Problem Tutorial: "Permutation"

We try to find the number of permutations that contain no arithmetic progression of length at least 3.

For small n ($n \le 20$), we have a very simple dynamic programming solution. Let $dp(x, S)$ be the number of ways such that the first x numbers are filled and the set of numbers used is S. We can simply enumerate an $i \notin S$ such that there is no pair of (j, k) where $j \in S$, $k \notin S$ and $2i = j + k$. And transfer from $dp(x, S)$ to $dp(x + 1, S \cup \{i\})$. The time complexity is $O(2^n \cdot n^2)$.

For large n, the restriction is actually very strong and the number of valid states S for each x is very small. When $n = 50$, the number of states is roughly 10^6. We can use hash map or just sorted vector to maintain the value. The time complexity is $O(n^2 \cdot |S|)$.

We can also use some bitwise operations to eliminate the factor n in time complexity. For each i, we only need to find some j and k such that $k = 2i - j$. If we store S in unsigned 64-bit integer, swap the high and low 32 bits of S, we can get the representation of $-j$. And with some bitwise shift operations, we can get all valid k. After that we can check if some i is valid in $O(1)$.

Problem Tutorial: "Tree Product"

Apparently, we can ignore all trees with one vertex.

Consider the diameter of $A \times B$ for two trees A and B, it would be

$$\max\{\text{diameter}(A) + 2 \cdot \text{height}(B), \text{diameter}(B)\}.$$

Since $2 \cdot \text{height}(B)$ will always be greater than or equal to $\text{diameter}(B)$. The diameter of $A \times B$ will be $\text{diameter}(A) + 2 \cdot \text{height}(B)$.

For a given permutation $p_1, p_2, \ldots, p_n$, the diameter of $T_{p_1} \times T_{p_2} \times \ldots \times T_{p_n}$ will be

$$\text{diameter}(T_{p_1}) + 2 \cdot \sum_{i=2}^{n} \text{height}(T_{P_i})$$

Just compute the value of $2 \cdot \sum_{i=1}^{n} \text{height}(T_i)$ and enumerate p_1, then we can easily find the minimum diameter and the maximum diameter.

Problem Tutorial: "Distinct Number"

Let's build the trie of the n intervals. The size maybe very large, but we can ignore the subtrees which contain all the possible leaves (i.e. complete binary tree). The actual size of this compressed trie will be $O(n \log A)$, where $A = \max(r_i)$.

Consider the i-th bit of y:

- If it is 0, for each node in the trie, we need to merge the right subtree to the left subtree.
- If it is 1, we do nothing for each node in the trie.

It can be proved that if we have a proper implementation for merging, the time complexity will be $O(n^2 \log A)$ or even $O(n^2)$:

- Implement the trie using pointers for left and right children.
- Merge the subtree of size a and b in $O(\min(a, b))$ time.
- When meeting a subtree which is a complete binary tree, terminal this merging.

The number of leaves in the final trie is the size of S.

Problem Tutorial: "Fibonacci Partition"

Consider the *Zenkorf representation* of X: $X = \sum_{i=1}^{\infty} a_i \cdot F_i$, where $a_i \in \{0, 1\}$ and $a_i \cdot a_{i+1} = 0$.

Let the indices of non-zero terms a_i be $p_1, p_2, \ldots, p_m$. We can use the following greedy algorithm to find the answer:

1. Start from $answer = 0$ and $last = 0$.
2. For each $i = 1, 2, \ldots, m$:
   - if $p_i - last \geq 3$, let $answer$ be $answer + \lceil \frac{p_i - last}{2} \rceil$ and $last$ be $p_i - 1$;
   - otherwise, let $answer$ be $answer + 1$ and $last$ be p_i.

If we use a binary search tree (like Treap) to maintain the sequence $\{p_i\}$, the value of $answer$ can be calculated easily. The problem is how to maintain the sequence when X changes.

Let's change the definition of F_n in the problem: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$, and generalize the definition to negative integers n. We can see that $F_{-n} = (-1)^{n+1} F_n$.

