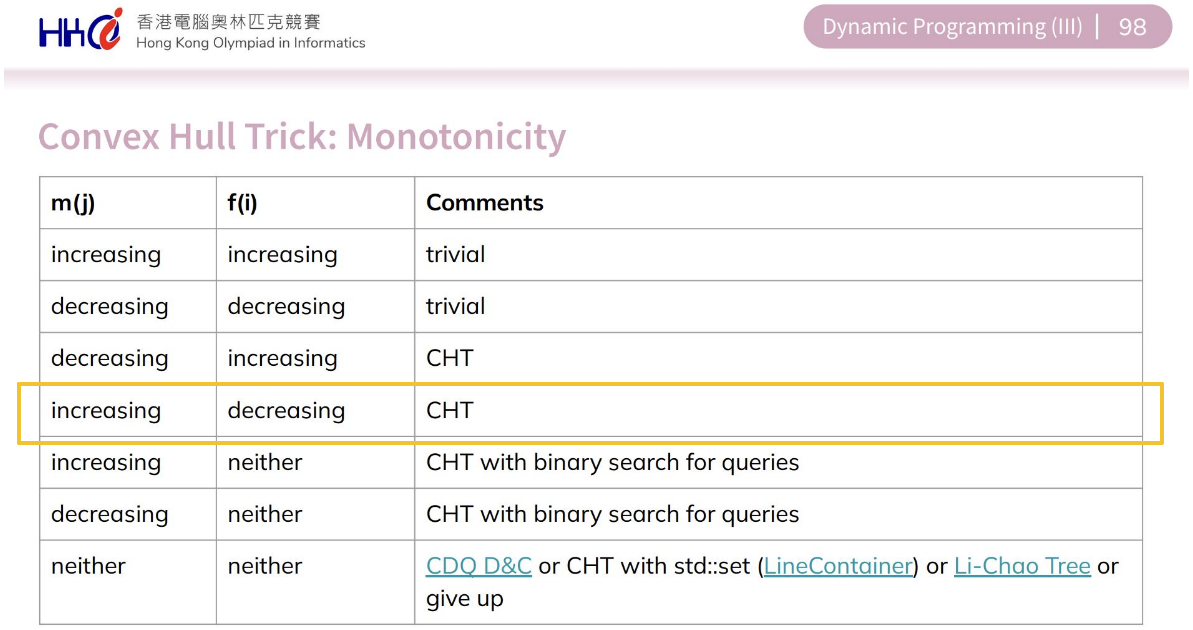
Observe that:

m(j) = j is monotonically increasing

f(i) = –2i is monotonically decreasing

Monotonicity of Functions

Taken from DP(III):



Convex Hull Trick: Monotonicity

m(j)	f(i)	Comments
increasing	increasing	trivial
decreasing	decreasing	trivial
decreasing	increasing	CHT
increasing	decreasing	CHT
increasing	neither	CHT with binary search for queries
decreasing	neither	CHT with binary search for queries
neither	neither	CDQ D&C or CHT with std::set (LineContainer) or Li-Chao Tree or give up

We can directly use the monotone queue implementation of CHT.

Solution 2: O(N log N)

- Build a Segment Tree, where each node stores the candidates [L, R] using a Monotone Deque.
- The implementation details are left as an exercise.

Expected Score: 92-100

Takeaways

- Most contests have some easy points up for grabs. Make sure you don’t lose those points!

- Being familiar with standard techniques and “templates” e.g. Line Container can give you a significant edge in competitions (especially NOI).

- Don’t just learn one way of solving a specific type of problems! Although the Line Container can deal with general CHT tasks well, there are cases when the extra O(log N) runtime might matter – learning the Monotone Deque approach is also important :)

Alternative Full Solution

- Another way to handle range query [lim_i, i - 1] **without** Segment Tree is by maintaining sqrt lines in blocks. Suppose B = sqrt(N).
- The blocks maintain lines of [0, B - 1], [B, 2B - 1], ..., [KB, (K+1)B - 1] as CHT with a monotonic queue each.

