icpc university commons
icpc.foundation
icpc asia
icpc nerc

HUAWEI

# Problem Analysis (Day 1)

# Problem A

- If at any time, the following situation happened:
- `A......B(exactly 2n cols between A&B, A goes first)`
- `........`
- Then A can always reach the left n cols, while B reaches the right n cols.
- Meanwhile, if at any time, the following situation happened:
- `A.......B(exactly 2n+1 cols between A&B, A goes first)`
- `.........`
- Then A can always reach the left n+1 cols, while B reaches the right n cols

# Problem A

- So we can make the following discussion:
- `If n == 1 return a[1][1]`
- `Else`
- `    m = n/2`
- `    X = the weight of grids in the first m columns`
- `    Y = the weight of grids in the first m+1 columns`
- `    Z = the weight of grids in the first row.`
- `    return min(Y,max(X,Z))`
- Note that `min(Y,max(X,Z))=max(X,min(Y,Z))`, so both are OK.

# Problem B

- Since the painting process is reversible, we may view the input as original state, and the goal is to paint all roads white
- To simplify the problem, we assume the tree is rooted at vertex 1, and add vertex 0 to be the father of 1

# Problem B

- The first key observation is that, each walk from $u$ to $v$ can be divided into two walks: $u \to 0$ and $0 \to v$.
- The second key observation is that, let $d_u$ denote the parity of number of black edges around $u$, then the tree is painted all white iff $\forall u \in \{1, \dots, n\}, d_u = 0$
- The walk $u \to 0$ only flips $d_u$ and $d_0$

# Problem B

- Thus we only need to choose some destinations on the tree, while satisfying the parity restriction for each vertex, and pair up the starts and destinations so that the total cost is minimized
- It can be solved using DP: Let $f_{ui}$ denote the minimum total cost in subtree $u$, given that we set exactly $i$ destinations in subtree $u$

# Problem B

- Let $sz_u$ be the number of starting points in subtree $u$, and $tz_u$ be the number of vertices in subtree $u$
- By a simple greedy argument, one can show that in the optimal solution, there are no more than $sz_u + tz_u$ destinations in subtree $u$
- Using this property, we can design a DP that runs in time $O(n^2)$
- The proof is similar to the knapsack-on-tree problem

# Problem C

- while True:
  - p = longest path in $G'$
  - query every edge of p in order until we found an edge in $E' \backslash E$
  - if no such edge is found:
    - return len(p)
  - else:
    - remove that edge from E′
- The algorithm lasts for at most $|E'| - |E| + 1$ rounds
  - Each round we remove an edge from $E' \backslash \mathrm{E}$
- For each round, at most $l(G) + 1$ edges are queried
  - Otherwise we found a path longer than $l(G)$, which is impossible

# Problem D

- First, sort n numbers. We find that the relative size of the final n numbers does not change in the end.
- We consider dynamic programming.
- Define $f_{i,j}$ as the minimum cost of the first i numbers fill all the numbers from 1 to j.
- We have $f_{i,j}$ =min($f_{i-1,j}$ , $f_{i-1,j-1}$ )+abs($a_i$ − j )
- The time of this solution is O(n^2)

# Problem D

- Consider the relationship between $a_{i+1}$ and $f_{i,j}$
- If $a_{i+1} >$ j , then it is better for $f_{i,j}$ to transfer to $f_{i+1,j+1}$.
- If $a_{i+1} = j$ ,then both types of transfer are possible.
- If $a_{i+1} < j$ , then it is better for $f_{i,j}$ to transfer to $f_{i+1,j+1}$
- (It is noted that if the transfer is to $f_{i,j}$, then the transfer $f_{i,j-1} \rightarrow f_{i+1,j}$ is better than $f_{i,j} \rightarrow f_{i+1,j}$. This is because $a_i < j$, which implies $f_{i,j-1} < f_{i,j}$.)