Combining with Lucas numbers, $L_0 = 2,\ L_1 = 1,\ L_n = L_{n-1} + L_{n-2}$, we have:

$$L_k \cdot F_n = F_{n+k} + (-1)^k F_{n-k}$$

If we partition a into several Lucas numbers (actually, the greedy algorithm also leads to a unique representation like Fibonacci number), with the above formula, we only need to care about add / subtract a Fibonacci number to / from X. This makes the problem easier.

After some observations, we can find that the change only affects the maximal consecutive *alternating segment* (a segment of the form `10101...10101`) in the *Zenkorf representation* of X. So, we can use binary search tree to maintain the maximal consecutive segments.

The following article helps a lot: `http://www.algonotes.com/en/fibonacci-arithmetic/`.

Let's consider how to add F_x to X first:

1. If $a_{x-1} = a_x = a_{x+1} = 0$, we can simply make $a_x = 1$.
2. At least one of a_{x-1}, a_x and a_{x+1} is non-zero. Assume the lower and upper bounds of the maximal consecutive alternating segment containing this non-zero value are l and r:
   - $x = r + 1$, we need to make $a_r = 0$ and $a_{r+2} = 1$.

- $l \le x \le r$ and $x \equiv l \pmod 2$, we need to make $a_l = a_{l+2} = \ldots = a_r = 0$ and $a_{l+1} = a_{l+3} = \ldots = a_{x-1} = a_{r+1} = a_{l-2} = 1$.
- $l \le x \le r$ and $x \not\equiv l \pmod 2$, we need to make $a_{x+1} = a_{x+3} = \ldots = a_r = 0$ and $a_{r+1} = 1$.

Note that if we make a_{r+2} or a_{l-2} be 1, adjacent 1s may appear in the *Zenkorf representation*. To eliminate this, we can just repeat the above procedure recursively. And only constant number of recursive calls will trigger.

As for subtracting a Fibonacci number F_x from X, we can use similar analysis.

The whole time complexity will be $O(n \log^2 A)$, where $A = \max(a_i, b_i)$.

Problem Tutorial: "Longest Common Subsequence"

This problem tries to find a common subsequence in the form $1^x 2^y 3^z$ and the value $x+y+z$ is maximum.

Let's try to fix x and z, we need to find the x first 1 and the z last 3 in a or b. After that, let the remaining subsegment in a and b be $[l_a, r_a]$ and $[l_b, r_b]$. The value of y will just be the minimum occurrence of 2 in each of the subsegment.

Let $sa(i)$ be the number 2 in $a_1, a_2, \ldots, a_i$ and $sb(i)$ be the number of 2 in $b_1, b_2, \ldots, b_i$. We can formula the above description as:

$$\min(sa(r_a) - sa(l_a - 1), sb(r_b) - sb(l_b - 1)).$$

Consider $sa(r_a) - sa(l_a - 1) \ge sb(r_b) - sb(l_b - 1)$ first, i.e. $sa(r_a) - sb(r_b) \ge sb(l_b - 1) - sa(l_a - 1)$. The optimal value is $sb(r_b) - sb(l_b - 1) + z$. We can use a data structure to maintain the $key = sa(r_a) - sb(r_b)$ and $value = z - sb(l_b - 1)$. And if we enumerate x from n to 1, and add the corresponding item with $key = sa(r_a) - sb(r_b)$ into the data structure, we can query the suffix minimum value of $key \ge sb(l_b - 1) - sa(l_a - 1)$ to get the optimal value.

And for $sa(r_a) - sa(l_a - 1) \le sb(r_b) - sb(l_b - 1)$, the case is similar. Since we only need to query suffix minimum value, we can use binary index tree.

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

And there also exists an $O(n)$ solution which is left as a challenge for participants.

Problem Tutorial: "Necklace"

Let the number of gems of color i in the the necklace be $cnt(i)$. A necklace of length m is good iff $m \ge 2 \cdot \max_i(cnt(i))$.

Let the maximum value of $cnt(i)$ be x. For gems with color i, only the largest x values could be on the necklace: let it be V_i.

