Problem Tutorial: "Salty Fish"

Assume we have stolen all the apples, now we need to give up some apples and do some hacks to make sure we can steel all the rest apples. Thus we need to minimize the sum of number of apples we give up and the amount of dollars we need to pay for hacks.

Let's build a directed graph with $n+m+2$ vertices (n nodes, m cameras, the source S and the sink T):

- $S \to$ each camera, the weight of this edge is the cost we need to pay for hack this camera.
- Each node $\to T$, the weight of this edge is the number of apples on this node.
- Each camera $\to$ each node covered by this camera, the weight of this edge is $+\infty$.

The answer of the problem is $\sum a_i -$ the minimum cut of the graph. According to the max-flow min-cut theorem, we only need to compute the max-flow instead.

Let's denote $v[i][j]$ as how much flow we can still offer from all the nodes in vertex i's subtree whose distance to i is j. For each camera whose top vertex is i, in order to achieve the max-flow, we need to receive the flow from farther nodes, so just greedy receive the flow from larger j to smaller j.

Let's store positive $v[i][j]$ using `std::map`, then we can compute the max-flow in $O(m \log n)$ time. But how can we get $v[i][]$? We can just do the calculation from bottom to top, and merge all the $v[][]$ of i's children to get $v[i][]$.

Use long path decomposition to merge $v[]$, overall time complexity is $O((n+m)\log n)$.

Problem Tutorial: "Nonsense Time"

Let's compute each answer from n downto 1. Now the problem is: we will **erase** each element one by one, and compute the length of LIS.

Let's pick an LIS L, in case of there are multiple LISs, choose one among them arbitrarily. If we erase an element not in L, the answer is not changed, otherwise let's calculate the new answer and pick an LIS again.

Thanks to the randomness, the length of LIS is expected as $O(\sqrt{n})$, and the overall time complexity is expected as $O(n\sqrt{n}\log n)$.

Problem Tutorial: "Milk Candy"

Let's build an undirected graph with vertices 0 to n, where vertex i denotes $s_i = x_1 + x_2 + \cdots + x_i$. If we know the value of $x_l + x_{l+1} + \cdots + x_r$, we can know the value of $s_r - s_{l-1}$, so let's add an edge between vertex $l-1$ and vertex r. If the graph is connected, we can know all the values of $s_1, s_2, \ldots, s_n$ according to $s_0 = 0$, and we can know $x_i = s_i - s_{i-1}$. So we need to buy exactly k_i edges from the i-th NPC to make the graph connected and minimize the total cost.

Let's buy all the edges, then we need to delete **no more than** $c_i - k_i$ edges from the i-th NPC, and delete $\sum(c_i - k_i)$ edges in total. Now choosing the delete edges optimally can be solved using the matroid intersection algorithm.

Problem Tutorial: "Radar Scanner"

The intersection of several rectangles is still a rectangle. Let's denote $f_{i,j}$ as the number of rectangles covered (i,j), then $ans = \sum C(f_{i,j}, 3)$.

The answer may not be correct, that is because the intersection may cover $(i, j-1)$ or $(i-1, j)$. So we need to subtract the number of ways that both cover (i,j) and $(i,j-1)$ from the answer, and subtract the number of ways that both cover (i,j) and $(i-1,j)$ from the answer. But the number of ways both cover (i,j) and $(i-1,j-1)$ will be subtracted twice, so we need to add it back.

Overall time complexity is $O(n + m^2)$, where m is the range of coordinate.

Problem Tutorial: "Snowy Smile"

Let's fix the top of the rectangle, and consider each pirate chests from top to bottom. Now it becomes a dynamic 1D version, which can be speeded up using segment tree.

Overall time complexity is $O(n^2 \log n)$.

Problem Tutorial: "Faraway"

Let's first get rid of the absolute value in $|x_i - x_e| + |y_i - y_e|$, each point (x_i, y_i) will split the whole plane into 4 areas, so there will be $O(n^2)$ areas.

In each area, if (x, y) is valid, $(x + 60a, y + 60b)$ is also valid. So let's just enumerate the value of x mod 60 and y mod 60, and calculate the number of such points in each area.

Overall time complexity is $O(60^2 n^3)$.

Problem Tutorial: "Support or Not"

Let's first find the k-th smallest distance. Binary search for the answer, and count how many pairs of spheres with distance no more than mid there are. We need to find the smallest mid such that there are at least k touched pairs.

Let's increase each sphere's radius by $\frac{mid}{2}$, what we need to do is to count how many pairs of spheres share common points.

Let's consider spheres from larger ones to smaller ones. Assume the largest one's radius is R, let's build a cube system in the 3D-space with length $2R$, i.e. put a sphere (x, y, z) into the cube $(\lfloor \frac{x}{2R} \rfloor, \lfloor \frac{y}{2R} \rfloor, \lfloor \frac{z}{2R} \rfloor)$. Obviously, if two spheres share common points, they must be in the same cube or in two adjacent cubes. So let's enumerate spheres from larger ones to smaller ones, and check it with all the spheres in the nearby cubes.