- For a query [L, R], loop through every blocks, if L <= Block_L && Block_R <= R:
  - Pop away no longer useful, lines. Eval the best line for x = i and update answer
- Remaining candidates in query that are not covered by blocks:
  - Iterate through all and update answer
- Query Time: O(sqrt(N)) blocks + O(sqrt(N)) individual candidates = O(sqrt(N))
- Total: O(N sqrt(N))

Alternative Full Solution

- The bottleneck of the sqrt solution is that maintaining sqrt(N)-length block would make sqrt(N) blocks.
- Can we make the block length longer?
- That would cost us more in traversing individual candidates.

- But what if we have dynamic block sizes?
- A property that we have not used yet: lim_i <= lim_(i + 1) <= lim_(i + 2) <= ...
- The query range left is also monotonic

Alternative Full Solution

- The idea is instead of maintain size-sqrt(N) block, we maintain a deque of blocks with size roughly following this trend:
  - `1 2 4 8 ... 2^k ... 8 4 2 1`
  - The number of blocks is always O(log(N)) level, which is better than O(sqrt(N))

- You can maintain the deque like this:
- **Add line:** while last two block have same size, merge them into a new block by adding all line from last to second last.
- **Query**: if the first block don’t contain the query_left, pop it away. If the query_left is within the first block but not the first line. Split the block into two halves. Repeat this split and pop until the first line of first block is query_l.

Alternative Full Solution

- **Time complexity:**
- Each element can only go through at most O(log N) merge, and O(log N) split, this part is O(N log N).
- Each query traverse the whole block deque once, which takes O(log N) time.
- Total: O(N log N)

