Problem Tutorial: "Points"

Note that saying $u_x + v_x \geq u_y + v_y$ is equivalent to saying $u_x - u_y \geq v_y - v_x$. This motivates tracking $u_x - u_y$ and $v_y - v_x$ for the elements in U and V respectively. For each element in U, call $u_x - u_y$ its ID, and for each element in V, call $v_y - v_x$ its ID. Create a segment tree where leaves are keyed on ID. Each node in the segment tree tracks five values, the minimum possible values of u_x, u_y, v_x, v_y, and the minimum possible value of $\max(u_x + v_x, u_y + v_y)$ over all points covered by the implied ID range. The first four values can be maintained directly, so it remains to maintain the fifth value.

Since $u_x + v_x \geq u_y + v_y$ is equivalent to $u_x - u_y \geq v_y - v_x$, for any internal node, the minimum possible value of $\max(u_x + v_x, u_y + v_y)$ is attained by either summing u_y from the left child and v_y from the right child, or by summing u_x from the right child and v_x from the left child. Therefore, updating the values for a given node can be done in constant time, so each update can be processed in logarithmic time.

Problem Tutorial: "Bingo"

We can see that $n^2 - n$ is an upper bound for k, because we need at least one unfilled cell in each row. Now the question is: is this upper bound tight? That is, can we fill exactly $n^2 - n$ cells? It turns out the answer is yes, except when $n = 2$. Here we present one of the ways to do that.

If we ignore the diagonal bingo lines for now, we can consider arranging the unfilled cells diagonally:

```
       #####.
####.  ####.#
###.#  ###.##
##.##  ##.###
#.###  #.####
.####  .#####
```

But if we consider the diagonal bingo lines again, this approach sometimes doesn't work - in particular, when n is even, the diagonal going from the top-left to bottom-right will always form a bingo line. To fix this problem, we can slightly modify this approach by swapping the unfilled cells on the first and last row:

```
       .#####
.####  ####.#
###.#  ###.##
##.##  ##.###
#.###  #.####
####.  #####.
```

Now this solution almost works, but still fails when $n = 2$. Indeed, there is no way to fill $n^2 - n$ cells. Fortunately $n = 2$ is the only exception for this approach, so we can just treat it as a separate case.

Problem Tutorial: "AND PLUS OR"

Consider each index $i \in [0, 2^N - 1]$ as a subset of $\{0, 1, \ldots, N-1\}$. We also denote a_i as a set function $a(i)$. Let $x = i \cap j$, $y = i - j$, $z = j - i$. Then we want to find three disjoint sets x, y, z such that $a(x \cup y) + a(x \cup z) < a(x) + a(x \cup y \cup z)$, or $a(x \cup y) - a(x) < a(x \cup y \cup z) - a(x \cup z)$.

For a fixed x, y, let $f(z) = a(x \cup y \cup z) - a(x \cup z)$. If such z exists, $f(\emptyset) < f(z)$.

Let $z_j = \{i_1, i_2, \ldots, i_j\}$ $(z_{|z|} = z)$. Then, there exists some $1 \leq j \leq |z|$ such that $f(z_{j-1}) < f(z_j)$. If we put $x' = x \cup z_{j-1}$, we have $a(x' \cup y) - a(x') < a(x' \cup y \cup \{i_j\}) - a(x' \cup \{i_j\})$.

In conclusion, there exists an answer of the form $(x, y, \{e\})$ where e is an element that does not belong to $x \cup y$. This gives an algorithm that runs in time $O(3^N \times N)$.

If you swap i, j from the above solution and proceed identically, you can see that there exists an answer of the form $(x, \{e\}, \{f\})$, where $e \neq f$ and $\{e, f\} \cap x = \emptyset$. The number of such tuples is at most $2^N \times N^2$, so we can enumerate all of them and obtain an $O(2^N N^2)$ solution.