Let V be $V_1 \cup V_2 \cup \ldots \cup V_n$. If we sort the values in V in decreasing order, the first $2x$ (except $x = 1$, we need at least 3 values) must be taken. For the remaining values, we can take all positive values. This will definitely give us the best solution.

We can use priority queue to speed up the procedure. When x becomes $x + 1$, only new values will be added to V. We can use a minimum heap A to maintain the first $2x$ values, and a maximum heap B to maintain the remaining values. For the newly added values, we add them to B first. After that, take top values from B to make the size of A at least $2(x+1)$. Also, take top positive values from B to A.

As for the solution, we can find the optimal x and try to simulate the above procedure using the optimal x to get a valid solution.

Problem Tutorial: "Paper-cutting"

Let's find out all possible rectangles after several foldings first. Apparently, we can consider the row and column folding separately. We only focus the row foldings.

Let $dp(i)$ denote whether the i-th row could be the left border of the final rectangle. It can be see that $dp(i)$ is true if there exists some j that $dp(j)$ is true and $r_j r_{j+1} \dots r_{2i-j-1}$ is palindrome.

We can use Manacher's algorithm to find the palindromic radius r_i for each even length palindrome, and maintain the maximum index j that $dp(j)$ is true, and check whether $i - r_i \le j$ to determine the value of $dp(i)$.

As for the possible right, up and down border of the final rectangle, the process is similar.

Now, we can have a $O(n^2m^2)$ solution to enumerate all possible borders and find the number of connected components.

Actually, only $O(nm)$ possible rectangle should be considered. For each left border l_i, we can find a minimum right border $r_i \ge l_i$. And for each up border u_j, we can find a minimum down border $d_j \ge u_j$. For each l_j and u_j, it can be showed that checking r_i and d_j is enough. Since large r_i or d_j will definitely make the number of connected components larger.

For each connected component, we can find a minimum-area rectangle $C_k = (x_1, x_2, y_1, y_2)$ (with sides parallel to the matrix sides) that covers all cells of the component. For a possible final rectangle $R_{i,j} = (l_i, r_i, u_j, d_j)$, the number of connected components is the number of rectangles C_k which has intersection with $R_{i,j}$.

We can do a sweep line on the l_i and add each C_k that the range $[x_1, x_2]$ has intersection with range $[l_i, r_i]$. And in the mean time, we can maintain the value of y_1 and y_2 using two binary index trees. For each u_j and d_j, the the number of connected components could be answered using the two binary index trees.

There is a hidden assumption make the above algorithm work: $r_1 \le r_2 \le \dots \le r_s$ and $d_1 \le d_2 \le \dots \le d_t$.

The overall time complexity will be $O(nm \log nm)$.

Problem Tutorial: "Partition Number"

Let $p(m)$ be the number of solutions of equation $x_1 + x_2 + \dots + x_k = m$ such that $x_1 \le x_2 \le \dots \le x_k$. Actually, it is the partition number. And let $odd(x)$ and $even(x)$ be the number of ways summing up to x using odd or even distinct numbers from A, respectively.

The answer for the problem will be:

$$\sum_{i=0}^{m}(even(i) - odd(i)) \cdot p(m-i)$$

which can be proved using inclusion-exclusion principle.

The value of $odd(x)$ and $even(x)$ can be calculated using simple dynamic programming. And the value of $p(x)$ can be calculated using the classic dynamic programming or pentagonal number theorem in $O(m\sqrt{m})$. Also, polynomial inversion and fast Fourier transform work in $O(m \log m)$.

Problem Tutorial: "Stirling Number"

Let us calculate the value of $(\sum_{k=0}^{m} {n \brack k}) \bmod p$.

Consider the proof of Lucas' theorem:

$$\binom{n'p+n_0}{k'p+k_0} \equiv \binom{n'}{k'}\binom{n_0}{k_0} \pmod p$$

Using $(x+1)^{n'p+n_0} = (x+1)^{n'p}(x+1)^{n_0}$, $(x+1)^p \equiv= x^p + 1 \pmod p$ and binomial expansion, we can easily prove the Lucas' theorem.