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

M2544 - Strided Clique MSF

Isaac Wong {WongChun1234}
2025-06-29

Problem Background

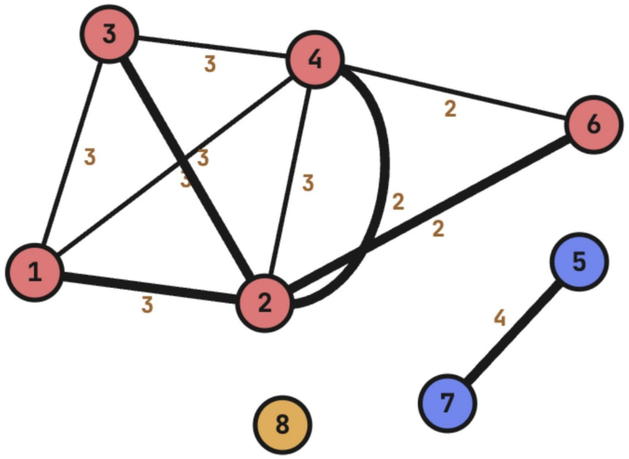
Problem Idea by `ethening`

Preparation by `WongChun1234` and `snowysecret`

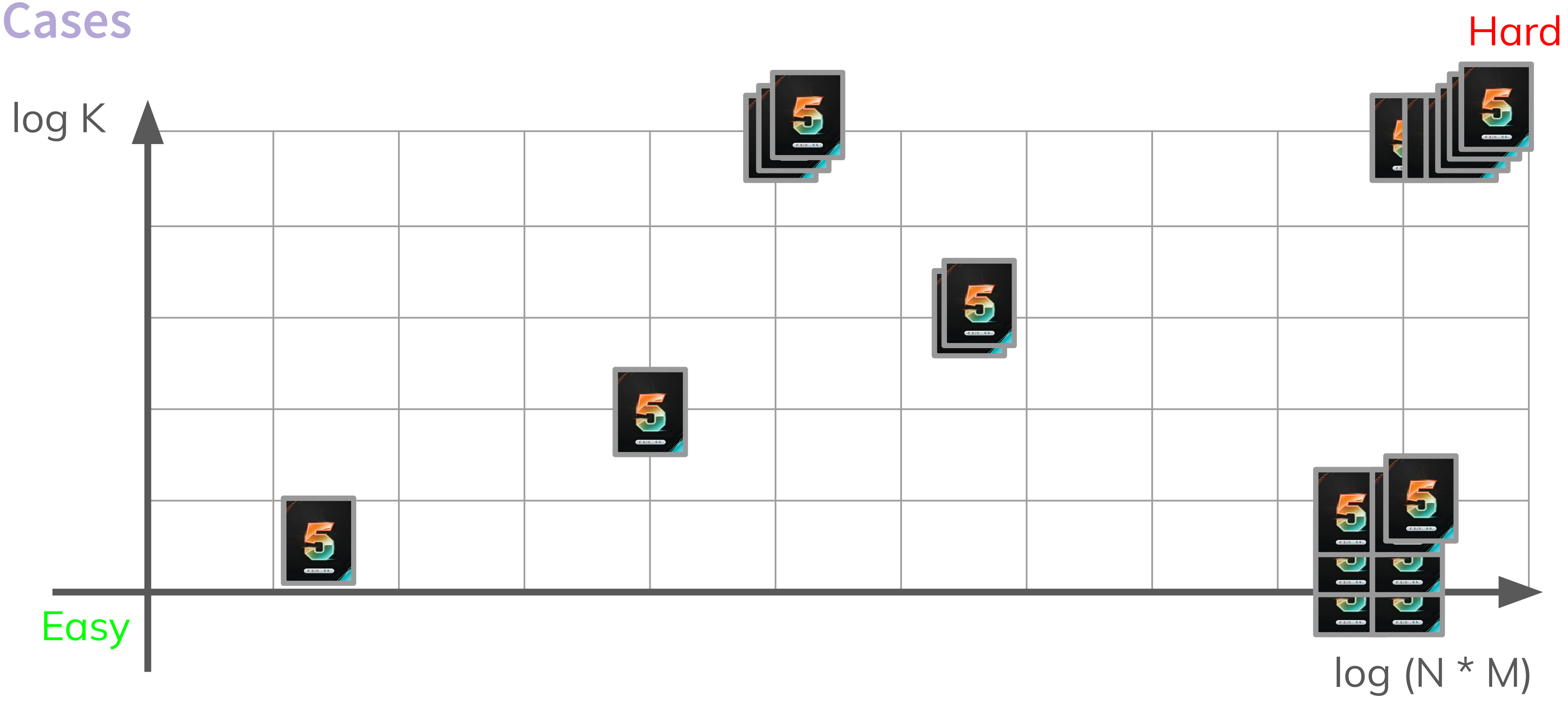
Presented by `WongChun1234`

The Problem

- Given a graph with **N** nodes and **0** edges
- Perform **M** operations to add edges
  - Set of node {$\mathbf{L_i}$, $\mathbf{L_i + K_i}$, $\mathbf{L_i + 2 * K_i}$, … , $\mathbf{R_i}$}
  - Add an edge between every pair of node in set
  - All edge has weight $\mathbf{W_i}$
- What is the weight sum of MSF of the graph?
  - MSF: MST for each connected component

