SPb SU Contest: LVII SPb SU Championship

August 27, 2023

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "`spbsu`" without changing the relative order of letters of this string.

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "`spbsu`" without changing the relative order of letters of this string.
- The key observation is that the cost of each non-chosen letter depends on its position inside the chosen subsequence. Letters near either end of the word cost one swap, while letters close to the center cost two swaps.

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "`spbsu`" without changing the relative order of letters of this string.
- The key observation is that the cost of each non-chosen letter depends on its position inside the chosen subsequence. Letters near either end of the word cost one swap, while letters close to the center cost two swaps.
- Now, there are two possible approaches.
  1. A dynamic programming `dp[n][k]`: the minimum number of swaps, if we have considered the first n characters of the input string and typed k of them into the string "`spbsu`".

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "`spbsu`" without changing the relative order of letters of this string.
- The key observation is that the cost of each non-chosen letter depends on its position inside the chosen subsequence. Letters near either end of the word cost one swap, while letters close to the center cost two swaps.
- Now, there are two possible approaches.
  1. A dynamic programming `dp[n][k]`: the minimum number of swaps, if we have considered the first n characters of the input string and typed k of them into the string "`spbsu`".
  2. A greedy algorithm that tries all possibilities for the central letter "`b`" and then selects other letters of the string "`spbsu`" to be as close to the chosen letter as possible.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.
- This problem has a proven deterministic solution (that we are going to discuss), but may also be solved by many randomizations that we can't neither prove nor break.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.
- This problem has a proven deterministic solution (that we are going to discuss), but may also be solved by many randomizations that we can't neither prove nor break.
- The algorithm itself is simple. We need to build a spanning tree and find the GCD of all cycles that have at most two edges outside the tree. It will be the answer.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.
- This problem has a proven deterministic solution (that we are going to discuss), but may also be solved by many randomizations that we can't neither prove nor break.
- The algorithm itself is simple. We need to build a spanning tree and find the GCD of all cycles that have at most two edges outside the tree. It will be the answer.
- Let g be the answer found by our algorithm. This number obviously cannot be smaller (in this context 0 is infinity) than the real answer, so we only have to prove that it is not greater or, equivalently, that g divides the length of each cycle.

Many Many Cycles

- Let a cycle be *simple* if it contains at most one edge outside the tree. The condition we check verifies that all simple cycles are divisible by g and also that the doubled length of the intersection of any two simple cycles is divisible by g.

Many Many Cycles

- Let a cycle be *simple* if it contains at most one edge outside the tree. The condition we check verifies that all simple cycles are divisible by g and also that the doubled length of the intersection of any two simple cycles is divisible by g.
- Note that the intersection of any number of simple cycles is actually an intersection of some two of them.

Many Many Cycles

- Let a cycle be *simple* if it contains at most one edge outside the tree. The condition we check verifies that all simple cycles are divisible by g and also that the doubled length of the intersection of any two simple cycles is divisible by g.
- Note that the intersection of any number of simple cycles is actually an intersection of some two of them.
- Any cycle could be represented as a XOR of several simple cycles, therefore, its length could be found as a linear combination with integer coefficients of lengths of those cycles and doubled lengths of their intersections. All summands are divisible by g, therefore, g divides the length of the cycle as well.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.
- Consider a bipartite graph with vertices denoting diagonals and edges denoting their intersections (i.e., squares on a chessboard).
- We can consider white and black squares independently, and we need to find a maximal matching in this graph.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.
- Consider a bipartite graph with vertices denoting diagonals and edges denoting their intersections (i.e., squares on a chessboard).
- We can consider white and black squares independently, and we need to find a maximal matching in this graph.
- Note that each vertex in one part is connected to a segment of vertices in another.

Bishops

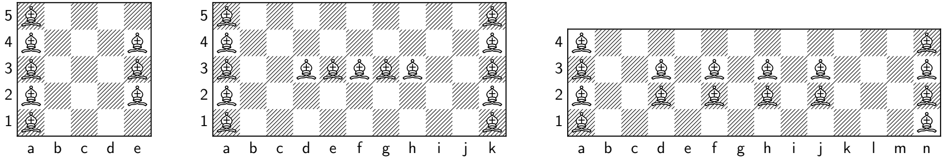
- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.
- Consider a bipartite graph with vertices denoting diagonals and edges denoting their intersections (i.e., squares on a chessboard).
- We can consider white and black squares independently, and we need to find a maximal matching in this graph.
- Note that each vertex in one part is connected to a segment of vertices in another.
- Finding a matching in a graph with this property is easy: we can simply sort all segments by their right ends and greedily choose the leftmost suitable vertex for each of them.

Bishops

- Now, the explicit construction.
- Rotate the board horizontally and then place bishops in cells with the following priority.
    1. On the left edge of the board.
    2. On the right edge of the board.
    3. On the one or two (depending on the parity of the board dimensions) middle rows going from the left to the right.

Bishops

- Now, the explicit construction.
- Rotate the board horizontally and then place bishops in cells with the following priority.
  1. On the left edge of the board.
  2. On the right edge of the board.
  3. On the one or two (depending on the parity of the board dimensions) middle rows going from the left to the right.
- Some examples:



Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.

Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.
- There are two approaches: sqrt-decomposition and bitsets. The first one can lead to two different solutions, so we start with it.

Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.
- There are two approaches: sqrt-decomposition and bitsets. The first one can lead to two different solutions, so we start with it.
- The first solution. Naive.
- Arrange vertices into two types: light and heavy, depending on their degree in the original graph.

Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.
- There are two approaches: sqrt-decomposition and bitsets. The first one can lead to two different solutions, so we start with it.
- The first solution. Naive.
- Arrange vertices into two types: light and heavy, depending on their degree in the original graph.
- For each heavy vertex, we will explicitly store all its neighbors of its color and the opposite color separately. To maintain this, when changing the color of any vertex, we will need to make at most $\mathcal{O}(\sqrt{m})$ additions and deletions, and at most one swap of the lists (if the recolored vertex is heavy).

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}\left(\sqrt{m}\right)$ edges.

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}(\sqrt{m})$ edges.
- The second solution. Short and without constants.
- Get rid of a queue with light edges! Instead, we can treat all vertices as heavy, but only store edges to vertices with smaller of equal initial degrees.

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}(\sqrt{m})$ edges.
- The second solution. Short and without constants.
- Get rid of a queue with light edges! Instead, we can treat all vertices as heavy, but only store edges to vertices with smaller of equal initial degrees.
- The only problem is the number of heavy vertices, which we solve by maintaining vertices with non-empty opposite-color neighbour lists in a lazy queue.

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}\left(\sqrt{m}\right)$ edges.
- The second solution. Short and without constants.
- Get rid of a queue with light edges! Instead, we can treat all vertices as heavy, but only store edges to vertices with smaller of equal initial degrees.
- The only problem is the number of heavy vertices, which we solve by maintaining vertices with non-empty opposite-color neighbour lists in a lazy queue.
- The running time of both solutions is $\mathcal{O}\left(m + q\sqrt{m}\right)$ (assuming $n \leq m$ and $q > 0$).

Graph Cuts

- There is an alternative approach with bitsets.
- Let's store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).

Graph Cuts

- There is an alternative approach with bitsets.
- Let's store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).
- Each add/remove operation is just a XOR operation of some matrix row with our current set. Each "?" query is just finding true bit in a bitset and flipping it.

Graph Cuts

- There is an alternative approach with bitsets.
- Let's store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).
- Each add/remove operation is just a XOR operation of some matrix row with our current set. Each "?" query is just finding true bit in a bitset and flipping it.
- The challenging part is avoiding Memory Limit Exceeded. To do so, we need to compress the incidence matrix.
- One way of doing this is to store it as a list of pairs of an index and a mask value.

Graph Cuts

- There is an alternative approach with bitsets.
- Let's store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).
- Each add/remove operation is just a XOR operation of some matrix row with our current set. Each "?" query is just finding true bit in a bitset and flipping it.
- The challenging part is avoiding Memory Limit Exceeded. To do so, we need to compress the incidence matrix.
- One way of doing this is to store it as a list of pairs of an index and a mask value.
- The complexity is $\mathcal{O}(m/w)$ per query. Some secondary optimizations may be required to get Accepted.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.
- Split the array into blocks of size B (will be defined later), compute prefix and suffix sums inside each block and construct the sparse table with $k - 2$ operations on the blocks themselves.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.
- Split the array into blocks of size B (will be defined later), compute prefix and suffix sums inside each block and construct the sparse table with $k - 2$ operations on the blocks themselves.
- Now, we can answer any query that is not contained inside a single block.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.
- Split the array into blocks of size B (will be defined later), compute prefix and suffix sums inside each block and construct the sparse table with $k - 2$ operations on the blocks themselves.
- Now, we can answer any query that is not contained inside a single block.
- To answer any other query we need to construct the same sparse table recursively inside each block.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B?

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.
- For example, in our case $B \approx \sqrt{n}$, and if we had one operation more, the optimal choice would have been $B \approx \log n$.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.
- For example, in our case $B \approx \sqrt{n}$, and if we had one operation more, the optimal choice would have been $B \approx \log n$.
- It could be shown that this construction is optimal both in time and memory.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.
- For example, in our case $B \approx \sqrt{n}$, and if we had one operation more, the optimal choice would have been $B \approx \log n$.
- It could be shown that this construction is optimal both in time and memory.
- In case we know that the number of queries is $\Theta(n)$, we can achieve complexity $\Theta(n\alpha(n))$ where α is the inverse Ackermann function (and this is also optimal).

Operations per query	Time to construct (Θ omitted)
0	n^2
1	$n \log n$
2	$n \log \log n$
3	$n \log^* n$
4	$n \log^* n$
5	$n \log^{**} n$
6	$n \log^{**} n$
7	$n \log^{***} n$
$\vdots$	$\vdots$
k + 1	$n \log^{\overbrace{* * \cdots *}^{\lfloor k/2 \rfloor}} n$
$\vdots$	$\vdots$