Some Notes on Algorithms

Ash Vaughn

April 23, 2026

Contents

1 Foreword

Most of the algorithms contained are highly practical and have been implemented in library.

2 Linear Algebra

2.1 Characteristic Polynomial

To compute the characteristic polynomial $p(x) = \det(xI - A)$ of an $n \times n$ matrix A, the algorithm first reduces A to an upper Hessenberg matrix H (where $H_{i,j} = 0$ for all $i > j + 1$) in $O(n^3)$ time. It then evaluates the characteristic polynomial using a dynamic programming approach.

Lemma 2.1 (Hessenberg Characteristic Polynomial Recurrence)**.** *Let H_k be the $k \times k$ leading principal submatrix of an upper Hessenberg matrix H, and let $p_k(x) = \det(xI_k - H_k)$ be its characteristic polynomial. With the base case $p_0(x) = 1$, the polynomials satisfy the recurrence:*

$$p_{k+1}(x) = (x - H_{k,k})p_k(x) - \sum_{i=0}^{k-1} \left(H_{i,k} \prod_{j=i}^{k-1} H_{j+1,j} \right) p_i(x)$$

Proof. We evaluate $p_{k+1}(x) = \det(xI_{k+1} - H_{k+1})$ by performing a Laplace expansion along its last column. The entries in the last column of $xI_{k+1} - H_{k+1}$ are $-H_{0,k}, -H_{1,k}, \ldots, -H_{k-1,k}$, with the diagonal entry being $(x - H_{k,k})$.

The cofactor for the diagonal entry $(x - H_{k,k})$ is the determinant of the top-left $k \times k$ block, which is $p_k(x)$.

For any other entry $-H_{i,k}$ (where $i < k$) in the last column, removing its row and column leaves a block upper-triangular matrix. The top-left block is $(xI_i - H_i)$, which has determinant $p_i(x)$. Because H is upper Hessenberg, the bottom-right block is upper triangular, with its main diagonal consisting entirely of the subdiagonal elements of H: $-H_{i+1,i}, -H_{i+2,i+1}, \ldots, -H_{k,k-1}$. The alternating signs from the cofactor expansion cancels with the negative signs of these subdiagonal elements. □

2.1.1 Determinant of Affine Matrix Polynomial

The algorithm computes the polynomial $P(x) = \det(A + xB)$ in $O(n^3)$ time.

The algorithm samples a random scalar $a \in \mathbb{F}$ and evaluates the matrix at this point, defining $A' = A + aB$. Assuming A' is invertible, we can factor it out of the original polynomial:

$$\begin{aligned} P(x) &= \det(A + aB + (x - a)B) \\ &= \det(A' + (x - a)B) \\ &= \det(A') \det(I + (x - a)A'^{-1}B) \end{aligned}$$

Let $M = -A'^{-1}B$. The polynomial becomes $P(x) = \det(A') \det(I - (x - a)M)$.

Notice that $\det(I - tM) = t^n \det(t^{-1}I - M) = t^n p_M(t^{-1})$, the reversed characteristic polynomial of M. Let $R(t)$ denote this reversed polynomial. The algorithm computes $p_M(t)$ using the Hessenberg DP, reverses its coefficients to obtain $R(t)$, multiplies the polynomial by the scalar $\det(A')$, and performs a Taylor shift $x \mapsto x - a$ to recover $P(x) = \det(A')R(x - a)$.

Lemma 2.2 (Failure Probability of the Affine Determinant)**.** *If $P(x) = \det(A + xB)$ is not identically the zero polynomial, the probability that the randomized reduction algorithm incorrectly fails (by returning 0) is bounded by $\frac{n}{|\mathbb{F}|}$.*

Proof. The algorithm only fails if the shifted matrix $A' = A + aB$ is not invertible, if and only if $P(a) = 0$.

The algorithm samples a uniformly and out of all possible random choices for a, at most n of them can cause A' to be singular. If $P(x)$ is identically the zero polynomial, the algorithm will correctly find $\det(A') = 0$ for any a. □

2.2 Frobenius Form

2.2.1 Algorithm Description

To compute the Frobenius form of a matrix $A \in \mathbb{F}^{n \times n}$ in $O(n^3)$ time, the algorithm employs a randomized Krylov subspace approach. The Frobenius form decomposes the vector space V into a direct sum of A-cyclic subspaces $V = \bigoplus_{i=1}^{m} W_i$. This yields a block-diagonal matrix where each block is the companion matrix of an invariant factor $P_i(z)$ of A. By the structure theorem of finitely generated modules over a PID, these invariant factors divide each other: $P_m(z) \mid P_{m-1}(z) \mid \cdots \mid P_1(z)$.

1. **State:** Maintain a set of generator vectors $\{w_1, \ldots, w_k\}$ for the resolved cyclic subspaces $W_1, \ldots, W_k$, and a row-reduced basis $\mathcal{B}$ representing their combined span $W = \bigoplus_{i=1}^{k} W_i$.

2. **Random Sampling:** Draw a uniformly random vector $x \in \mathbb{F}^n$ to act as the candidate generator for the next cyclic block, W_{k+1}.

3. **Krylov Sequence:** Generate the Krylov sequence $x, Ax, A^2x, \ldots$ and incrementally reduce each new vector against the existing basis $\mathcal{B}$ using Gaussian elimination. This process halts when a linear dependence is found, yielding the minimal annihilating polynomial $C(z)$ of x modulo the previously resolved space W.

4. **Orthogonalization:** The linear dependence indicates that $C(A)x \in W$. To ensure the newly found cyclic subspace forms a direct summand, x must be updated to $\tilde{x} = x - u$ (for some $u \in W$) such that $C(A)\tilde{x} = 0$. The algorithm extracts the necessary shift by performing a polynomial division over the concatenated dependency arrays, and backtracks to apply it to x. It then recomputes the Krylov sequence.

5. **Iteration:** The vectors $\tilde{x}, A\tilde{x}, \ldots, A^{\deg(C)-1}\tilde{x}$ are added to $\mathcal{B}$. This repeats until $\dim(W) = n$.

2.2.2 Correctness

Theorem 2.3 (Divisibility of the Residuals)**.** *Let $W = \bigoplus_{i=1}^{k} W_i$ be the resolved space, where each $W_i = \langle w_i \rangle_A$ is a cyclic submodule with minimal polynomial $P_i(z)$. Let x be the random candidate vector, and $C(z)$ be its minimal polynomial modulo W. The Krylov reduction yields a dependency of the form:*

$$C(A)x = \sum_{i=1}^{k} R^{(i)}(A)w_i$$

Then the polynomial $C(z)$ divides $R^{(i)}(z)$ in $\mathbb{F}[z]$ for all $1 \leq i \leq k$.

Proof. As shown in the next section, with good probability, the minimal polynomial of the newly sampled block must divide the minimal polynomials of all previously extracted blocks: $C(z) \mid P_i(z)$ for all $i \leq k$.

Because the true vector space decomposes into a direct sum of cyclic submodules, we can uniquely decompose our random candidate as $x = \tilde{x} + \sum_{i=1}^{k} v_i$, where $\tilde{x}$ belongs to the target complementary subspace (meaning it is exactly annihilated by $C(A)$) and $v_i \in W_i$.

Applying $C(A)$ to this decomposition yields $C(A)x = C(A)\tilde{x} + \sum_{i=1}^{k} C(A)v_i = \sum_{i=1}^{k} C(A)v_i$. Since each $v_i \in \langle w_i \rangle_A$, there exist polynomials $S^{(i)}(z)$ such that $v_i = S^{(i)}(A)w_i$. Thus:

$$C(A)x = \sum_{i=1}^{k} C(A)S^{(i)}(A)w_i$$

Equating this with the relationship obtained from our Krylov reduction, we have:

$$\sum_{i=1}^{k} \big(C(A)S^{(i)}(A) - R^{(i)}(A)\big)w_i = 0$$

Because W is a direct sum, the components must independently evaluate to zero: $\big(C(A)S^{(i)}(A) - R^{(i)}(A)\big)w_i = 0$. This implies that the minimal polynomial of w_i, which is $P_i(z)$, must divide $C(z)S^{(i)}(z) - R^{(i)}(z)$.

Since we previously established that $C(z) \mid P_i(z)$, it follows by transitivity that $C(z)$ must divide $C(z)S^{(i)}(z) - R^{(i)}(z)$. Consequently, $C(z)$ must divide the residual polynomial $R^{(i)}(z)$ in $\mathbb{F}[z]$. □

2.2.3 Failure Probability

Theorem 2.4. *The probability that the randomized Krylov algorithm fails to extract the Frobenius form of $A \in \mathbb{F}^{n \times n}$ is bounded by $\frac{n}{|\mathbb{F}|}$.*

Proof. The algorithm fails if and only if, at some iteration, the randomly drawn vector x fails to generate the correct cyclic subspace in the quotient V/W. Let $P(z)$ be the true invariant factor of the remaining space. The vector x fails if its minimal polynomial modulo W is a strict divisor of $P(z)$.

We view V/W as a finitely generated torsion module over the principal ideal domain $\mathbb{F}[z]$, where z acts as A. The module decomposes into primary cyclic submodules:

$$V/W \cong \bigoplus_{j=1}^{\ell} \mathbb{F}[z]/\langle Q_j(z)^{e_j} \rangle$$

where $Q_j(z)$ are the distinct monic irreducible factors of $P(z)$, and e_j are their multiplicities.