Problem Tutorial: “Two Bullets”

(Draft)

For two buildings i, j $(1 \le i < j \le N)$ with $A_i > A_j$, we can see that the building j must be destroyed after the building i. From this, we can construct a DAG where there exists an edge $i \to j$ iff $i < j$, $A_i > A_j$. The operation can be considered as removing at most two nodes with indegree 0.

On a general transitive directed graph, this problem is solved by Coffman and Graham (Coffman-Graham Algorithm) in polynomial time. The algorithm consists of two steps: Constructing a topological ordering, and constructing an operation sequence from the topological ordering. When constructing a topological ordering, we take the vertices such that their *lexicographical order* is minimized.

We add the vertex repeatedly in the topological order, where the chosen vertex must have all incoming neighbors already in the topological order. If there are more than one such vertices, they are compared by the following rules: Let $N(v) = \{t_1, t_2, \ldots, t_k\}$ be the decreasingly-sorted sequence where each element denotes a time when its incoming neighbors were added. Then, the algorithm picks a vertex that minimizes $N(v)$ per its lexicographic order.

Given this topological ordering, we can construct an optimal solution by considering vertices in reverse topological order. At each step, consider all nodes with outdegree 0, and take two nodes that have the highest index in the topological order. If there are no such two nodes, you should take just one.

Implementing this procedure in the most naive way will yield a $O(N^3)$ complexity. To optimize this procedure, we need several lemmas and data structures. Let $level(v)$ be the length of the longest path from v to any vertex with outdegree 0. In a given graph, this value can be computed in $O(N \log N)$ time with a segment tree. Then, it can be shown that, if $level(i) < level(j)$, the topological ordering will always select j earlier than i.

As a result, the topological sort procedure can be reduced to an actual "sort"procedure, where you take all nodes with same level and sort them by their neighbors. To compare $N(v)$ and $N(w)$ efficiently, note that this is essentially equivalent to comparing the maximum between $N(v) - N(w)$ and $N(w) - N(v)$. If you plot the points in 2D coordinate as (i, A_i), then the area represented by $N(v) - N(w)$, $N(w) - N(v)$ is a rectangle. As a result, the comparison between two neighbor set is essentially a 2D range query which can be done with 2D segment trees. This solves the topological ordering in $O(N \log^3 N)$.

Given the topological ordering, you can maintain the set of 0-degree vertices in heap, where each vertex are ordered by their topological order. If a vertex is removed, then you can recover the newly created 0-degree vertices from the segment tree.

For all the omitted proofs, you can refer to the following papers:

- Coffman, Edward G., and Ronald L. Graham. "Optimal scheduling for two-processor systems."Acta informatica 1.3 (1972): 200-213.
- Sethi, Ravi. "Scheduling graphs on two processors."SIAM Journal on Computing 5.1 (1976): 73-82.

Remark. As Yuhao Du once noted (`https://codeforces.com/blog/entry/63630`), the minimum time is equal to N - (maximum matching in the complement of dependency graph). Since the complement of the permutation graph is a permutation graph, this problem is related to the two-processor scheduling on the permutation graph and the maximum matching on it. This enables a different kind of (and probably faster) approach. In the permutation graph, you can recover the solution from the maximum matching in $O(n \log n)$ time. There exists an algorithm that computes the maximum matching in permutation graphs (`https://link.springer.com/article/10.1007/BF01178659`) in $O(n \log \log n)$ time, you can yield a faster solution. In this perspective, the two-processor scheduling approach in this tutorial also finds a maximum matching, which is also used in another approach to compute the maximum matching in permutation graph. (`https://carleton.ca/scs/wp-content/uploads/TR-95-12.pdf`)

Problem Tutorial: "Yet Another Interval Graph Problem"

For convenience, we'll call a set of intervals good if all connected components have size at most K.

Take the union of intervals in each connected component. The intervals represented by each component are disjoint. From this, we will use the DP approach where we fix the union of intervals and fill each component.