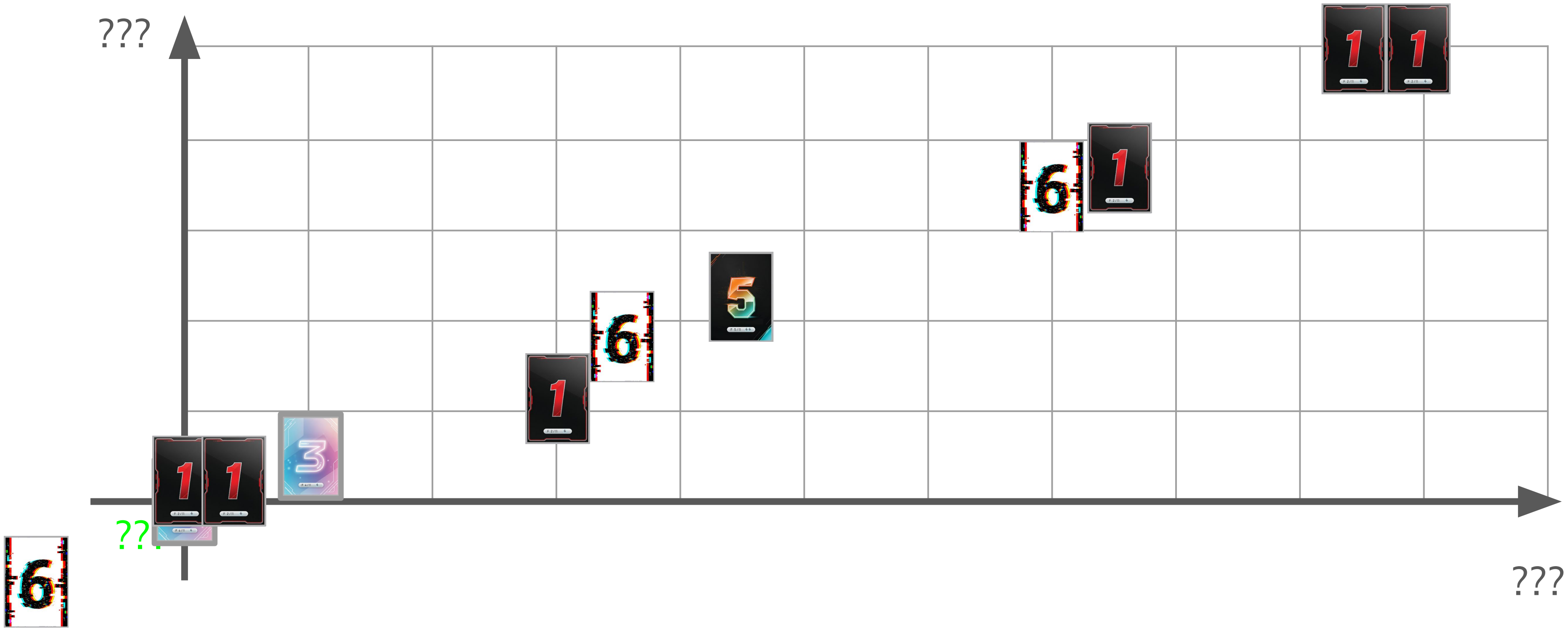


Cases



Statistics

???



Case 1~2 ($N, M, K \le 100$)

- A standard MSF problem
- You can just use your normal MST code
- Solution sketch:
    - Add every edge in every operation
    - Run kruskal’s / prim’s algorithm

Expected Score: 10

Case 3~4, 5~7 (N, M, K ≤ 2000), (M ≤ 10)

- We need to reduce the number of edges in the graph

Key Observation: We can only consider edges in one of the MST of the clique

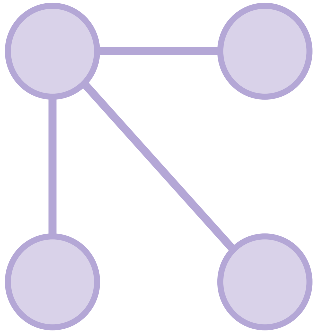
Proof by Induction: Each node in the clique must be able to reach any other node only using the MST

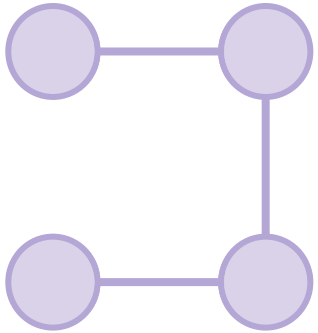
- We can directly run MST algorithm on the graph with O(**NM**) edges

Expected Score: 35

Case 8~9 (K = 1)

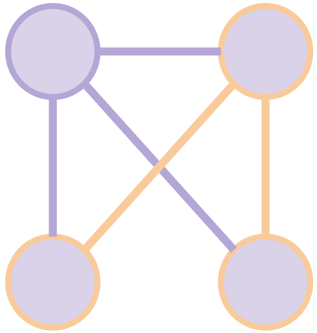
- Which MST of the clique should we choose?
- Two main candidates are star and chain
- Which one is better for this task?