A random vector $x \in V/W$ fails to have $P(z)$ as its minimal polynomial if and only if it is annihilated by $P(z)/Q_j(z)$ for at least one irreducible factor Q_j. This occurs exactly when the projection of x onto the primary component $\mathbb{F}[z]/\langle Q_j(z)^{e_j} \rangle$ is a multiple of $Q_j(z)$.

Because x is drawn uniformly at random, its projection onto any component is uniformly distributed. The submodule consisting of multiples of $Q_j(z)$ has a vector space codimension equal to $\deg(Q_j)$. Therefore, the probability that x falls into this "bad" submodule is $|\mathbb{F}|^{-\deg(Q_j)}$.

Applying the union bound over all distinct irreducible factors $Q_j \mid P(z)$, the probability that x fails in a given step is bounded by:

$$\mathbb{P}(\text{Failure at step}) \leq \sum_{Q_j | P} \frac{1}{|\mathbb{F}|^{\deg(Q_j)}}$$

Because $\sum \deg(P_i) = n$, the worst-case upper bound occurs when the characteristic polynomial splits entirely into n distinct linear factors. In this scenario:

$$\mathbb{P}(\text{Total Failure}) \leq \frac{n}{|\mathbb{F}|}$$

□

2.2.4 Polynomial Matrix Composition and Powering

With the Frobenius form $A = T^{-1}FT$ explicitly constructed, we can evaluate any polynomial function $f(A)$, such as matrix powers A^k.

Module Isomorphism on Cyclic Blocks Let $W_i = \langle w_i \rangle_A$ be a cyclic subspace of dimension d_i with minimal polynomial $P_i(z)$. The subspace is spanned by the Krylov basis $\mathcal{B}_i = \{w_i, Aw_i, \ldots, A^{d_i-1}w_i\}$.

There is a natural $\mathbb{F}[z]$-module isomorphism between W_i and the polynomial quotient ring $\mathbb{F}[z]/\langle P_i(z) \rangle$, given by the mapping:

$$A^j w_i \longleftrightarrow z^j \pmod{P_i(z)}$$

Under this isomorphism, applying the matrix function $f(A)$ to a basis vector $A^j w_i$ corresponds simply to polynomial multiplication in the quotient ring:

$$f(A)\big(A^j w_i\big) = A^j f(A) w_i \longleftrightarrow z^j f(z) \pmod{P_i(z)}$$

Constructing the Blocks To construct the matrix block $S_i = f(C_i)$ representing the action of $f(A)$ on the subspace W_i, we need to find the coordinates of $f(A)(A^j w_i)$ in the basis $\mathcal{B}_i$. By the isomorphism above, these coordinates are the coefficients of the polynomial:

$$R_{i,j}(z) = z^j f(z) \pmod{P_i(z)}$$

1. First, computes the base polynomial $R_{i,0}(z) = f(z) \pmod{P_i(z)}$.
2. This polynomial forms the 0-th row (or column depending on preference) of the block S_i.
3. For each subsequent basis vector $j \in \{1, \ldots, d_i - 1\}$, the algorithm calculates the next polynomial by shifting the previous one and reducing modulo $P_i(z)$:

   $$R_{i,j}(z) = z \cdot R_{i,j-1}(z) \pmod{P_i(z)}$$

   This populates the $d_i \times d_i$ block S_i in $O(d_i^2)$ time.

Once all blocks S_i are constructed, they are assembled into the block-diagonal matrix $S = \text{diag}(S_1, \ldots, S_m)$, and the final matrix is recovered via the similarity transform $f(A) = T^{-1}ST$.

Theorem 2.5 (Complexity of Matrix Exponentiation)**.** *Using the Frobenius form, the matrix power A^k can be computed in $O(n^3 + n \log n \log k)$ time.*

Proof. We compute the Frobenius form $A = T^{-1}FT$ in $O(n^3)$ time. For each invariant factor $P_i(z)$, our target function is $f(z) = z^k$. We evaluate the base polynomial $R_{i,0}(z) = z^k \pmod{P_i(z)}$ using binary exponentiation in the polynomial ring $\mathbb{F}[z]/\langle P_i(z) \rangle$ or using specialized algorithms. Because $\deg(P_i) = d_i$, each polynomial multiplication takes $O(d_i \log d_i)$ with NTT. The total time to compute $z^k \pmod{P_i(z)}$ is $O(d_i^2 \log k)$.

Summing this cost over all blocks yields $\sum_i O(d_i \log d_i \log k) \leq O(n \log n \log k)$. Constructing the blocks via shifts takes $\sum_i O(d_i^2) = O(n^2)$ time. Finally, recovering the matrix via $T^{-1}ST$ requires two $O(n^3)$ matrix multiplications. □

2.3 Black Box Linear Algebra

In the following algorithms, we consider a matrix $A \in \mathbb{F}^{n \times n}$ given strictly as a black-box function $f(v) = Av$.

2.3.1 Minimal Polynomial

Algorithm Description To compute the minimal polynomial of A, the algorithm generates two random vectors $u, v \in \mathbb{F}^n$. It then computes the scalar sequence $s_k = u^T A^k v$ for $k = 0, \ldots, 2n$. Finally, it applies an algorithm to find the minimal linear recurrence of the sequence s_k, which corresponds to the minimal polynomial of A.

Proof of Correctness Let $A \in \mathbb{F}^{n \times n}$. We define three polynomials:

1. $m_A(x)$: The minimal polynomial of the matrix A.
2. $m_{A,v}(x)$: The minimal polynomial of A with respect to the vector v, which is the monic polynomial of least degree such that $m_{A,v}(A)v = 0$.
3. $m_{u,v}(x)$: The minimal polynomial of the scalar sequence $s_k = u^T A^k v$, which is the monic polynomial of least degree such that $\sum_{i=0}^{d} c_i s_{k+i} = 0$ for all k.

Divisibility:
Because $m_A(A) = 0$, it follows that $m_A(A)v = 0$, meaning $m_{A,v}(x)$ must divide $m_A(x)$. Similarly, since $m_{A,v}(A)v = 0$, left-multiplying by $u^T A^k$ gives $u^T A^k m_{A,v}(A)v = 0$. This means the sequence s_k satisfies the recurrence defined by $m_{A,v}(x)$, so $m_{u,v}(x)$ must divide $m_{A,v}(x)$. This establishes the chain of divisibility:

$$m_{u,v}(x) \mid m_{A,v}(x) \mid m_A(x)$$

To prove $m_{u,v}(x) = m_A(x)$ with high probability, we bound the probability of a strict division occurring at either step.

Part 1: The Probability that $m_{A,v}(x) = m_A(x)$
Let $m_A(x)$ be factored into its distinct irreducible monic polynomials over $\mathbb{F}$:

$$m_A(x) = p_1(x)^{e_1} p_2(x)^{e_2} \cdots p_r(x)^{e_r}$$

where $r \leq n$. For $m_{A,v}(x)$ to be a proper divisor of $m_A(x)$, it must divide the polynomial $q_i(x) = m_A(x)/p_i(x)$ for at least one index $i \in \{1, \ldots, r\}$. This happens if and only if $q_i(A)v = 0$, which means v resides in the nullspace of the matrix $q_i(A)$.

Let $W_i = \ker(q_i(A))$. Because $m_A(x)$ is strictly the minimal polynomial of A, the matrix $q_i(A)$ cannot be the zero matrix. Therefore, W_i is a proper subspace of $\mathbb{F}^n$, meaning $\dim(W_i) \leq n - 1$. If v is chosen uniformly at random from $\mathbb{F}^n$, the probability that v falls into a specific proper subspace W_i is at most $1/|\mathbb{F}|$. By the union bound over all r irreducible factors, the probability that v falls into any of these kernels is:

$$P(m_{A,v}(x) \neq m_A(x)) = P\left(v \in \bigcup_{i=1}^{r} W_i\right) \leq \frac{r}{|\mathbb{F}|} \leq \frac{n}{|\mathbb{F}|}$$

Part 2: The Probability that $m_{u,v}(x) = m_{A,v}(x)$
Assume we have chosen a "good" vector v such that $m_{A,v}(x) = m_A(x)$. We now pick $u \in \mathbb{F}^n$ uniformly at random. By the same logic, for $m_{u,v}(x)$ to be a proper divisor of $m_A(x)$, it must divide $q_i(x) = m_A(x)/p_i(x)$ for some index i. This happens if and only if the sequence generated by $q_i(x)$ evaluates to zero for all k:

$$u^T A^k q_i(A)v = 0 \quad \text{for all } k \geq 0$$

Let $w_i = q_i(A)v$. Because we assumed $m_{A,v}(x) = m_A(x)$, we know that $w_i \neq 0$. The condition $u^T A^k w_i = 0$ for all k means that the vector u must be orthogonal to the Krylov subspace $\mathcal{K} = \text{span}\{w_i, Aw_i, A^2 w_i, \ldots\}$.