Now, is well known that the coefficients of $s_n(x) = x(x+1)(x+2)\dots(x+n-1)$ are ${n \brack 0}, {n \brack 1}, \dots, {n \brack n}$.

We can find that $s_p(x) \equiv x^p - x \pmod p$.

Let $n = n'p + n_0$, and then

$$\begin{aligned} s_n(x) &= \prod_{t=0}^{n'-1}(x+tp)(x+tp+1)\dots(x+tp+p-1) \cdot \prod_{r=0}^{n_0-1}(x+n'p+r) \\ &\equiv \prod_{t=0}^{n'-1} x(x+1)\dots(x+p-1) \cdot \prod_{r=0}^{n_0-1}(x+r) \equiv s_p^{n'}(x)s_{n_0}(x) \\ &\equiv (x^p - x)^{n'} s_{n_0}(x) \pmod p \end{aligned}$$

Using the binomial expansion, we have:

$${n \brack k} \equiv (-1)^{n'-j}\binom{n'}{j}{n_0 \brack i} \pmod p$$

where $k = n' + j(p-1) + i$, and $0 \le i < p-1$ if $n_0 = 0$ or $0 < i \le p-1$ if $n_0 > 0$.

After enumerating i, we only need to find the value of:

$$\sum_{j=0}^{\lfloor \frac{m-i-n'}{p-1} \rfloor} (-1)^{n'-j}\binom{n'}{j}$$

We can rephrase it as: given n, m, and x, find the value of $\sum\limits_{i=0}^{m} \binom{n}{i} x^i$.

Notice that we can use a process like Lucas' theorem to divide x^n: $x^n \equiv x^{\lfloor \frac{n}{p} \rfloor} \cdot x^{n \bmod p} \pmod p$. So x^i and $\binom{n}{i}$ can be divided together using the Lucas' theorem.

After that we have:

$$\sum_{i=0}^{m}\binom{n}{i}x^i=\sum_{t=0}^{\lfloor\frac{m}{p}\rfloor-1}\binom{\lfloor\frac{n}{p}\rfloor}{t}x^t\cdot\sum_{r=0}^{p-1}\binom{n \bmod p}{r}x^r+\binom{n \operatorname{div} p}{m \operatorname{div} p}\cdot x^{m \operatorname{div} p}\cdot\sum_{r=0}^{m \bmod p}\binom{n \bmod p}{r}x^r$$

Since n is a given number, there are only $O(\log n)$ different values of $n \bmod p$. We can precalculate the prefix sum of $\binom{n \bmod p}{r}x^r$ for each distinct $n \bmod p$.

Or just use the following formula to speed up:

$$\sum_{k=0}^{m}(-1)^k\binom{s}{k}=(-1)^m\binom{s-1}{m}$$

After that, we can use divide and conquer to find the coefficients of $s_{n_0}(x)$ in $O(n_0\log n_0)$ time.

The total time complexity will be $O(p\log p)$.

Problem Tutorial: "Ternary String Counting"

Let's try the $O(n^3)$ solution first. Let $\mathit{ways}(i,j,k)$ be the number of ways that we have to fill the values of $s_1,s_2,\ldots,s_i$, and the nearest two different characters are at position j and k ($i>j>k$).

Each of the m restrictions actually determines the ranges of j and k. For each valid j and k that: $j\in[\mathit{jmin}_i,\mathit{jmax}_i]$ and $k\in[\mathit{kmin}_i,\mathit{kmax}_i]$, we have the following transitions:

1. $\mathit{ways}(i+1,i,k)\overset{+}{\leftarrow}\mathit{ways}(i,j,k)$
2. $\mathit{ways}(i+1,i,j)\overset{+}{\leftarrow}\mathit{ways}(i,j,k)$
3. $\mathit{ways}(i+1,j,k)\overset{+}{\leftarrow}\mathit{ways}(i,j,k)$

We can see easily that the first dimension of $\mathit{ways}(\cdot,\cdot,\cdot)$ is useless: in the first two cases, the second dimension will always be i, and in the third case, j and k will never change.