# Problem D

- Note that only when $a_{i+1} = j$, there are two kinds of transitions
- Maintain the two stacks of $f_{i,j}$on the left and right of $a_{i+1}$ .We should support the addition of the arithmetic series on the stack, and the elements of the stack can be put forward or added.
- Specifically, we can maintain the second-order difference
- This result in complexity O(n)

# Problem E

- Consider the case that m>n. The last n+1 columns form a square. If we delete the last n columns, the remaining path is invariant. So we can get answer(n,m) from answer(n,m-n).
- Similar to Euclidean algorithm. The time complexity is O(log(nm)).

# Problem F

- Define $\mathrm{rank}_X(s) \coloneqq \{t \in X : t < s\}$. We sort $a, b$ at the beginning and assume $a_1 < a_2 < \cdots < a_n$ and $b_1 < b_2 < \cdots < b_m$ .
- **Observation:** $\forall p \in [1, n], \forall k \in \left[\mathrm{rank}_C(a_p), \mathrm{rank}_C(a_{p+1})\right)$ , let the $k$-th smallest string in $C$ be $c_k = a_i + b_j$, then $a_i$ is a prefix of $a_p$.
- For each query, suppose $\mathrm{rank}_C(a_i)$ can be computed efficiently, we can determine $p$ by applying binary search and reduce the problem to determine the $k$-th smallest string in $\{\alpha_i + b_j : 1 \le i \le t, 1 \le j \le m\}$, where $\alpha_1, \alpha_2, \ldots, \alpha_t = \{a_i : a_i \text{ is prefix of } a_p\}$.

# Problem F

- Let $\alpha_1, \alpha_2, \dots, \alpha_t = \{a_i : a_i \text{ is prefix of } a_p\}$.
- $\text{rank}_C(a_p) = pm - \sum_{i=1}^{t} |\{\alpha_i + b_j \geq a_p : 1 \leq j \leq m\}|$
- $|\{\alpha_i + b_j \geq a_p : 1 \leq j \leq m\}| = m - \text{rank}_B(a_p[|\alpha_i| + 1 \dots |a_p|])$
- $\text{rank}_B(a_p[|\alpha_i| + 1 \dots |a_p|])$ can be computed by suffix array efficiently.
- So it takes $O(\sqrt{L_A} \log n)$ to do the binary search.

# Problem F

- Notice that $t = O(\sqrt{L_A})$, where $L_A = \sum_{a \in A} |a|$. So we can apply a carefully implemented QuickSelect algorithm.
- QuickSelect(A, k):
  - Select a random element from A for pivot
  - Al, Ar = Partition(A, pivot)
  - if k==size(Al)+1: return pivot
  - else if k<=size(Al): return QuickSelect(Al, k)
  - else return QuickSelect(Ar, k - size(Al) - 1)

# Problem F

- Notice that the argument A in QuickSelect can be represented as $\mathrm{U}_{1\le i\le t}\{\alpha_i + b_j : j \in [l_i, r_i]\}$. So we can maintain $l_1, l_2, \ldots, l_t, r_1, r_2, \ldots, r_t$ and implement Partition in an efficient way: Notice each $l_i$ and $r_i$ can be determined by running binary search on $\{\alpha_i + b_j : j \in [1, m]\}$ (Use LCP supported by suffix array for comparison).
- The total time complexity is $O\left(L_A \log L_A + q\sqrt{L_A}\log^2 m\right)$ (Suppose $L_A = L_B, n = m$).

# Problem G

- What would the intersection of the d-set of paths looks like?
  - If the intersection of these paths exists, then the intersection of d-sets would be the d-set of the intersection of these paths.
  - If there exists two paths without intersection, then the intersection of d-sets would be the intersection of d-sets of farthest two paths, which would be a circle in the tree. The centre of circle is in the midpoint of two paths, and the radius is related with d.

# Problem G

- We can use segment tree about time, and this helps to remove the delete operations.
- If the intersection of the paths exists, then we can simply maintain the endpoint of it. It can be shown that the intersection is always an path.
- If the intersection of the paths does not exist. Then we can maintain the farthest two paths of the set. And the update could only happen between the new added one and the old one.
- These requires O(nlogn) LCA operations.