Since $w_i \neq 0$, the subspace $\mathcal{K}$ has a dimension of at least 1. Therefore, its orthogonal complement $\mathcal{K}^\perp$ is a proper subspace of $\mathbb{F}^n$ with dimension at most $n - 1$. The probability that a uniformly random u falls into $\mathcal{K}^\perp$ for a specific index i is at most $1/|\mathbb{F}|$. Applying the union bound again across the r factors:

$$P(m_{u,v}(x) \neq m_A(x) \mid v \text{ is good}) \leq \frac{r}{|\mathbb{F}|} \leq \frac{n}{|\mathbb{F}|}$$

Conclusion
Combining the two failure probabilities using the union bound, the total probability that the algorithm fails to find the true minimal polynomial is bounded by:

$$P(\text{failure}) \leq \frac{2n}{|\mathbb{F}|}$$

Therefore, for a sufficiently large field (where $|\mathbb{F}| \gg 2n$), the minimal recurrence of the scalar sequence s_k is exactly $m_A(x)$ with high probability.

2.3.2 Determinant

Description To evaluate $\det(A)$ for a matrix $A \in \mathbb{F}^{n \times n}$, the algorithm generates a random diagonal matrix D (represented by its diagonal entries $c_1, \ldots, c_n$) and a random vector $v \in \mathbb{F}^n$. It defines the linear map $A' = AD$ and computes its minimal polynomial $m_{AD}(x)$ using a minimal recurrence algorithm on the Krylov sequence $v, A'v, (A')^2 v, \ldots$. If the degree of $m_{AD}(x)$ is n, the determinant of A is extracted from the polynomial's constant term.

Proof of Correctness The minimal polynomial $m_{AD}(x)$ always divides the characteristic polynomial $p_{AD}(x) = \det(xI - AD)$. If A is singular, AD is singular, and $m_{AD}(x)$ will trivially have a constant term of 0, yielding a correct determinant of 0. Therefore, we assume A is non-singular. If we can prove that AD generically has distinct eigenvalues, then with high probability, its minimal polynomial will have degree n, allowing us to safely assume $m_{AD}(x) = p_{AD}(x)$.

Lemma 2.6 (Krylov Subspace Dimension)**.** *A matrix $M \in \mathbb{F}^{n\times n}$ is non-derogatory (i.e., its minimal polynomial has degree n) if and only if there exists a vector $y \in \mathbb{F}^n$ such that the Krylov matrix*

$$K_M(y) = \begin{bmatrix} y & My & M^2y & \dots & M^{n-1}y \end{bmatrix}$$

is non-singular.

Proof. By the rational canonical form, the vector space decomposes into cyclic subspaces governed by the invariant factors of M. M is non-derogatory if and only if its largest invariant factor (the minimal polynomial) has degree n. If so, summing the generators of the coprime cyclic subspaces yields a vector y that generates an n-dimensional cyclic subspace, making $K_M(y)$ full rank. Conversely, if $K_M(y)$ is full rank, y generates an n-dimensional cyclic subspace, forcing $\deg(m_M) = n$. □

Let $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_n)$ be formal variables, and let $D_{\mathbf{x}} = \operatorname{diag}(x_1, \dots, x_n)$. We construct the formal Krylov matrix $K_{AD_{\mathbf{x}}}(\mathbf{y})$, whose determinant $F(\mathbf{x}, \mathbf{y}) = \det(K_{AD_{\mathbf{x}}}(\mathbf{y}))$ is a multivariate polynomial.

Lemma 2.7 (Generic Distinct Eigenvalues)**.** *If A is non-singular, the formal polynomial $F(\mathbf{x}, \mathbf{y})$ is not identically zero.*

Proof. Let $B = A^{-1}$. Since $\det(A)$ is a non-zero constant, the roots of $\det(\lambda I - AD_{\mathbf{x}}) = 0$ are distinct if and only if the roots of $Q(\lambda, \mathbf{x}) = \det(\lambda B - D_{\mathbf{x}})$ are distinct. We proceed by induction on n.

Base Case ($n = 1$): $Q(\lambda, x_1) = \lambda B_{11} - x_1$. Since A is invertible, $B_{11} \neq 0$, yielding exactly one root. The discriminant is trivially non-zero.

Inductive Step: Assume the claim holds for $n - 1$. Because B is non-singular, we may permute the coordinates such that the upper-left $(n-1) \times (n-1)$ submatrix, B_{n-1}, is invertible. Expanding $Q(\lambda, \mathbf{x})$ along the last diagonal entry yields:

$$Q(\lambda, \mathbf{x}) = Q_0(\lambda, x_1, \dots, x_{n-1}) - x_n Q_1(\lambda, x_1, \dots, x_{n-1})$$

where $Q_1 = \det(\lambda B_{n-1} - D_{n-1})$. By the inductive hypothesis, there exists an assignment $\mathbf{c} \in \mathbb{F}^{n-1}$ such that $Q_1(\lambda, \mathbf{c})$ has $n - 1$ distinct, non-zero roots. Fixing these values, we study the two-variable polynomial $Q(\lambda, x_n) = Q_0(\lambda) - x_n Q_1(\lambda)$.

We construct the reverse polynomial. Let $w = 1/\lambda$, and define:

$$R(w, x_n) = w^n Q(1/w, x_n) = R_0(w) - x_n w R_1(w)$$

Substituting $z = 1/x_n$ and multiplying by z yields:

$$H(w, z) = zR(w, 1/z) = zR_0(w) - wR_1(w)$$

Evaluating at $z = 0$ leaves $H(w, 0) = -wR_1(w)$. Because $Q_1(\lambda)$ has $n-1$ distinct non-zero roots, its reverse polynomial $R_1(w)$ has $n - 1$ distinct finite roots, none of which are zero. Therefore, $H(w, 0)$ has exactly n distinct roots.

Thus, there exists an assignment for $\mathbf{x}$ yielding n distinct roots, meaning the generic $AD_{\mathbf{x}}$ is non-derogatory and $F(\mathbf{x}, \mathbf{y}) \not\equiv 0$. □

Lemma 2.8 (Failure Probability)**.** *For random assignments from a finite field $\mathbb{F}$, the probability that the algorithm fails is bounded by $\frac{n(n+1)}{2|\mathbb{F}|}$.*

Proof. To bound the probability of failure, we calculate the total degree of $F(\mathbf{x}, \mathbf{y})$. The k-th column of the Krylov matrix (for $k = 0, \dots, n-1$) is $(AD_{\mathbf{x}})^k \mathbf{y}$. The entries in this column are homogeneous polynomials of degree k in $\mathbf{x}$ and degree 1 in $\mathbf{y}$. Because the determinant is a sum of products containing exactly one entry from each column, the total degree of F is:

$$\deg(F) = \sum_{k=0}^{n-1} k + \sum_{k=0}^{n-1} 1 = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$$

By the Schwartz-Zippel Lemma, if we draw the entries for D and v uniformly at random from a finite field $\mathbb{F}$, the probability that they form a root of F is bounded by:

$$\mathbb{P}(F(\mathbf{c}, \mathbf{v}) = 0) \leq \frac{n(n+1)}{2|\mathbb{F}|}$$

Finally, the algorithm extracts the constant term c_0 of the minimal polynomial. Since $p_{AD}(0) = \det(-AD) = (-1)^n \det(A) \det(D)$, we correctly compute the determinant as:

$$\det(A) = \frac{(-1)^n c_0}{\prod_{i=1}^{n} c_i}$$

□

2.3.3 Wiedemann's Algorithm

Description Wiedemann's algorithm solves the linear system $Ax = b$ for x. It generates a random vector u and computes the scalar sequence $s_k = u^T A^k b$. It finds the minimal polynomial of this sequence, $p(x) = \sum_{i=0}^{d} p_i x^i$. Assuming A is non-singular, $p_0 \neq 0$. The algorithm then computes the solution vector using Horner's method: $x = -\frac{1}{p_0} \sum_{i=1}^{d} p_i A^{i-1} b$.

Proof of Correctness Let $m_{A,b}(x)$ be the minimal polynomial of A with respect to the vector b, meaning it is the polynomial of minimal degree such that $m_{A,b}(A)b = 0$. With high probability, the minimal polynomial $p(x)$ of the sequence $u^T A^k b$ (found via linear recurrence finding) is exactly $m_{A,b}(x)$.

By definition, $p(A)b = 0$. Expanding this polynomial expression yields:

$$\sum_{i=0}^{d} p_i A^i b = 0$$

Because A is non-singular, the constant term of its minimal polynomial is non-zero, so $p_0 \neq 0$. We can isolate the $i = 0$ term:

$$p_0 b + \sum_{i=1}^{d} p_i A^i b = 0$$

$$p_0 b + A\left(\sum_{i=1}^{d} p_i A^{i-1} b\right) = 0$$

Rearranging to solve for b, we get:

$$A\left[-\frac{1}{p_0} \sum_{i=1}^{d} p_i A^{i-1} b\right] = b$$

Since $Ax = b$, the unique solution x is the term inside the brackets. To compute x efficiently without storing $O(d)$ dense vectors, the algorithm uses Horner's method, evaluating the sum iteratively from $i = d$ down to 1 via $R \leftarrow A(R) + p_i b$, initializing with $R = p_d b$. After d iterations, R equals the summation, giving the correct solution x using only $O(d)$ matrix-vector multiplications and $O(n)$ space.

2.4 Useful Facts

Lemma 2.9 (Cauchy-Binet Formula)**.** *Let A be an $m \times n$ matrix and B be an $n \times m$ matrix (with $m \leq n$). Then*

$$\det(AB) = \sum_{S \in \binom{[n]}{m}} \det(A_{[m],S}) \det(B_{S,[m]})$$

where $\binom{[n]}{m}$ denotes the set of all m-element subsets of $\{1, \ldots, n\}$.