Note that if all the radius are the same, when there are $O(\sqrt{k})$ spheres inside a cube, we can always find k touched pairs. So let's break the algorithm when we find k touched pairs.

But when the radius are not all the same, the above proposition is false. So when the current spheres' radius is less than $\frac{R}{2}$, we need to rebuild such cube system, but we will rebuild for at most $O(\log r)$ times.

When we find the k-th smallest distance ans, let's take $mid = k - 1$, and run the algorithm described above, we will find the whole answer. Assume we find m pairs, we can conclude that the lost $k - m$ answers are all ans.

To speed up the algorithm, we can use hash table to find cubes, the overall time complexity is $O(n \log^2 r + n\sqrt{k} \log r)$.

Problem Tutorial: "TDL"

Assume $f(n, m) - n = t$, then we have $n = t \oplus k$. t won't be very large, so enumerate all the values of t and check it is OK.

Problem Tutorial: "Three Investigators"

A similar well-known problem to this problem is: select no more than k non-decreasing subsequences and maxmize the total **length** of them. The solution to that problem is maintaining the first k rows of Young Tableau, and the answer is equal to the total length of the first k rows of Young Tableau.

In this problem, we need to maxmize the sum instead of length. We can split x into x copies of x to reduce this problem into that well-known one. For example, "`4,1,2`"→"`4,4,4,4,1,2,2`".

The only thing we need to deal with is that x may be very large, we can't insert value x into the Young Tableau for x times. We can store each row of Young Tableau using `std::map`, which means compressing the same value into one element in `std::map`.

Since the number of insertions in the k-th row is at most twice as many as the number of insertions in the $(k-1)$-th row, so the overall time complexity is $O(2^k n \log n)$, where $k = 5$.

Problem Tutorial: "Road Manager"

Let's denote pre_i as the MST of the first i columns, and denote suf_i as the MST of the last $m - i + 1$ columns. To answer a query, we need to merge pre_l and suf_r, and we also need to merge pre_{i-1} with the input to get pre_i. So if we can merge two parts of the graph efficiently, we can solve this problem.

Let's denote $T(l, r)$ as the MST of the columns $l, l + 1, \ldots, r$. How can we merge $T(l, k), T(k, r)$ into $T(l, r)$? Let's just merge this two trees, and then run a Kruskal algorithm to find the new MST. The only thing bad is that the number of vertices may be doubled.

Let's mark vertices in column l or r as important vertices, there will be at most $2n$ important vertices. You may notice that only the important vertices are useful, since there are no more edges linked to unimportant vertices if we try to merge more parts from the graph. Let's compress those unimportant vertices whose degree is no more than 2, and delete all the subtrees with no important vertices, we will finally get a tree with at most $4n{=}O(n)$ vertices. Then the brute force merge method is fast enough. When we are deleting a subtree or compressing a chain, we need to add the weights of edges inside it into the answer. And when we are compressing a chain, we need to keep the maximum-weight edge, that is because the Kruskal algorithm will always cut the maximum-weight edge on a cycle.

Overall time complexity is $O((m + q)n \log n)$.

Problem Tutorial: "Monster Hunter"

When the tree looks like a star, we can figure out a best ordering $p_1, p_2, \ldots, p_{n-1}$ to beat each monster by greedy sorting. There are two cases:

- p_1's father is the root of the tree, we can always beat this monster. So we can move this monster to the root, and merge this node with the root.
- p_1's father is not the root of the tree, we can always beat this monster when we have beaten its father. So we can merge this node and its father.

No matter which case, we can always reduce the problem of size n into a problem with size $n - 1$. To speed up the algorithm, we need a heap to find p_1 and a disjoint set union to merge nodes.

Overall time complexity is $O(n \log n)$.

Problem Tutorial: "Game Prediction"

We can calculate the difference D between S and E instead of the real value of them. When we know D, we can know S and E according to $S + E = a_l + a_{l+1} + \cdots + a_r$ and $S - E = D$.

When there are three adjacent values A, B, C satisfying $A \leq B$ and $B \geq C$, if one player takes A or C, the other player will always take B, then the previous player is forced to take C or A to reduce his loss. Thus we can merge these three adjacent values A, B, C into a single one $A - B + C$.

When there is only one query, we can consider each value from left to right, and keep a stack to store them. When we are facing a new value, push it into the stack, and merge the top three values on the stack as many times as we can. We will finally get a stack with values first decreasing then increasing, which looks like a valley. When the game is hold on such a "valley", the greedy strategy works, the player can always choose the bigger value among the two ends.

Thanks to the randomness, there won't be many elements in the stack, so we can just store all the stacks in each segment tree's node to answer queries efficiently.