Discretize the intervals to have coordinates up to $2N$, and define $f(x)$ to be the maximum cost of intervals such that the chosen intervals are good, considering only intervals whose rightmost endpoint is less than or equal to x. The answer is therefore $\sum_{i=1}^{N} w_i - f(2N)$.

If we define $g(a, b)$ to be the maximum cost of intervals to take such that, considering only intervals that lie entirely in $[a, b]$, the resulting set of intervals belongs to a single good connected component, then we have that $f(x) = \max_{0 \le a < x} f(a) + g(a+1, x)$. To compute $g(a, b)$, we can simply find the intervals that lie entirely within the interval $[a, b]$ and take the ones with top-k weight. This may result in not having the *single* connected component, but it doesn't matter since each connected component will remain good anyway. Since $g(a, b)$ can be computed naively in $\mathcal{O}(N^3 \log N)$ time, it remains to compute g efficiently.

If we sweep a from 0 to $x - 1$, with some careful bookkeeping we can compute $f(x)$ in $\mathcal{O}(x)$ time, which would give us an $\mathcal{O}(N^2)$ algorithm. We'll consider all intervals with left endpoint less than or equal to x active, and furthermore partition them into intervals which extend past x and intervals which do not. When we sweep a in increasing order, some intervals become inactive. We wish to maintain the sum of the K heaviest intervals that do not go past x, which can be done by sweeping those in decreasing order of weight and making sure we have included no more than K of them.

Problem Tutorial: "Lag"

(Draft)

Let's solve a variant of the problem where all windows have bottom left corner in the bottom left corner of the screen and all moves simply affect the top right corner. To count pixels in this case, we can process updates offline and maintain several segment trees along the axes that the corners can move in, where we count the number of points that appear at a given point along that axis.

The original problem can therefore be solved by using this as a subroutine, and doing inclusion-exclusion on each of the four corners of the original rectangles.

Problem Tutorial: "Endless Road"

Use coordinate compression. Let's call each length 1 interval after coordinate compression an *atomic interval*. In other words, coordinate compression partitions the road into at most $2N$ atomic intervals.

For each member, we maintain the total length of the unplanted intervals $remain[i]$, so that we can find the member to plant. When a member plants the flowers, we plant them for each atomic interval one by one. Planting for some atomic interval implies reducing the value $remain[i]$ for all members containing that interval, which can be naively processed in $O(N)$ time. This happens for at most $O(N)$ times, so we obtain an $O(N^2)$ algorithm.

Assume that $L_i \le L_{i+1}, R_i \le R_{i+1}$. If we remove an atomic interval, the members for which we should reduce the value $remain[i]$ form an interval over that sorted sequence. This interval can be found with binary search.

Thus, we want to find the minimum over $remain[i]$, find atomic intervals newly removed, and for each of them do a range decrement operation. The first and third operations can be done by a segment tree with lazy propagation in $O(\log N)$ time. The second operation can be done by maintaining a set of unplanted atomic intervals. This in total gives an $O(N \log N)$ algorithm. To avoid processing a member twice, set $remain[i]$ to a sufficiently large number after the turn. Keep in mind that ties should be broken by the original indices of the members.

For a full solution, we need a following key observation:

Lemma. For two different members i, j, if $L_i \leq L_j, R_j \leq R_i$, then member i always works earlier than member j.

Proof. Observe that $remain[i] \leq remain[j]$ always holds for such a pair of members. If $remain[i] < remain[j]$, the statement holds. If $remain[i] = remain[j]$, then $i < j$ by the non-decreasing length condition in the problem.

Let's call a member *candidate* if it does not contain any other members. In other words, it does not certainly work later than other members. To compute the set of candidates, we sort all members in the increasing order of L, where ties are broken by the decreasing order of R. Then we scan in the decreasing order of sorted indices ($N - 1$ to 0), adding a member if its R_i is smaller than all other scanned intervals. In other words, a member in the i-th sorted order is a candidate if the suffix minimum of index i is different from that of $i + 1$. Be careful that the index here differs from the index given in the input.