Proof. **Fact 1:** For any $n \times n$ matrix X and any integer $1 \leq k \leq n$, the coefficient of z^{n-k} in the polynomial $\det(zI_n + X)$ is exactly the sum of all $k \times k$ principal minors of X.

Fact 2: For any $m \times n$ matrix A and $n \times m$ matrix B we have from Lemma 8.2:

$$\det(zI_n + BA) = z^{n-m} \det(zI_m + AB)$$

Applying Fact 1 with $k = m$, the coefficient of z^{n-m} in $\det(zI_n + BA)$ is the sum of the $m \times m$ principal minors of BA. A principal minor of BA corresponding to an index subset $S \subseteq [n]$ of size m is the determinant of the submatrix $(BA)_{S,S}$, and $(BA)_{S,S} = B_{S,[m]} A_{[m],S}$.

Because $B_{S,[m]}$ and $A_{[m],S}$ are both square:

$$\det((BA)_{S,S}) = \det(B_{S,[m]} A_{[m],S}) = \det(B_{S,[m]}) \det(A_{[m],S})$$

The coefficient of z^{n-m} on the left-hand side is:

$$\sum_{S \in \binom{[n]}{m}} \det(B_{S,[m]}) \det(A_{[m],S})$$

On the right-hand side, the coefficient $[z^{n-m}]$ is $[x^0] \det(zI_m + AB) = \det(AB)$. □

Lemma 2.10.

$$\det'(I) = \mathrm{Tr}$$

Proof. We have:

$$\det{}'(I)(T) = \nabla_T \det(I) = \lim_{\varepsilon \to 0} \frac{\det(I + \varepsilon T) - \det(I)}{\varepsilon}$$

$\det(I + \varepsilon T)$ is a polynomial in ε of order n. The constant term is 1, while the linear term is $\operatorname{Tr}(T)$. □

Lemma 2.11. *For an invertible matrix A, we have:*

$$\det{}'(A)(T) = \det(A) \operatorname{Tr}(A^{-1}T)$$

Proof. Consider the following function of X:

$$\det(X) = \det(AA^{-1}X) = \det(A) \det(A^{-1}X)$$

Using Lemma 2.10:

$$\det{}'(A)(T) = \det(A) \det{}'(I)(A^{-1}T) = \det(A) \operatorname{Tr}(A^{-1}T)$$

□

Lemma 2.12 (Jacobi's Formula)**.**

$$\frac{d}{dt} \det A = \operatorname{Tr}\left(\operatorname{adj}(A) \frac{dA}{dt}\right)$$

Proof. Let $A(t)$ be a differentiable matrix function. We introduce a variable x and consider $A_x(t) = A(t) - xI$, $A_x(t)$ is invertible for infinitely many values of x.

For all x where $A_x(t)$ is invertible, we can apply Lemma 2.11 with $T = \frac{dA_x}{dt} = \frac{dA}{dt}$:

$$\frac{d}{dt} \det(A_x) = \det(A_x) \operatorname{Tr}\left(A_x^{-1} \frac{dA}{dt}\right)$$

Using $\det(A_x)A_x^{-1} = \operatorname{adj}(A_x)$, this becomes:

$$\frac{d}{dt} \det(A(t) - xI) = \operatorname{Tr}\left(\operatorname{adj}(A(t) - xI) \frac{dA}{dt}\right)$$

Since they agree for infinitely many x, they agree as polynomials, substituting $x = 0$ we have:

$$\frac{d}{dt} \det A = \operatorname{Tr}\left(\operatorname{adj}(A) \frac{dA}{dt}\right)$$

□

Lemma 2.13. *Let A be an $n \times n$ matrix. For any subset of indices $S \subseteq \{1, 2, \ldots, n\}$, let A_S denote the $(n - |S|) \times (n - |S|)$ principal submatrix. Then:*

$$\frac{p_A^{(k)}(x)}{k!} = \sum_{|S|=k} p_{A_S}(x)$$

Proof. We evaluate $p_A(x + y)$ in two different ways.

We can consider the Taylor expansion of $p_A(x + y)$ around x:

$$p_A(x + y) = \sum_{k=0}^{n} \frac{p_A^{(k)}(x)}{k!} y^k$$

Second:

$$p_A(x + y) = \det\big((x + y)I - A\big) = \det\big(yI + (xI - A)\big)$$

The coefficient y here is the sum of the determinant of all $(n - k) \times (n - k)$ sized minors $\det(xI_{n-k} - A_S) = p_{A_S}(x)$.

Equating the coefficients of y^k yields:

$$\frac{p_A^{(k)}(x)}{k!} = \sum_{|S|=k} p_{A_S}(x)$$

□

2.5 Majorization

Theorem 2.14 (Hardy-Littlewood-Pólya)**.** *Let $x, y \in \mathbb{R}^n$. If $x \prec_w y$, then there exists a doubly stochastic matrix M such that $x \leq My$. If $x \prec y$ then $x = My$.*

Proof. Let $\mathcal{M}$ be the compact, convex set of $n \times n$ doubly stochastic matrices, and let $\mathcal{V} = [0,1]^n$, which is also compact and convex. Consider the bilinear function $f(M, v) = v^T(My - x)$.

For any fixed $v \in \mathcal{V}$, let $P \in \mathcal{M}$ be a permutation matrix that orders the components of y comonotonically with v. Because $x \prec y$, an application of summation by parts yields $v^T x \leq v^T Py$. Consequently, we can always choose $M = P$ to see that $\max_{M\in\mathcal{M}} f(M, v) \geq f(P, v) \geq 0$.
By Sion's Minimax Theorem:

$$\max_{M\in\mathcal{M}} \min_{v\in\mathcal{V}} v^T(My - x) = \min_{v\in\mathcal{V}} \max_{M\in\mathcal{M}} v^T(My - x) \geq 0$$

This guarantees the existence of some $M_0 \in \mathcal{M}$ such that $v^T(M_0 y - x) \geq 0$ for all possible vectors $v \in \mathcal{V}$.

Applying this inequality with the standard basis vectors $v = e_i \in \mathcal{V}$ implies that $M_0 y \geq x$ component-wise. However, because M_0 is doubly stochastic, the sum of the components of $M_0 y$ exactly equals the sum of the components of y. By the definition of majorization, $\sum y_i = \sum x_i$ if $x \prec y$, thus $M_0 y = x$. □

Theorem 2.15 (Majorization Inequality)**.** *Let $x, y \in \mathbb{R}^n$ and $f : \mathbb{R} \to \mathbb{R}$ be convex. If $x \prec y$, or if $x \prec_w y$ and f is monotonically non-decreasing, then*

$$\sum_{i=1}^{n} f(x_i) \leq \sum_{i=1}^{n} f(y_i)$$

Proof. Because $x \prec_w y$ in both cases, Theorem 2.14 provides a doubly stochastic M such that $x \leq My$. For exact majorization ($x \prec y$), the vectors have equal sums ($\sum x_i = \sum y_i = \sum (My)_i$), which forces the equality $x = My$.

Therefore, the inequality $f(x_i) \leq f((My)_i)$ holds trivially for exact majorization, and it holds for weak majorization because f is non-decreasing.

Because the rows of M sum to 1, the term $(My)_i$ is a convex combination. Applying Jensen's inequality yields:

$$f(x_i) \leq f\left(\sum_{j=1}^{n} M_{ij} y_j\right) \leq \sum_{j=1}^{n} M_{ij} f(y_j)$$

Summing over all i we have:

$$\sum_{i=1}^{n} f(x_i) \leq \sum_{j=1}^{n} f(y_j) \left(\sum_{i=1}^{n} M_{ij}\right) = \sum_{j=1}^{n} f(y_j)$$

□

2.6 Perron-Frobenius Theorem

The Perron-Frobenius theorem establishes the spectral properties of non-negative matrices. A matrix $A \geq 0$ is irreducible if its directed graph is strongly connected; for any i, j, there exists an integer $k > 0$ such that $(A^k)_{ij} > 0$.

Theorem 2.16 (Perron-Frobenius for Irreducible Matrices)**.** *Let A be an $n \times n$ non-negative, irreducible matrix. Then:*

1. *A has a real, strictly positive eigenvalue $r > 0$ (the Perron root).*
2. *There exists a strictly positive eigenvector $v > 0$ corresponding to r.*
3. *For any other eigenvalue $\lambda \in \mathbb{C}$ of A, $|\lambda| \leq r$.*
4. *The eigenvalue r is algebraically (and geometrically) simple.*

Proof. Let $\Delta = \{x \in \mathbb{R}^n : x \geq 0, \sum_{i=1}^n x_i = 1\}$ be the simplex. For any non-zero $x \geq 0$, define the function:

$$r(x) = \max\{\rho \geq 0 \mid Ax \geq \rho x\} = \min_{x_i > 0} \frac{(Ax)_i}{x_i}$$

Let $r = \sup_{x\in\Delta} r(x)$.

Because A is irreducible, the matrix $B = (I + A)^{n-1}$ is positive ($B > 0$). Furthermore, B commutes with A. For any $x \in \Delta$, if $Ax \geq \rho x$, then $A(Bx) = B(Ax) \geq \rho(Bx)$, which implies $r(Bx) \geq r(x)$. The map $x \mapsto Bx$ sends Δ into $\mathbb{R}^n_{>0}$. In this interior, $r(x)$ is continuous. Since the image $B(\Delta)$ is compact and contained in the interior, the supremum r is finite and achieved at some positive vector $v = Bx > 0$. Thus, $Av \geq rv$.