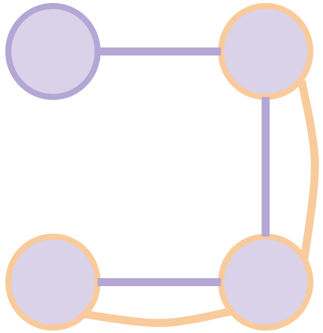




Case 8~9 (K = 1)

- Which MST of the clique should we choose?
- Two main candidates are star and chain
- Which one is better for this task?
- The chain MST always connects same pairs of nodes





Case 8~9 (K = 1)

- However, we still have too many edges
- Optimize the process of finding the MST

Kruskal's Algorithm:

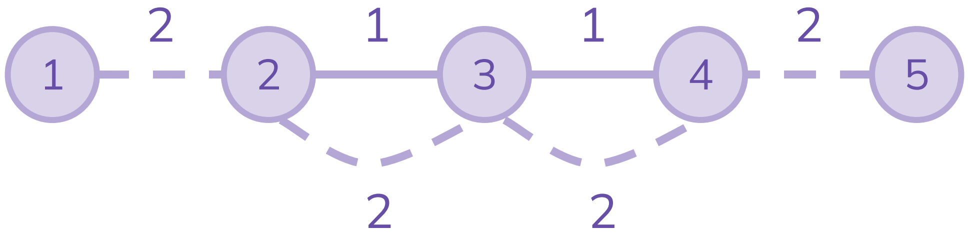
```
For all edges in increasing weight :
    Connect it if endpoints are not connected yet
```

- We can "skip" some edges if we know for sure they are not useful
  i.e. their endpoints are already connected
- We must use all other edges

Case 8~9 (K = 1)

Sample Input:
```
5 2
2 4 1 1
1 5 1 2
```

- We still consider operation **1** first
- Do we still need to consider all edges for operation **2**?
- If we know nodes **2** to **4** are already connected, we can skip (**2, 3**) and (**3, 4**)
- In fact, we (our programs) know this!



Case 8~9 (K = 1)

- Consider the **dsu[]** array and the actual root
- We merge **u** to **v** for every edge (**u, v**)

Sample Input:

```
5 2
2 4 1 1
1 5 1 2
```

root(i)	1	2	3	4	5
dsu[i]	1	2	3	4	5

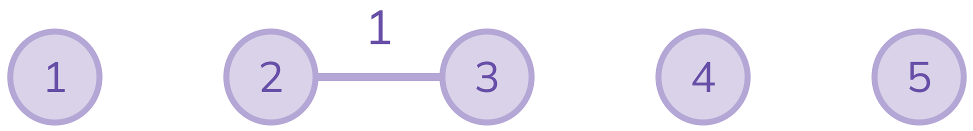
Case 8~9 (K = 1)

- Consider the **dsu[]** array and the actual root
- We merge **u** to **v** for every edge (**u, v**)

Sample Input:

```
5 2
2 4 1 1
1 5 1 2
```

root(i)	1	3	3	4	5
dsu[i]	1	3	3	4	5

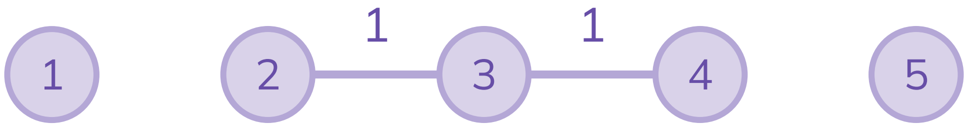


Case 8~9 (K = 1)

- Consider the **dsu[]** array and the actual root
- We merge **u** to **v** for every edge (**u, v**)

Sample Input:
5 2
2 4 1 1
1 5 1 2

root(i)	1	4	4	4	5
dsu[i]	1	3	4	4	5

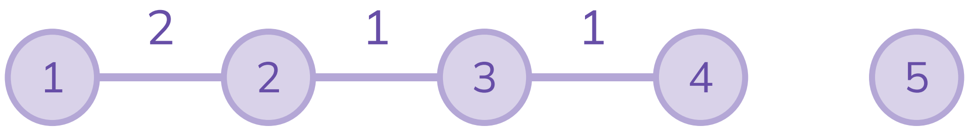


Case 8~9 (K = 1)