# Problem G

- We can use segment tree about time, and this helps to remove the delete operations.
- If the intersection of the paths exists, then we can simply maintain the endpoint of it. It can be shown that the intersection is always an path.
- If the intersection of the paths does not exist. Then we can maintain the farthest two paths of the set. And the update could only happen between the new added one and the old one.
- These requires O(nlogn) LCA operations, and the problem becomes
- `how to get the size of the d-set of a path?`

# Problem G

- We can transform the problem into the d-set of path from x to root if we find the LCA of the two endpoints. And for each point y of d-set, we would try to calculate it in LCA(x,y)
- Prepare the Heavy-Light segmentation of the tree. Assume that z is the father of x.
  - If x is the heavy son of z, then the point updated in z would be in his light child. There are at-most O(nlogn) points in this case, and this can be done in a 2D-point counting
  - If x is the light son of z, then this would happen for only O(logn) times. In each query would be the number of nodes in subtree which has depth smaller than D, which is another 2D-point counting.

# Problem H

- We can interpret the program as a tree structure.
  - Every node represent a single bitwise-instruction or a FOR block.
  - The father of a node $u$ is defined as the node of the smallest FOR block containing the instruction/block represented by $u$.
- For each query, we can apply binary search recursively to determine the node of the $k$-th bitwise instruction. This takes $O(d \log n)$ time, where $d$ is the depth of the tree.
- Notice that only $\log k$ FOR loops are executed for the second time. We can reformat the tree so that $d \leq \lg 10^9$ (by removing useless loops).

# Problem H

- Also, notice that bits are independent from each other
- Thus to calculate a loop quickly, we can save the “transfer function” of the 16 states for each bit, and do binary lifting.

# Problem I

- If we flip the value of point with odd depth, then the transfer between states would be:
  - `Choose two adjacent points with different values, and swap the values between them(and it's easy to find that we can remove the constraints of different values.`
- And the starting state S could reach ending state T iff the they have the same numbers of points with value 1.
- The minimum of operations required would be the sum of absolute value of the difference between the number of points with value 1 in the subtree for each subtree.

# Problem I

- In the original problem, we can simply fill 0 in starting state, and 1 in the ending state. This maximize the difference, and each possible changing makes it smaller.
- The answer is contributed from each edge, and for each edge, the contribution looks like:
  - `For all possible 01 sequence of length n with m 1s, assume that the number of 1s in the first k numbers is T, and add |T-c| into the contribution.`

# Problem I

- In the original problem, for each ? we can simply fill it so that the difference is maximized, and each possible changing makes it smaller.
- The answer is contributed from each edge, and for each edge, the contribution looks like:
  - `For all possible 01 sequence of length n with m 1s, assume that the number of 1s in the first k numbers is T, and add |T-c| into the contribution.`
- This gives a simply O(n^2) algorithm, but it could not pass the problem.

# Problem I

- Althrough it seems hard to solve these n-1 subproblems, (k,c) pairs may be very close to each other. This is because k and c is only related with the number of question marks in the subtree, and the number of 0,1 in the subtree. So the (t,c) in father node may be close to (t,c) in its heavy child.
- Also the (t,c) pare can be transferred into new (t,c) pair with Manhattan distance at most 1 if we maintain the answer, the number of 01 sequences with less then c 1s in the first t characters, and something else.
- This result in complexity O(nlogn)

# Problem J

- There are two steps to solve the problem.
- The first step is described as below:
  - a. Let V denote the vertex set of the graph initially
  - b. Suppose V={a[1],a[2],...,a[nV]}, feed the sequence a[1],a[2],...,a[nV] to the interactor. Let S={a[i] | cnt[i]=0}
  - c. Let V'=V\S. Modify V to V'. If V is not empty, return to step b.
- The total length of the sequences is not greater than n+m. Let's prove it.