$Av = rv$. Suppose for contradiction that $Av \geq rv$ but $Av \neq rv$. This means the vector $z = Av - rv$ is non-negative and non-zero ($z \gneq 0$). Because $B > 0$, multiplying by B yields a positive vector: $Bz > 0$. Thus, $B(Av - rv) > 0 \implies A(Bv) > r(Bv)$. Since strict inequality holds for all coordinates of Bv, we can find some small

$\epsilon > 0$ such that $A(Bv) \geq (r+\epsilon)(Bv)$. This implies $r(Bv) \geq r+\epsilon$, which contradicts the maximality of r. Therefore, we must have $Av = rv$. Since $v > 0$, r must be strictly positive (as $A \neq 0$).

Strict positivity of the eigenvector. Suppose $u \geq 0$ is any non-zero eigenvector for r, meaning $Au = ru$. Then $Bu = (1+r)^{n-1}u$. Since $u \gneq 0$ and $B > 0$, we have $Bu > 0$, forcing $(1+r)^{n-1}u > 0$, which means u must be strictly positive.

Maximality of r. Let λ be any eigenvalue of A with eigenvector $y \in \mathbb{C}^n$, so $Ay = \lambda y$. Taking absolute values component-wise, and using the triangle inequality alongside the non-negativity of A, we get:

$$|\lambda||y| = |\lambda y| = |Ay| \leq A|y|$$

This implies that the non-negative, non-zero vector $|y|$ satisfies $A|y| \geq |\lambda||y|$. By the definition of our function, $r(|y|) \geq |\lambda|$. Since r is the supremum over all such vectors, $r \geq |\lambda|$.

Simplicity of r. By applying the exact same argument to A^T (which is also irreducible), there exists a strictly positive left eigenvector $w^T > 0$ such that $w^T A = rw^T$.

To show the geometric multiplicity is 1, suppose v and y are two linearly independent eigenvectors for r. Since $v > 0$, we can choose a real scalar c such that $z = v - cy$ has at least one coordinate equal to zero, but z is not the zero vector. However, $Az = rz$. Taking absolute values, $A|z| \geq r|z|$. This forces $A|z| = r|z|$, making $|z|$ a non-negative eigenvector for r. Then $|z|$ must be strictly positive, contradicting that it has a zero coordinate. Thus, the geometric multiplicity is 1.

To prove the algebraic multiplicity is 1, we utilize the strictly positive left eigenvector w^T. If the algebraic multiplicity were strictly greater than 1, there would exist a generalized eigenvector $g \in \ker(A - rI)^2$ such that $(A - rI)g = v$. Multiplying both sides on the left by w^T yields:

$$w^T(A - rI)g = w^T v$$

Because w^T is a left eigenvector associated with r, we have $w^T A = rw^T$, meaning $w^T(A - rI) = 0$. Thus, the left side is 0. However, because both $w > 0$ and $v > 0$, their inner product is strictly positive ($w^T v > 0$). Therefore, r must be an algebraically simple root. □

3 Graphs

3.1 Flow

3.1.1 Highest Label Push-Relabel

We analyze the highest-level selection rule for the preflow-push algorithm.

Let $K = \sqrt{m}$. For any node v, define $d'(v) = |\{w : d(w) \leq d(v)\}|$. Thus $d'(v)$ is the number of nodes with a distance label no greater than v's label. We define the potential function Φ as:

$$\Phi = \sum_{v \text{ active}} \frac{d'(v)}{K}$$

Initially, $\Phi \leq n^2/K$, and $\Phi \geq 0$ throughout the execution. Let us examine how the operations affect Φ:

- **Relabeling:** A relabeling of a node v increases Φ by at most n/K, because $d'(v)$ may increase to at most n, while $d'(w)$ for all other nodes $w \neq v$ does not increase.
- **Saturating push:** This operation creates at most one new active node, increasing Φ by at most n/K.
- **Nonsaturating push:** A nonsaturating push from v to w deactivates v and activates w (if it is not already active). Since the push only happens on eligible edges where $d(w) < d(v)$, we know $d'(w) \leq d'(v)$, meaning Φ does not increase.

To bound the number of nonsaturating pushes, we divide the execution into phases. A phase consists of all pushes between two consecutive changes of the maximum active node label, $d^* = \max\{d(v) : v \text{ is active}\}$. We classify a phase as *cheap* if it contains at most K nonsaturating pushes, and *expensive* otherwise.

Claim 3.1. *The number of phases is at most $4n^2$, and the number of nonsaturating pushes in cheap phases is at most $4n^2K$.*

Proof. Initially, $d^* = 0$, and any increase in d^* is strictly due to a relabeling. The number of relabelings is bounded by $2n^2$. Thus, d^* increases at most $2n^2$ times and consequently decreases at most $2n^2$ times, limiting the total number of changes (and thus phases) to $4n^2$. The bound on cheap pushes follows immediately since each cheap phase has at most K pushes. □

Claim 3.2. *An expensive phase containing $Q > K$ nonsaturating pushes decreases Φ by at least Q.*

Proof. During an expensive phase, d^* remains constant, meaning all Q nonsaturating pushes originate from nodes at level d^*. The phase ends either when all nodes at level d^* are deactivated, or a node is relabeled to $d^* + 1$. Therefore, level d^* must contain $Q > K$ nodes (active or inactive) at all times during the phase. For any nonsaturating push along an edge (u, v), because there are at least K nodes at level $d(u)$, each push decreases Φ by at least one. □

The total increase of Φ is bounded by the sum of increases from relabelings and saturating pushes. There are at most $2n^2$ relabelings and $2nm$ saturating pushes, so the total increase is at most $(2n^2 + 2nm)(n/K)$. Because $\Phi \geq 0$, the total decrease is bounded by the total increase plus the initial Φ value (n^2/K).

This bounds the number of nonsaturating pushes in expensive phases by $(2n^3 + 2n^2m + n^2)/K \leq (3n^3 + 2n^2m)/K$. Adding the cheap pushes, the total number of nonsaturating pushes is at most $(3n^3 + 2n^2m)/K + 4n^2K$. By setting $K = \sqrt{m}$ and noting $n = O(m)$, this evaluates to $O(n^2\sqrt{m})$.

Theorem 3.3. *The running time of the preflow-push algorithm using the highest-level selection rule is $O(n^2\sqrt{m})$.*

Proof. Each push and relabel operation can be implemented in $O(1)$ time. The highest-level selection rule can be implemented using a bucket data structure where each bucket i holds all nodes w with $d(w) = i$. Maintaining this structure incurs a total running time overhead of $O(nm)$. Therefore, the algorithm's running time is dominated by the $O(n^2\sqrt{m})$ nonsaturating pushes, resulting in an overall time complexity of $O(n^2\sqrt{m})$. □

3.1.2 Incremental SCCs

Algorithm 1: Incremental SCCs

Input: Vertices V, chronological sequence of edges E
Output: List of edges M that strictly merge strongly connected components

Function `IncrementalSCC`(V, E)**:**
  $C_{final} \leftarrow$ `SCC`(V, E);
  $E_{valid} \leftarrow \langle (u, v) \in E \mid C_{final}[u] = C_{final}[v] \rangle$;
  $M \leftarrow \langle\rangle$;
  `Solve`(V, E_{valid}, M);
  return M;

Function `Solve`(V', E', M)**:**
  if $|E'| = 0$ **then**
    return
  if $|E'| = 1$ **then**
    $(u, v) \leftarrow E'[0]$;
    `// If edge endpoints are distinct in the contracted graph, it merges them`
    if $u \neq v$ **then** Append $E'[0]$ to M;
    return;
  `// Divide the sequence of edges in half`
  $mid \leftarrow \lfloor |E'|/2 \rfloor$;
  $E_{left} \leftarrow \langle E'[0], \ldots, E'[mid - 1] \rangle$;
  `// Evaluate the graph formed by only the first half of the edges`
  $C \leftarrow$ `SCC`(V', E_{left});
  `// C[v] gives the component ID of vertex v`
  `// Partition edges based on the intermediate components`
  $E_L \leftarrow \langle\rangle$; $E_R \leftarrow \langle\rangle$;
  for $i \leftarrow 0$ **to** $|E'| - 1$ **do**
    $(u, v) \leftarrow E'[i]$;
    if $C[u] = C[v]$ **then**
      `// Edge is internal to a component; it only affects the left half`
      if $i < mid$ **then** Append (u, v) to E_L;
    else
      `// Edge spans components; contract its endpoints for the right half`
      Append $(C[u], C[v])$ to E_R;
  `Solve`(V', E_L, M) ; `// Find merges that occur in the first half`
  $V_{contracted} \leftarrow \{C[v] \mid v \in V'\}$;
  `Solve`$(V_{contracted}, E_R, M)$; `// Find merges that occur in the second half`

3.1.3 Flow Rounding

Flow rounding converts a fractional circulation into an integral one without worsening the overall cost. We do this by eliminating fractional cycles.

Availability and Cycle Cancelling To maintain edge capacity constraints while pushing flow around cycles, the algorithm relies on the concept of availability. The availability of a directed edge (u, v) is the amount of flow that can be pushed before the flow reaches the next integer bound.