Actually the transitions are equivalent to: for a given rectangle R, make all the entries outside R be zero and:

1. find the sum of a specific row,
2. find the sum of a specific column,
3. do not change the values.

We can also see that, if an entry becomes zero, it will always be zero. And for a fixed row, the non-zero entries form a consecutive range.

For each row i of the dynamic programming array, we can maintain the boundaries left_i and right_i of the non-zero entries. For each row and column, we can maintain the row-sum and column-sum.

For each rectangle R, we can just clear the entries for each row and maintain the row-sum or column-sum in the mean time.

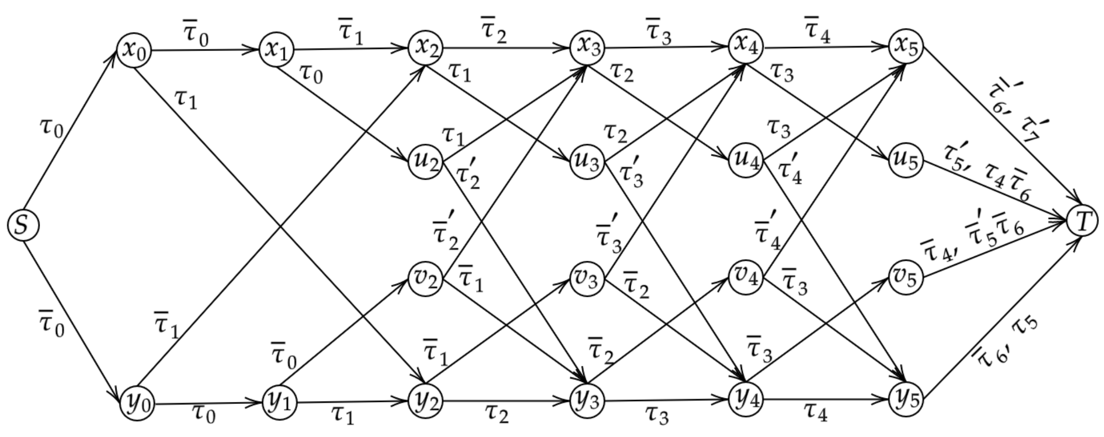
The time and space complexity is $O(n^2)$.

Problem Tutorial: "Anti-hash Test"

Let's build the suffix automaton of the *Thue-Morse word* of order m and compact all states with only single outgoing edge. Actually, this is the *CSAM* of the *Thue-Morse word* of order m.

For large m ($m > 5$), we can find some periodic patterns in the *CSAM*. We can use brutal method for $m \leq 5$ and clever method based on the periodic patterns for $m > 5$.

The figure below is the *CSAM* of *Thue-Mors word* of order m:



More specifically, the leftmost state S is the starting state and the rightmost state T is the ending state. If we label the states in the first row as $x_0, x_1, \ldots, x_{n-2}$ and label the states in the last row as $y_0, y_1, \ldots, y_{n-1}$ and the middle ones are $u_2, u_3, \ldots, u_{n-2}$ and $v_2, v_3, \ldots, v_{n-2}$, the transition in each edge will be:

$x_i \xrightarrow{\overline{\tau}_i} x_{i+1}, \quad x_i \xrightarrow{\tau_{i-1}} u_{i+1}$

$y_i \xrightarrow{\tau_i} y_{i+1}, \quad y_i \xrightarrow{\overline{\tau}_{i-1}} v_{i+1}$

$u_i \xrightarrow{\tau_{i-1}} x_{i+1}, \quad u_i \xrightarrow{\tau'_i} y_{i+1}$

$v_i \xrightarrow{\overline{\tau}_{i-1}} y_{i+1}, \quad v_i \xrightarrow{\overline{\tau}'_i} x_{i+1}$

where τ_i is the *Thue-Morse word* of order i, $\overline{\tau}_i$ is the inversion of τ_i, $\tau'_i = \overline{\tau}_{i-2}\overline{\tau}_{i-1}$, $\overline{\tau}'_i$ is the inversion of τ'_i.