Given that we can maintain the candidates (in `std::set`) we can solve the problem. For each member not in the candidate, set $remain[i]$ to a sufficiently large number. Then the intervals could be found with an `lower_bound` operation in the set, and all other things can be done accordingly.

However, the set of candidates changes after we process each member. Specifically, we delete the candidate after the turn, and the removal of the candidate creates other possible candidates. To solve this, it's helpful to get back to the naive algorithm of finding candidates: It computes the suffix minimum and finds where it changes. As the member is removed (or, the member's value becomes infinity, in terms of suffix minimum), we have to reconstruct the position where it changes.

Let's store the list of endpoints in the range minimum segment tree. Then we repeatedly find the minimum element m (rightmost in case of a tie), add it to the candidate, and start ignoring the position m and left of it. This algorithm finds the candidate in the same way the above algorithm does. The only difference is that this runs in the backward direction. Now, if we remove candidate i from the set, we find the candidate j that is adjacent to the left of i from `std::set`. Now, we start in a state where j is the latest candidate found and repeat the algorithm. We finish the algorithm when we find the candidate already in the set, or we just run out of new candidates. This algorithm runs in $\log N$ time for each found candidate, and we don't find the same candidate twice, so the total complexity is $O(N \log N)$ for candidate reconstruction. If we finish processing each member, we have to set its value infinity in the segment tree.

When activating a new member, we don't know its actual $remain[i]$ value because we did not maintain it. We keep a segment tree that maintains the sum of the unplanted intervals. When we remove an atomic interval, we modify the set and the segment tree simultaneously. Then, in time of activation, we can compute the $remain[i]$ value for new candidates.

Summing it down:

- We keep a `std::set` for maintaining the candidates.
- We keep a minimum segment tree with lazy propagation to maintain $remain[i]$ array for candidates.
- We keep a `std::set` and sum segment tree (Fenwick tree) to maintain the unplanted atomic intervals.
- We keep a minimum segment tree to reconstruct the candidates.

The total time complexity is $O(N \log N)$.

Problem Tutorial: "Diameter Pair Sum"

(Draft)

To solve this problem in linear time per query, find the center of the queried component by computing a diameter and locating its ancestor. For convenience, assume that the center is a vertex of the tree. Root the tree from the center, and compute the following value for each vertex to its subtree:

- The farthest distance from the vertex to its descendants.
- The number of such descendants.
- For a pair of vertex which is farthest from the root, the sum of distance from root to their LCA.

Those values can be computed for each subtree for linear time and can be used to compute the desired answer.

To support the link-cut operation, we can adopt similar dynamic programming in the Top Tree. In the rake node, we simply store the pointwise sum of the DP. In the compress node, we store the DP value by considering the case where the leftmost or rightmost node becomes the root. If we store one vertex index that gives the farthest distance, we can compute the diameter and the center of the component in $O(\log n)$ time, where we can reroot the tree and compute the answer for the query in $O(\log n)$ time.

As a result, the problem can be solved in $O(n + q \log n)$ time.

Problem Tutorial: "Fake Plastic Trees 2"

We use dynamic programming on the tree, where we solve a problem for all rooted subtrees and recursively merge the solutions. Root the tree at vertex 1. Let T_v be a subtree rooted at vertex v.

Let $DP[v][k][x]$ be a boolean value which is true if there are k **closed** subtrees and one **extendable** subtree that has the weight sum of x in T_v. An extendable subtree is either empty, or contains a vertex v. It is *extendable* in a sense that it may contain other vertices and increase its size. If an extendable subtree is empty, then all subtrees under v are closed, including the one that contains the vertex v.