Link-Cut Trees To efficiently detect cycles and push flow, the algorithm maintains a forest of the current fractional edges using a Link-Cut Tree. It supports querying the minimum availability along a path, calculating path costs, and pushing flow along paths in $O(\log n)$ time.

Algorithm 2: Rounding Fractional Circulations

Input: Graph $G = (V, E)$, Original fractional circulation f
Output: Integral circulation f' with equal or lesser cost

```
Function RoundFlow(G, f):
    f' ← f;
    Initialize empty Link-Cut Tree forest G' = (V, E' = ∅);
    for each edge (u, v) ∈ E with fractional f'(u, v) do
        if FindRoot(u) ≠ FindRoot(v) then
            Link(u, v) ;                                   // Edge does not form a cycle
        else
            // A fractional cycle is formed by (u, v) and the v → u tree path
            P ← Path between v and u in G';
            Determine cycle direction (forward or reverse) that yields ≤ 0 cost;
            // Push flow until at least one edge becomes integral
            δ ← min(FindMin(u, v), available(v, u));
            PushFlow(u, v, δ) ;                            // Update flow along cycle
            // Remove all newly integral edges from the forest
            while (e_int ← FindIntegralEdge(u, v)) ≠ null do
                Cut(e_int);
            if f'(u, v) is still fractional then
                Link(u, v);
    return f';
```

Correctness

Theorem 3.4. *The algorithm correctly produces an integral circulation of no worse cost in $O(m \log n)$ time.*

Proof. By evaluating the cost of the path in the Link-Cut tree, we can always push flow around a newly formed cycle in a direction where the total cost is non-positive, ensuring the overall circulation cost never increases. At termination, every fractional edge has been evaluated. The remaining edges in the Link-Cut tree explicitly form a forest, meaning the final circulation f' must be integral. □

Proposition 3.5. *Let f be an integral circulation on a digraph D and d any positive integer. Then f can be written as the sum $g_1 + g_2 + \cdots + g_d$ of integral circulations such that for each index j and each edge e:*

$$\lfloor f(e)/d \rfloor \leq g_j(e) \leq \lceil f(e)/d \rceil$$

Proof Sketch. We proceed by induction on d. Round $\frac{f}{d}$ to some integral circulation g, then consider $f - g = g_2 + \cdots + g_d$ inductively, then the bounds we get for $g_2, \ldots, g_d$ imply also that they satisfy the needed bounds.

□

3.1.4 Arboricity

The k-disjoint forest partition algorithm determines whether the edges of a weighted multigraph G can be partitioned into k edge-disjoint forests. By binary searching over k, this algorithm can compute the exact arboricity of a graph. It operates in $O(n(n+m)^2 \log(kn))$ time using a series of max-flow computations.

Nash-Williams Theorem The Nash-Williams Theorem states that a multigraph can be partitioned into k edge-disjoint forests if and only if for every non-empty subset of vertices $U \subseteq V$, the sum of the weights of the edges induced by U satisfies:

$$c(E(U)) \leq k(|U| - 1)$$

Network Construction For a chosen "root" vertex $r \in V$, we construct a bipartite flow network to test all subsets U containing r. The network consists of a source s, a sink t, a layer of vertex-nodes representing V, and a layer of edge-nodes representing E.

1. **Source to Vertices:** A directed edge from s to every vertex $v \in V \setminus \{r\}$ with capacity k.
2. **Vertices to Edges:** For every edge $e = (u, v)$ in the original graph, we add directed edges from node u to node e, and from node v to node e, both with capacity ∞.
3. **Edges to Sink:** A directed edge from every edge-node e to t with capacity $c(e)$ (the weight of the edge).

Consider an arbitrary finite-capacity $s-t$ cut, partitioned into sets S and T, let $U = V \cap T$.

Because the directed edges from vertices to edge-nodes have infinite capacity, placing an edge-node in T while one of its endpoints remains in S would cut infinite capacity. Therefore, the only edge-nodes in T are in $E(U)$ and to minimize the cut capacity for a given U, we let exactly $E(U)$ be on the sink side.

The cut edges come from:

1. Edges from s to the vertices in $U \setminus \{r\}$.
2. Edges from $E \setminus E(U)$ to the sink t.

And the total capacity is:

$$\text{Cut Capacity} = k|U \setminus \{r\}| + c(E) - c(E(U))$$

For the maximum flow to satisfy the total edge weight $c(E)$, every cut must have a capacity of at least $c(E)$. Thus:

$$k|U \setminus \{r\}| + c(E) - c(E(U)) \geq c(E) \implies c(E(U)) \leq k|U \setminus \{r\}| = k(|U| - 1)$$

Algorithm 3: k-Disjoint Forest Partition

Input: Graph V, edges E with weights c_e, integer k
Output: True if E can be partitioned into $\leq k$ forests, False otherwise

```
Function IsKForestPartitionable(V, E, k):
    n ← |V|; m ← |E|;
    C_total ← Σ_{e∈E} c_e;
    if C_total > k(n − 1) then return False;
    for r ← 0 to n − 1 do
        Initialize flow network G_flow with source s, sink t;
        for v ∈ V do
            if v ≠ r then Add edge s → v with capacity k;
        for each edge e = (u, v) ∈ E do
            Add edge u → e with capacity ∞;
            Add edge v → e with capacity ∞;
            Add edge e → t with capacity c_e;
        flow ← DinitzScaling(G_flow, s, t);
        if flow < C_total then return False;
    return True;
```

Complexity

Theorem 3.6. *The algorithm correctly verifies the k-forest partition property in $O(n(n+m)^2 \log(kn))$ time.*

Proof. We use Dinitz's algorithm with capacity scaling, which finds a maximum flow in $O(VE \log U)$ time. For our network, the number of vertices is $V = n + m + 2$ and the number of edges is $E = n + 3m$. The maximum capacity connected to the source is $U \leq kn$. Thus we get a runtime of $O((n+m)^2 \log(kn))$. Because we execute this flow algorithm n times (once for each root r), the overall time complexity is $O(n(n+m)^2 \log(kn))$. □

3.2 Matching

3.2.1 Bipartite Regular Matching

Given a d-regular bipartite graph $G = (P, Q, E)$ with $|P| = |Q| = n$, a perfect matching can be found in $O(n \log n)$ expected time.

The algorithm iteratively augments a partial matching M by discovering augmenting paths via random walks. If M currently leaves k nodes unmatched in both P and Q, we implicitly construct a directed matching graph H:

1. Orient all edges of G from P to Q.
2. Contract each matched edge pair $(u, v) \in M$ into a single supernode.
3. Add a source vertex s connected by d parallel edges to each unmatched node in P, directed outward.
4. Add a sink vertex t connected by d parallel edges from each unmatched node in Q, directed inward.

Any $s \to t$ path in H corresponds directly to a valid augmenting path in G with respect to M. The algorithm simply performs a standard random walk starting from s, choosing an outgoing edge uniformly at random at each step, until it hits t.

Lemma 3.7 (Random Walk Hitting Time)**.** *Given a partial matching leaving k vertices unmatched, the expected number of steps before a random walk started at s reaches t is at most $2 + \frac{n}{k}$.*

Proof. Construct a modified graph H^* by identifying the source s and sink t into a single unified node s^*. In H^*, the out-degree of every vertex equals its in-degree, making it an Eulerian digraph.

The out-degree of the unified node s^* is kd, while the out-degree of every other node is $d - 1$. The expected time for a random walk in H^* starting at s^* to return to s^* is exactly the inverse of its stationary probability π_{s^*}. Because H^* is balanced, this stationary probability is simply the ratio of the degree of s^* to the sum of all degrees in H^*. Evaluating this yields an expected return time of:

$$\frac{1}{\pi_{s^*}} = \frac{n(d-1) + 2kd + kd}{kd} \leq 2 + \frac{n}{k}$$

Running a walk from s^* until it returns to s^* in H^* is mathematically identical to running a walk from s until it hits t in the original matching graph H. □

To construct a perfect matching, the algorithm starts with an empty matching ($k = n$) and repeatedly runs untruncated random walks to augment the matching n times until $k = 0$. By the lemma, the total expected number of steps taken across all random walks is bounded by:

$$\sum_{k=1}^{n} \left(2 + \frac{n}{k}\right) = 2n + nH_n = O(n \log n)$$

3.3 Shortest Paths

3.3.1 Eppstein's Algorithm

Eppstein's algorithm elegantly computes the k shortest paths from a source s to a destination t in a directed graph in $O((m + n) \log n + k \log k)$ time. Rather than searching for full paths from scratch, the algorithm characterizes every possible $s \to t$ path as the base shortest path plus a sequence of deviations, or sidetracks.

First, we compute the shortest path from every vertex u to t using a reverse Dijkstra's search. This yields the shortest path distance $d[u]$ for all vertices and implicitly defines a shortest path tree T rooted at t.

For any directed edge $e = (u, v)$ with weight w, its sidetrack cost represents the distance penalty incurred by taking e instead of following the shortest path tree T from u. It is defined as:

$$\delta(e) = w + d[v] - d[u] \geq 0$$

Any path P from s to t can be uniquely identified by the sequence of non-tree edges (sidetracks) it traverses: $e_1, e_2, \ldots, e_p$. The total length of P is exactly the base distance plus the sum of its sidetrack penalties:

$$\text{length}(P) = d[s] + \sum_{i=1}^{p} \delta(e_i)$$

State Space and Persistent Heaps To efficiently find the k lightest sidetrack sequences, we construct a min-heap $H[u]$ for every vertex u. This heap contains all possible sidetrack edges originating from u and all ancestors of u in T, ordered by $\delta(e)$.