# Problem J

- Let's consider the subgraph induced by the vertex set V in step b and compare it with the subgraph induced by the vertex set V' in step c. Suppose the subgraphs are G and G'.
- The number of vertices in G minus the number of vertices in G' is exactly |S|. And the number of edges in G minus the number of edges in G' is not less than |V|-|S|, because for every vertex which is not in S, there must be an edge connecting it with some vertex in S.

# Problem J

- So using a sequence whose length is |V|, we reduce the sum of the number of vertices and edges in G by at least |V|. Thus the total length of the sequences is no more than n+m, which is the initial sum of the number of vertices and edges.
- Now let V[1],V[2],...,V[k] and S[1],S[2],...,S[k] denote the V and S generated by step b each time.

# Problem J

- The second step is described as below:
  - For i = k-1 to 1 do:
    - Suppose we have found out all edges in the subgraph induced by $S[i+1] \cup ... \cup S[k]$.
    - Let T denote a subset of $S[i+1] \cup ... \cup S[k]$. Suppose $S[i]=\{v[1],v[2],...,v[s]\}$. We will find out all edges between T and $\{v[l],v[l+1],...,v[r]\}$ in the function solve(l,r,T).
    - In solve(l,r,T):
      - Suppose we already know the number of edges between x and the set $\{v[l],v[l+1],...,v[r]\}$ for every vertex x in T before solve(l,r,T)
      - Let mid=(l+r)/2. We feed the sequence v[l],v[l+1],...,v[mid], every vertex in T to the interactor.

# Problem J

- We can use this to find out the number of edges between x and the set {v[l],v[l+1],...,v[mid]} for every vertex x in T.
- Since we already know the number of edges between x and the set {v[l],v[l+1],...,v[r]} for every vertex x in T before solve(l,r,T), we can also find out the number of edges between x and the set {v[mid+1],...,v[r]}.
- Let T_left={x∈T | the number of edges between x and {v[l],...,v[mid]} is not 0}, and T_right={x∈T | the number of edges between x and {v[mid+1],...,v[r]} is not 0}. Call solve(l,mid,T_left) and solve(mid+1,r,T_right).
- The edges are found out when l=r.

- However, we don't know the number of edges between x and the set {v[l],v[l+1],...,v[r]} for every vertex x in T before we call solve(1,s,S[i+1]∪...∪S[k]).

# Problem J

- So we should feed this sequence to the interactor before we call solve(1,s,S[i+1]∪...∪S[k]): every vertex in S[i], every vertex in S[i], every vertex in S[i+1]∪...∪S[k].
- To deal with self loops, the "every vertex in S[i]" appears twice.

- The total length of the sequences is not greater than 2n+m+0.5nlogn+mlogn.
- For the sequences every vertex in S[i], every vertex in S[i], every vertex in S[i+1]∪...∪S[k], the total length is not greater than 2n+m, which is similar to the first step.

# Problem J

- For the sequences v[l],v[l+1],...,v[mid], every vertex in T:
  - Consider v[l],v[l+1],...,v[mid]. If we use v[l],v[l+1],...,v[r] instead of v[l],v[l+1],...,v[mid], obviously every v will appear log|S[i]| times in solve(1,s,S[i+1]∪...∪S[k]), which means the total length will be |S[i]|log|S[i]|. So the real length will be 0.5|S[i]|log|S[i]|. And we can easily find out that Σ0.5|S[i]|log|S[i]|≤0.5nlogn.
  - Consider every vertex in T. For every edge between S[i] and S[i+1]∪...∪S[k], it will have a contribution of log|S[i]|, which is not greater than logn. So the total length will not exceed mlogn.

# Problem J

- So the total length of the two steps is not greater than $3n+2m+0.5n\log n+m\log n$, which is exactly 176000 when $n=4000$, $m=10000$.

# Problem K

- BFS from all source points，every time we go from a point to another point or go from an edge to the next edge in clockwise direction.
- The time complexity is O((n^2)m) or O(nmlogn).

# Thank you!


icpc.foundation
icpc | university commons
icpc | asia
icpc | nerc