Sample Input:
```
5 2
2 4 1 1
1 5 1 2
```

- Consider the **dsu[]** array and the actual root
- We merge **u** to **v** for every edge (**u, v**)
- At this moment, we already know nodes **1** to **4** are connected

root(i)	4	4	4	4	5
dsu[i]	2	3	4	4	5



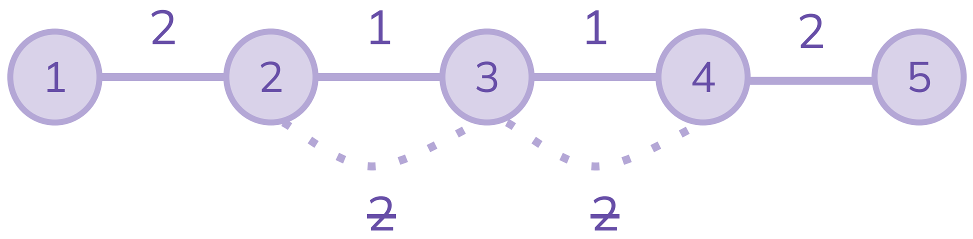
Case 8~9 (K = 1)

Sample Input:

```
5 2
2 4 1 1
1 5 1 2
```

- Consider the **dsu[]** array and the actual root
- We merge **u** to **v** for every edge (**u, v**)
- At this moment, we already know nodes **1** to **4** are connected
- We can directly process edge (**4, 5**) and onwards

root(i)	4	4	4	4	5
dsu[i]	2	3	4	4	5



Case 8~9 (K = 1)

- What is the time complexity of the algorithm after this optimization?
- We now guarantee that we will use every edge considered
- In total, only at most N - 1 edges will be used in the MSF
- The algorithm runs in O(N)

Original Solution	New Solution

`for (int i = l; i < r; i++){`
`  if (root(i) != root(i + 1)){`
`    union(i, i + 1);`
`    ans += w;`
`  }`
`}`
	`for (int i = root(l); i < r; i = root(i)){`
	`  union(i, i + 1);`
	`    ans += w;`
	`}`
	
	Expected Score: 45

Case 10~11 (K ≤ 2)

- Let’s consider a easier version **K = 2** first
- Can we use the same solution?
- Yes! We can consider odd-indexed elements and even-indexed elements separately.
- They both can be reduced to **K = 1** case separately.
- We don’t even need to use two **dsu** arrays as odd-indexed elements and even-indexed elements in the array does not affect each other.

Case 10~11 (K ≤ 2)

Sample Input:
5 2
2 4 2 1
1 5 1 2

- Now let's go back to K ≤ 2
- Can we still use the same solution?
- Kinda... This time we need to use two **dsu** arrays...

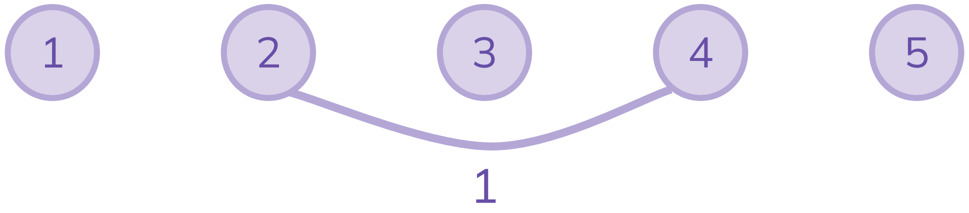
root(i)	1	2	3	4	5
dsu[i]	1	2	3	4	5

Case 10~11 (K ≤ 2)

- Now let's go back to K ≤ 2
- Can we still use the same solution?
- Kinda... This time we need to use two **dsu** arrays...