Because $H[u]$ shares almost all its elements with the heap of its parent in T, we use a Persistent Skew Heap. This allows us to implicitly build the heaps for all vertices in $O(m \log n)$ time. Once the heaps are built, enumerating the k-shortest paths reduces to a Dijkstra-like search over the structure of these heaps, transitioning either by swapping to heavier sidetracks within the same heap or appending new sidetracks from a destination vertex.

Algorithm 4: Eppstein's k-Shortest Paths

Input: Graph $G = (V, E)$, source s, destination t, integer k
Output: Lengths of the k shortest $s \to t$ paths

Function `Eppstein`(G, s, t, k)**:**
  $d, T \leftarrow$ `ReverseDijkstra`(G, t);
  if $d[s] = \infty$ **then return** $\emptyset$;
  Order V by $d[u]$ ascending
  for $u \in V$ **do**
    $root \leftarrow$ heap root of u's parent in T (or null if none);
    for *each outgoing edge* $e = (u, v) \in E \setminus T$ **do**
      $\delta \leftarrow \text{weight}(e) + d[v] - d[u]$;
      $root \leftarrow$ `PersistentInsert`$(root, \delta, v)$;
    $H[u] \leftarrow root$;
  $Q \leftarrow$ empty min-priority queue;
  Q.push(length $= d[s]$, node $= H[s]$);
  $R \leftarrow$ empty list;
  while Q *is not empty* **and** $|R| < k$ **do**
    $(L, node) \leftarrow Q$.pop();
    Append L to R;
    if $node.left \neq$ null **then**
      Q.push$(L - node.\delta + node.left.\delta, node.left)$;
    if $node.right \neq$ null **then**
      Q.push$(L - node.\delta + node.right.\delta, node.right)$;
    $next\_heap \leftarrow H[node.dest]$;
    if $next\_heap \neq$ null **then**
      Q.push$(L + next_heap.\delta, next_heap)$;
  return R;

Complexity Analysis

Theorem 3.8. *Eppstein's algorithm correctly enumerates the k shortest paths in $O((m + n) \log n + k \log k)$ time.*

Proof. The base distances and shortest path tree are found in $O((m+n) \log n)$ time. Constructing the persistent heaps requires inserting at most m non-tree edges. Each persistent insertion takes $O(\log n)$ time, leading to $O(m \log n)$ time and space for the heap construction.

During the path extraction phase, every valid transition in the state space explores paths in non-decreasing order of total weight because the heaps are strictly min-ordered ($\delta(\text{child}) \geq \delta(\text{parent})$) and $\delta(e) \geq 0$. Since each state expands to at most 3 subsequent states, the priority queue size never exceeds $3k$. Performing k extractions from this queue takes $O(k \log k)$ time. Summing these phases yields the final bounds. □

4 Trees

4.1 LCA

1. Pre-Order Reduction Let $\texttt{dfs}[0 \dots n-1]$ be the sequence of nodes visited in a standard pre-order DFS traversal. Let $\texttt{pos}[u]$ be the pre-order index of node u, such that $\texttt{dfs}[\texttt{pos}[u]] = u$. Let $p[u]$ be the parent of node u, and let $\text{depth}[u]$ be its distance from the root.

We construct our $O(n)$ RMQ from Algorithm 12 over an implicit array Z of size n, defined by the depth of the parent of the node at each pre-order step:

$$Z[i] = \text{depth}[p[\texttt{dfs}[i]]]$$

2. Queries To find $LCA(a, b)$, we first ensure that a is visited before b by swapping them if $\texttt{pos}[a] > \texttt{pos}[b]$. If $a = b$, the LCA is simply a. Otherwise, we query the RMQ for the index m that minimizes Z in the range $[\texttt{pos}[a]+1, \texttt{pos}[b]]$. The LCA is exactly the parent of the node at that minimum index: $p[\texttt{dfs}[m]]$.

3. Correctness There are two cases:

- **Case 1 (a is an ancestor of b):** The pre-order interval $[\texttt{pos}[a] + 1, \texttt{pos}[b]]$ contains only proper descendants of a. The node in this interval with the shallowest parent is the direct child of a that lies on the path to b. The parent of this child is a.
- **Case 2 (a and b are in separate branches):** Let $L = LCA(a, b)$. After the DFS visits a and finishes its subtree, it backtracks to L and eventually descends into the branch containing b. Let v be the direct child of L that roots the subtree containing b. The node v is visited after a and before b, so $\texttt{pos}[v] \in [\texttt{pos}[a] + 1, \texttt{pos}[b]]$. Its parent is L. Because every other node visited in this interval is a descendant of L, their parents have a depth of at least $\text{depth}[L]$. Thus, the RMQ identifies v, and returns its parent $p[v] = L$.

Algorithm 5: LCA

```
1  Function InitLCA(n, p, dfs, pos, depth):
       // Define a closure to compare the parent depths of two pre-order indices
2      cmp ← Closure(i, j) returning depth[p[dfs[i]]] < depth[p[dfs[j]]];
       // Initialize the RMQ data structure using this closure
3      rmq ← RMQ_Init(n, cmp);
4      return LCA_Struct{p, dfs, pos, rmq};

5  Function QueryLCA(a, b):
6      if a = b then
7          return a;
       // Order by entry time
8      l ← min(pos[a], pos[b]);
9      r ← max(pos[a], pos[b]);
10     m ← rmq.query(l + 1, r);
11     return p[dfs[m]];
```

4.2 $O(n)$ Binary Lifting

Binary Lifting usually requires $O(n \log n)$ time and space to precompute an ancestor table, but using a modified method we can do it in $O(n)$ preprocessing and space.

1. Preprocessing We define $\texttt{jmp}[v]$ as a pointer to an ancestor of v. The idea is to grow the jump distances in powers of two as we ascend.

During a pre-order traversal, consider a node v with parent p. We look at the jump from the parent $p \to \texttt{jmp}[p]$, and the subsequent jump $\texttt{jmp}[p] \to \texttt{jmp}[\texttt{jmp}[p]]$. If these two jumps cover the exact same change in depth, we combine them. The new jump from v covers the edge (v, p), the first jump of length L, and the second jump of length L. Thus, the new jump pointer points to $\texttt{jmp}[\texttt{jmp}[p]]$ and covers a distance of $1 + 2L$.

If the two preceding jumps do not match in length, we set $\texttt{jmp}[v] = p$, resetting the jump length to 1.

2. Querying The algorithm has two phases:

1. **Equate Depths:** If u and v are at different depths, we identify the deeper node and use our `depth_jmp` function to lift it exactly to the depth of the shallower node, this works basically the same as the LCA function. If the nodes are now identical, we have found the LCA.

2. **Simultaneous Ascent:** If the nodes are distinct but at the same depth, we ascend them simultaneously. We evaluate the jump pointers $\texttt{jmp}[u]$ and $\texttt{jmp}[v]$:

   - If $\texttt{jmp}[u] \neq \texttt{jmp}[v]$, the jump lands strictly below the LCA. We can safely take this maximal jump: $u \leftarrow \texttt{jmp}[u]$ and $v \leftarrow \texttt{jmp}[v]$.
   - If $\texttt{jmp}[u] = \texttt{jmp}[v]$, the jump has either hit the LCA exactly or overshot it into a higher common ancestor. Because we cannot distinguish between these two cases, we take a single step up to the parents: $u \leftarrow p[u]$ and $v \leftarrow p[v]$.

We repeat this until $u = v$, at which point u is the LCA.

3. Complexity Let $L(u) = \text{depth}[u] - \text{depth}[\texttt{jmp}[u]]$. By construction, $L(u)$ is always of the form $2^k - 1$ for some integer $k \geq 1$. We call k the rank of the jump. A jump of rank k implicitly spans a tree of $2^k - 1$ edges.

Lemma 4.1 (Monotonicity of Jump Sequences)**.** *For any node u, let $J(u) = [L_1, L_2, \ldots, L_m]$ be the sequence of jump lengths encountered by repeatedly following the* `jmp` *pointers from u to the root. The sequence $J(u)$ is monotonically non-decreasing.*

Proof. The sequence of jumps from a node depends entirely on its depth d. Let J_d denote the sequence of jump lengths from a node at depth d.

By induction, all jump lengths are drawn from the set of skew-binary weights: $\mathcal{W} = \{2^k - 1 \mid k \geq 1\}$.

We will prove by induction on depth d that:

$$J_d \text{ satisfies } L_1 \leq L_2 < L_3 < \cdots < L_m.$$

Base Case ($d = 1$):
Trivial.

Inductive Step:
Assume J_d satisfies the invariant. We consider J_{d+1}:

- **Case A:** J_d does *not* start with identical elements.
  By the invariant, we have $L_1 < L_2 < L_3 < \ldots$
  The algorithm prepends a jump of 1, making $J_{d+1} = [1, L_1, L_2, \ldots]$.
  Because all elements are ≥ 1, we have the invariant.