The transitions from $x_{n-2}, y_{n-2}, u_{n-2}, v_{n-2}$ to T are different:

$x_{n-2} \xrightarrow{\overline{\tau}'_{n-1}} T, \quad x_{n-2} \xrightarrow{\tau'_n} T$

$y_{n-2} \xrightarrow{\overline{\tau}_{n-1}} T, \quad y_{n-2} \xrightarrow{\tau_{n-2}} T$

$u_{n-2} \xrightarrow{\tau'_{n-2}} T, \quad u_{n-2} \xrightarrow{\tau_{n-3}\overline{\tau}_{n-1}} T$

$v_{n-2} \xrightarrow{\overline{\tau}_{n-3}} T, \quad v_{n-2} \xrightarrow{\overline{\tau}'_{n-2}\overline{\tau}_{n-1}} T$

And in addition, $T, x_{n-2}, y_{n-3}, x_{n-4}, y_{n-5}, \ldots$ are the accepting states.

Actually, we don't need to build the whole *CSAM*: only the first $O(\log |u|)$ states are enough.

We can find the corresponding state of string u and find the size of the right set. The size of the right set is the number of different path ending in an accepting state, which can be calculated easily. This solves the first problem.

As for the second problem, we need to find the right set of other states. And also only $O(\log |u|)$ states are needed. Since the size of $\text{Right}(x_i)$ is greater than or equal to $\text{Right}(x_{i+1})$.

Problem Tutorial: "Tokens on the Tree"

Let's try to fix w and b first. And without loss of generality, we can assume $w \geq b$.

Let S be the set of vertices which must be in the white connected component.

- $S \neq \varnothing$. After deleting all vertices in S, we will have some connected components. We can see that $f(w, b)$ will be the number of connected components whose size is at least b.
- $S = \varnothing$. We can see that $f(w, b)$ will only be 1 or 2. And $f(w, b)$ will be 1 if and only if there exists a vertex u such that, after deleting this vertex, there exist at least three connected components, the size of two of them $\geq w$ and the size of the remaining one $\geq b$.

We can solve the problem based on the above observation.

Let's try to fix the value of w. A vertex x must be in the white connected component if an only if the maximum size of the subtree of x is less than or equal to w.

When $w = n$, all the vertices are in the white connected component. As w decreases, the size of S will decrease too. We can use a disjoint-set data structure to maintain the sizes of the connected components.

For the fixed w, if S is nonempty, we need to find the value of

$$w \cdot \left(\sum_{b=1}^{\min(w, n-w)} b \cdot f(w, b) \right).$$

When maintaining the size of the remaining connected components, we can use a segment tree or binary index tree to maintain the value of $s(x)$, which means the number of connected components whose size $\geq x$, and the value of $s(x) \cdot x$.

The value of $\sum_{b=1}^{\min(w,n-w)} b \cdot f(w, b)$ is just a range sum in the segment tree or binary index tree.

If S is empty, we need to maintain some other information. We only need to know the number of b which makes $f(w, b) = 1$. For each vertex x, we need to maintain whether it has at least two subtrees with size $\geq w$ and the size of the third largest subtrees.

For each vertex x, we maintain the sorted list L_x of subtree in the decreasing order of subtree size. As w decreases, we can pop the corresponding subtree from each L_x and find the maximum value *bound* of the size of the third largest subtree.

For $b \leq \min(w, \mathit{bound})$, the value of $f(w, b)$ is 1. And for $\mathit{bound} < b \leq w$, the value of $f(w, b)$ is 2.

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

Problem Tutorial: "A + B Problem"

The following greedy method works for this problem, the proof is omitted: we try to fill 0 in the least significant bit. And in this way, we will maximize the benefit of each 1.

Consider assigning each bit from left to right:

- If it is 0, we will try to assign this 0 to the number which has less bits unassigned.
- If it is 1, we will first assign this 1 to the number which has less bits unassigned. Assume this 1 is in the x-th position, we need to check if we can assign 1 to another number until the x-th position and still have enough bits to assign the remaining unassigned bits. If this is true, we can keep this number (it makes no difference to which number we assign this 1, and we should keep the higher bit for the incoming 1s). Otherwise, we should assign this 1 to another number.