Sample Input:
5 2
2 4 2 1
1 5 1 2

root(i)	1	4	3	4	5
dsu[i]	1	4	3	4	5

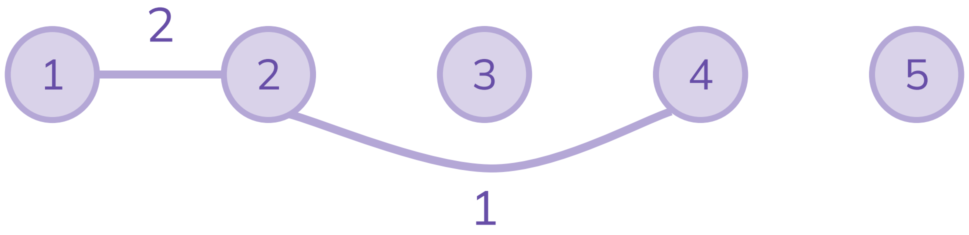


Case 10~11 (K ≤ 2)

Sample Input:
5 2
2 4 2 1
1 5 1 2

- Now let's go back to K ≤ 2
- Can we still use the same solution?
- Kinda... This time we need to use two **dsu** arrays...
- Even though **root(2)** is still **4**, the situation is different now

root(i)	4	4	3	4	5
dsu[i]	2	4	3	4	5

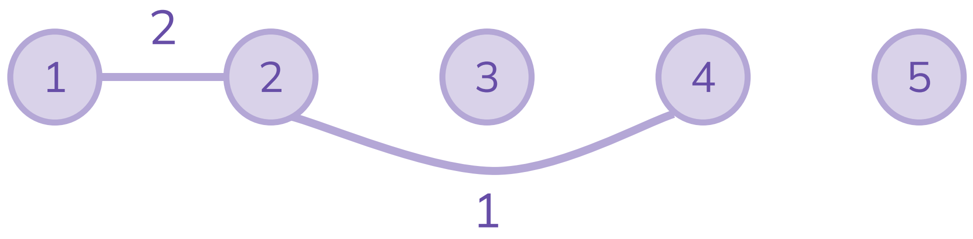


Case 10~11 (K ≤ 2)

- Use **root1** for **K = 1** and **root2** for **K = 2**
- It looks fine now, we can keep going

Sample Input:
5 2
2 4 2 1
1 5 1 2

root1(i)	2	2	3	4	5
root2(i)	1	4	3	4	5



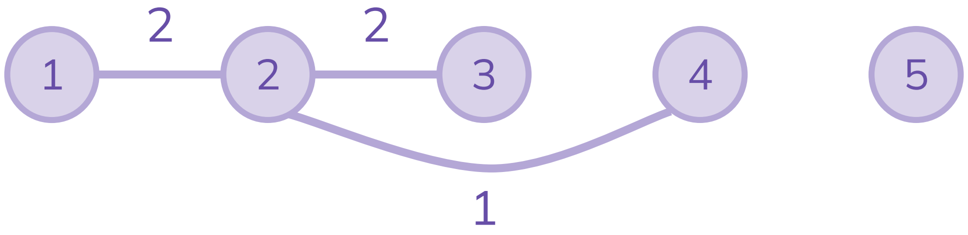
Case 10~11 (K ≤ 2)

- Use **root1** for **K = 1** and **root2** for **K = 2**
- It looks fine now, we can keep going

Sample Input:

```
5 2
2 4 2 1
1 5 1 2
```

root1(i)	3	3	3	4	5
root2(i)	1	4	3	4	5

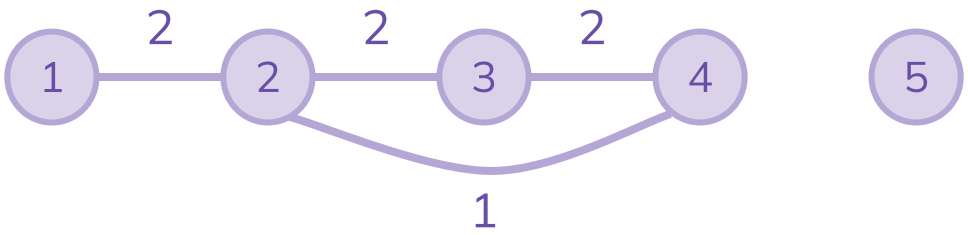


Case 10~11 (K ≤ 2)

- Use **root1** for **K = 1** and **root2** for **K = 2**
- It looks fine now, we can keep going
- Nevermind we are not fine