Then, we have the following recursive rule:

- **Base case:** For a leaf v, you have two cases: add v to an extendable, or closed subtree. If you add it to a closed subtree, then $L \le A_v \le R$ should hold.
- **Merge:** For two children w_1, w_2 of v, we will shrink them to obtain one child w equivalent to both children. If $DP[w_1][k_1][x_1]$ and $DP[w_2][k_2][x_2]$ are both true, then we can consider $DP[w][k_1 + k_2][x_1 + x_2]$ to be true.
- **Extend:** For a vertex v with a single child w, we will add the vertex v as a parent. If $DP[w][k][x]$ is true, then $DP[v][k][x + A_v]$ is true. If $L \le x + A_v \le R$, then we may want to close the subtree rooted in v, which makes $DP[v][k + 1][0]$ true.

Naively implementing this yields $O(NK^2R^2)$ time complexity, where the bottleneck comes from Merge rule. However, note that $k_1 \le |T_{w_1}|$ and $k_2 \le |T_{w_2}|$, which means the iteration count can be bounded as $c \times R^2 \times \sum_{v \in T}(\min(K, |T_{w_1}|) \times \min(K, |T_{w_2}|))$ for some constant c. Here, the following well-known lemma holds.

Lemma 1. Given a binary tree T, $\sum_{v \in T} \min(K, |T_{w_1}|) \times \min(K, |T_{w_2}|) = O(|T|K)$.

Proof. Use double counting. $\min(K, |T_{w_1}|)$ is the number of vertices in the left subtree with K highest DFS preorder numbers, $\min(K, |T_{w_2}|)$ is the number of vertices in the right subtree with K lowest DFS preorder numbers. The number of pairs of such vertices is at most the number of vertices with DFS preorder number difference at most $2K$, which is $O(|T|K)$.

Thus the dynamic programming runs in $O(NKR^2)$ time.

Let's optimize this dynamic programming solution. It is convenient to consider the DP value as a set of weights. Let $S(v, k) = \{x | DP[v][k][x]\}$. For two set A, B, define the *sumset* $A + B = \{a + b | a \in A, b \in B\}$. With this notation, we can revisit the recursive rule in much cleaner way.

- **Base case:** $S(v, 0) = \{A_v\}$ (if $A_v \le R$), $S(v, 1) = \{0\}$ (if $A_v \in [L, R]$).
- **Merge:** $S(w, k) = \bigcup_{0 \le i \le k} S(w_1, i) + S(w_2, k - i)$.

- **Extend:** $S'(v,k) = S(w,k) + \{A_v\}$. $S(v,k) = S'(v,k)$. If $[L,R] \cap S'(v,k) \neq \emptyset$, then $S(v,k+1) := S(v,k+1) \cup \{0\}$.

And we want to determine if $0 \in S(1, K+1)$. To optimize this lets proceed to another lemma.

Lemma 2. $\max S(v,k) - \min S(v,k) \leq k(R-L)$.

Proof. The sum of weights in closed subtrees is in the range $[kL, kR]$, and the sum of weights in T_v is fixed.

Note that we are only interested in if there is an element in some interval of length $R-L$. ($0 \in S$ is equivalent to $[-(R-L), 0] \cap S \neq \emptyset$).

Consider the following algorithm $reduce(S)$ for some set S. Let $a, b, c \in S$ and $a < b < c, c - a \leq R - L$. Then, we can remove the middle element b, while not changing any interval query result of length $R-L$. An easy way to implement this algorithm is to place buckets of length $R-L+1$ and take only the minimum and maximum elements over the buckets. For any set $S(v,k)$ the algorithm can be implemented in time $O(|S(v,k)| + k)$ and can guarantee $|reduce(S(v,k))| \leq 2k$.

To obtain a polynomial-time algorithm, we want to apply $S(v,k) := reduce(S(v,k))$ after each Merge and Extend case. We perform the sumset operation for $O(NK)$ times (by Lemma 1), and each operation takes $O(K^2)$ time, thus the final time complexity is $O(NK^3)$ and the problem is solved. However, to prove its correctness, we have to show that the reduce operation preserves the query result even after applying the *sumset* operation. This can be proved in the following step.