Sample Input:
```
5 2
2 4 2 1
1 5 1 2
```

root1(i)	4	4	4	4	5
root2(i)	1	4	3	4	5

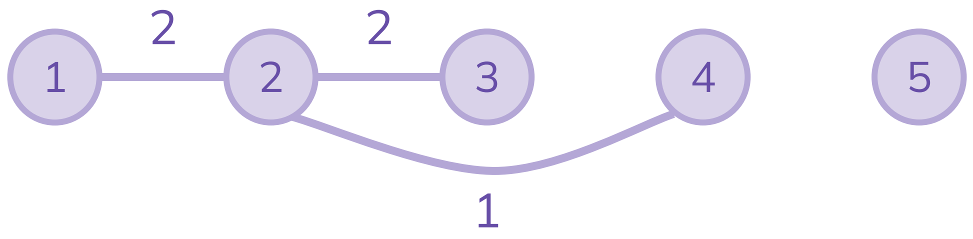


Case 10~11 (K ≤ 2)

Sample Input:
5 2
2 4 2 1
1 5 1 2

- We also need to keep track of the real root(i)
- Now we know we don't need edge **(3, 4)**

root(i)	4	4	4	4	5
root1(i)	3	3	3	4	5
root2(i)	1	4	3	4	5

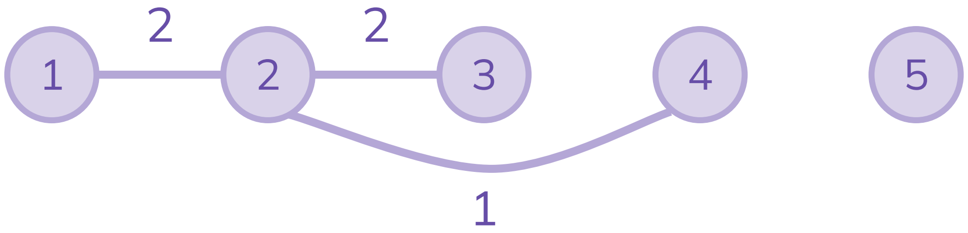


Case 10~11 (K ≤ 2)

Sample Input:
5 2
2 4 2 1
1 5 1 2

- We also need to keep track of the real root(i)
- Now we know we don't need edge **(3, 4)**
- However, we still need to update dsu1[] as nodes 1 to 4 are connected

root(i)	4	4	4	4	5
root1(i)	4	4	4	4	5
root2(i)	1	4	3	4	5



Case 10~11 (K ≤ 2)

- What is the time complexity of this solution?
- In every iteration we either merge **realdsu**, **dsu1** or **dsu2**.
- The total time complexity is O(**NK**)

Expected Score: 55

Case 12~14 (K ≤ 10)

- The solution is basically same
- However, you should write expandable code in order to pass this
- eg. use root(k, x) instead of root1(x) and root2(x)
- You should also use one dsu only for each k

Sample Implementation

```
for (int i = root(k, l); i < r; i = root(k, l)){
  if (realroot(i) != realroot(i + k)){
    realunion(i, i + k);
    ans += w;
  }
  union(k, i, i + k);
}
```

Expected Score: 70

Case 15~20 ($K \le 10^5$)

- Our solution now uses O(**NK**) time and memory
- For small **K**, our solution is good enough
- However, for large **K**, it costs us a lot of time and memory

- Notice that when K_i is large, we add very few edges to the graph
- We can set a parameter **B** and separate the operations into two category.
    - When $K_i \le B$, run case 10~14 solution
    - When $K_i > B$, add all N / K_i edges directly
- The total time complexity is O($NB + N^2/B$)
- Set B as $\sqrt{N}$, total time complexity becomes O($N\sqrt{N}$)
- In practice, $K_i \le B$ cases runs much slower, so you should set a smaller **B**

Case 15~20 ($K \leq 10^5$)

- If you used **std::set** / **std::priority_queue** / **segment tree** for your K = 1 solution and expand on it, you might get **TLE** for having a log factor
- Running the O(NM) solution on (some $_{\text{(weak)}}$) N, M $\leq 10^5$ cases is surprisingly fast

Expected Score: 70 - 100

Case 1~9, 11~20 (No additional Constraints)

- ~~Run O(NM) solution~~
- ~~Exits when all N - 1 edges has been found~~

Expected Score: 95