Definition. For $b \in \mathbb{N}$ and $A \subseteq \mathbb{N}$, define

- $apx^-(b, A) := \max\{a \in A | a \leq b\}$
- $apx^+(b, A) := \min\{a \in A | a \geq b\}$

We say A Δ-approximates B if $A \subseteq B$ and for every $b \in B$, $apx^+(b,A) - apx^-(b,A) \leq \Delta$.

Lemma 3. If A is a Δ-approximation of B, then $[x, x+\Delta] \cap A = \emptyset$ iff $[x, x+\Delta] \cap B = \emptyset$.

Proof. $\leftarrow$ is trivial. Suppose $[x, x+\Delta] \cap A = \emptyset$ and $y \in [x, x+\Delta] \cap B$. Then $y \in B$ and $apx^+(y,A) - apx^-(y,A) > \Delta$.

Lemma 4. If A_1 Δ-approximates B_1 and A_2 Δ-approximates B_2, then $A_1 + A_2$ Δ-approximates $B_1 + B_2$.

Proof. Consider all $x + y \in B_1 + B_2$. If $x + y \in A_1 + A_2$ or $x \in A_1$ or $y \in A_2$, then the statement holds.

Otherwise, Let $x_1 = apx^-(x, A_1)$, $x_2 = apx^+(x, A_1)$, $y_1 = apx^-(y, A_2)$, $y_2 = apx^+(y, A_2)$, and WLOG $x_2 - x_1 \leq y_2 - y_1$.

Let Z be the nondecreasing sequence $\{x_1 + y_1, x_2 + y_1, x_1 + y_2, x_2 + y_2\}$. Observe that $Z \subseteq A_1 + A_2$, $x_2 - x_1 \leq \Delta, y_2 - y_1 \leq \Delta, (y_2 - y_1) - (x_2 - x_1) \leq \Delta$. We can see any element $b \in [x_1 + y_1, x_2 + y_2]$ have $apx^+(b, A_1 + A_2) - apx^-(b, A_1 + A_2) \leq \Delta$, which is the case for $x + y$.

Lemma 5. $reduce(S)$ is a $(R-L)$-approximation of S.

Proof. Immediate from the algorithm.

Problem Tutorial: “Curly Racetrack”

For convenience, we’ll call a grid valid if the admin can fill in the rest of the racetrack.

Let’s first start by taking the role of the admin - how can we verify that a grid is valid? There are some states that are obviously invalid - for example, if two adjacent cells have incompatible roads. It turns out that this is a necessary and sufficient condition.

We need to make a few observations to prove this. Two horizontally adjacent cells must have curly tiles with opposite horizontal orientations (either they point into each other or they point away from each other). We can make a similar realization for vertically adjacent cells. If we color the cells according to their horizontal and vertical orientations, we can see that if two adjacent cells both have curly tiles in them, they must have different colors.

Because of this alternating pattern, we are now motivated to flip every other square's color. This now means that a segment of curly tiles is monochromatic. Note that the admin has the power to select other tiles, which effectively flips the color of the cell within the row or column. Therefore, as long as two nonempty adjacent cells are not different colors, the admin can always insert tiles in the given locations to respect the desired coloring. This completes the given proof.

The desired condition is equivalent to minimizing the number of cells that do not contain curly tiles. Perform the reduction above and color the grid along both rows and columns, validating that the admin is able to insert tiles to fix cells which are tentatively colored differently. Assuming it is possible for the admin to fill in the grid as is to make it valid, we have maximal horizontal and maximal vertical segments where we must have a non-curly tile because the cells on both ends are different colors. These segments form a bipartite graph, the maximum matching of which tells us which cells should not contain curly tiles. All other cells can contain curly tiles, per above.