- **Case B:** J_d starts with two identical elements.
  By the invariant, we have: $J_d = [L, L, L_3, L_4, \ldots]$ with $L < L_3 < L_4 \ldots$
  The algorithm merges the first two jumps, replacing them with $2L + 1$, making $J_{d+1} = [2L + 1, L_3, L_4, \ldots]$.
  Because $L \in \mathcal{W}$, the next available weight in $\mathcal{W}$ is $2L + 1$. Since $L_3 \in \mathcal{W}$ and $L_3 > L$, we have $L_3 \geq 2L + 1$.
  Therefore, the new sequence satisfies the invariant again.

Thus J_d is always monotonically non-decreasing for any depth d. □

By the lemma, $J(u)$ can contain at most two jumps of any length $2^k - 1$. Thus the entire sequence $J(u)$ consists of at most $2\lfloor \log_2 n \rfloor$ jumps.

During a query, we attempt to follow $J(u)$ to cover a target distance. Every operation falls into one of two categories:

1. Successful Jumps:
When we take $\texttt{jmp}[u]$, we consume one of the elements of $J(u)$, so there are at most $O(\log n)$ of these operations.

2. Penalty Steps:
If taking the jump L would overshoot our target we cannot consume it. Instead, we take a penalty step to the parent p. Because $L > 1$ was built by merging a step of 1 and two jumps of size $(L-1)/2$, stepping to the parent "unpacks" L. The new sequence from the parent, $J(p)$, replaces L with two smaller jumps: $[(L-1)/2, (L-1)/2, \ldots]$. We only unpack a jump of size L when it is greater than the remaining distance. Because the remaining distance strictly decreases, we will never need to unpack a jump of size L more than once across the query. Thus, the total number of penalty steps is bounded by the number of unique jump sizes, which is $\leq \log_2 n$.

Algorithm 6: Binary Lifting

```
Function BuildJumpTable(n, par, dfs, depth):
    jmp ← array of size n;
    for each v in dfs do
        p ← par[v];
        if v = p then
            jmp[v] ← v;
        else
            p_j ← jmp[p];
            p_jj ← jmp[p_j];
            if depth[p] − depth[p_j] = depth[p_j] − depth[p_jj] then
                jmp[v] ← p_jj;
            else
                jmp[v] ← p;
    return jmp;

Function QueryLCA(u, v, par, jmp, depth):
    if depth[u] > depth[v] then
        Swap u and v;
    v ← DepthJump(v, depth[u], par, jmp, depth);
    while u ≠ v do
        // We would overshot the LCA.
        if jmp[u] = jmp[v] then
            u ← par[u];
            v ← par[v];
        else
            u ← jmp[u];
            v ← jmp[v];
    return u;
```

4.3 Static Top Tree

4.3.1 Description

The Static Top Tree is a tree-decomposition data structure that allows for the efficient evaluation of path and subtree aggregates over a static tree. It achieves $O(\log n)$ depth by combining Heavy-Light Decomposition with weight-balanced binary trees.

The algorithm views the tree through two structures:

- **Path (Cluster):** A sequence of vertices. It conceptually has two boundary nodes.
- **Point (Cluster):** A subtree attached to a single boundary node.

The tree is decomposed into operations:

1. **Vertex(u):** Creates a Path consisting of a single vertex u.
2. **Compress(P_1, P_2):** Merges two adjacent Paths P_1 and P_2 along a heavy path into a single Path.
3. **Rake(Pt_1, Pt_2):** Merges two Points (light subtrees) sharing the same boundary node into a single Point.
4. **AddEdge(P):** Converts a Path P into a Point, effectively viewing the heavy path as a light child hanging off a parent vertex.
5. **AddVertex(Pt, u):** Attaches a Point Pt (the raked light children of u) to the vertex u, promoting it to a Path.

Construction The algorithm first roots the tree and computes subtree sizes. It identifies heavy edges (the child with the largest subtree size). This naturally partitions the tree into disjoint heavy paths.
For a given vertex u, the algorithm processes its light children recursively by converting them to Paths, using AddEdge to turn them into Points, and using a weight-balanced divide-and-conquer (Rake) to combine all light children into a single Point. This Point is then attached to u using AddVertex. Finally, the vertices along a heavy path are combined using a weight-balanced divide-and-conquer (Compress) into a single Path representing the entire heavy path.

4.3.2 Complexity

Theorem 4.2. *The Static Top Tree evaluates paths and updates single nodes in $O(\log n)$ time, requiring $O(n \log n)$ construction time and $O(n)$ space.*

Proof. **Depth Constraint ($O(\log n)$):** The time complexity for updates and queries depends strictly on the maximum depth of the constructed operation tree. A traversal from the root of the Static Top Tree to a leaf crosses different structural boundaries.
Let $W(u)$ be the size of the subtree rooted at u in the original tree. In the balanced divide-and-conquer step (merge), a sequence of items with total weight W is split into two halves of weight at most $W/2 + w_i$. Because the weights used are the actual sizes of the subtrees in the original tree, moving from a parent to a child in the Compress or Rake trees reduces the geometric weight bounds proportionally.

A path from the root of the top tree down to a target vertex will:

1. Traverse down a Compress binary tree. By the weight-balanced split, the weight decreases by a constant fraction at each step.
2. Transition through an AddEdge / AddVertex node. This corresponds to crossing a light edge in the original tree. By the properties of Heavy-Light Decomposition, a light child has at most half the size of its parent ($W(v) \leq W(u)/2$).

Whether the traversal drops down a Compress/Rake tree or takes a light branch, the represented subtree size W strictly decreases by a constant factor. Therefore, the maximum sequence of operations is bounded by $O(\log n)$.

Construction Time and Space: The HLD sizing takes $O(n)$. During construction, the merge function processes arrays of elements. Since each level of the divide-and-conquer split takes linear time with respect to the array size, and the sum of array sizes across the tree relies on n, the splitting creates a recurrence bounded by $O(n \log n)$. Every vertex induces exactly one Vertex/AddVertex node and at most one AddEdge node. The binary trees for Compress and Rake introduce at most $k - 1$ internal nodes for k leaves. Thus, the total number of nodes in the top tree is strictly bounded by $O(n)$, yielding an $O(n)$ space complexity. □

4.4 XOR-Linked Traversal

XOR-Linked Traversal is a method to traverse a tree bottom-up iteratively with better constant factor.

We store:

- $d[u]$: The degree of node u.
- $p[u]$: The XOR sum of the indices of all neighbors of node u. Initially, $p[u] = \bigoplus_{v \in \text{neighbors}(u)} v$.

If a node u is a leaf, its degree is 1. Therefore, $p[u]$ is simply the index of its only neighbor, v. We can process the edge (u, v), and then delete the edge from the graph using $p[v] \leftarrow p[v] \oplus u$ and decrementing its degree.

If decrementing v's degree makes it a new leaf, we set $u \leftarrow v$ and continue.

Algorithm 7: XOR-Linked Traversal

Input: Number of nodes N, Initial XOR-sum array p, Initial degree array d
Output: Updated arrays p and d, processing nodes bottom-up

```
for i ← 0 to N − 1 do
    u ← i;
    while d[u] = 1 do
        d[u] ← 0;
        v ← p[u];
        Process(u, v);
        p[v] ← p[v] ⊕ u;
        d[v] ← d[v] − 1;
        u ← v;
return p, d;
```

4.5 DFS

The algorithm constructs a DFS order, alongside subtree sizes and depths, with a better constant factor.

Algorithm 8: DFS

Input: Number of nodes N, Parent array p, Degree array d
Output: Arrays $p, ss, pos, dfs, depth$

Initialize $ss \leftarrow [1, \ldots, 1]$ of size N;
Initialize $dfs \leftarrow [0, \ldots, 0]$ of size N;
Initialize $depth \leftarrow [0, \ldots, 0]$ of size N;
$idx \leftarrow N - 1$;
foreach *edge* (u, v) *processed via XOR-Linked Traversal* **do**
    $ss[v] \leftarrow ss[v] + ss[u]$;
    $dfs[idx] \leftarrow u$;
    $idx \leftarrow idx - 1$;
$pos \leftarrow ss$;
for $i \leftarrow 1$ **to** $N - 1$ **do**
    $u \leftarrow dfs[i]$;
    $v \leftarrow p[u]$;
    $depth[u] \leftarrow depth[v] + 1$;
    $temp \leftarrow pos[v]$;
    $pos[v] \leftarrow pos[v] - pos[u]$;
    $pos[u] \leftarrow temp$;
for $i \leftarrow 0$ **to** $N - 1$ **do**
    $pos[i] \leftarrow pos[i] - 1$;
    $dfs[pos[i]] \leftarrow i$;
return $p, ss, pos, dfs, depth$;

1. $ss[v] \leftarrow ss[v] + ss[u]$ calculates the subtree size for every node.
2. We iterate top-down using the order in dfs. We initialize a pos array with the subtree sizes ss.
    - Imagine node v owns a contiguous block of indices ending at $pos[v]$.
    - It gives its child u a chunk of this block equal to u's subtree size. u's block now ends where v's block ended.
    - v's remaining block end pointer is shifted back by u's size ($pos[v] - pos[u]$).
    - As this repeats for all children of v, the children partition the space reserved for v's descendants, and v's pointer ends up pointing to the beginning of its block.
3. $pos[i]$ points to the 1-based index where node i should appear. Subtracting 1 makes it a 0-indexed DFS array, and placing i at $dfs[pos[i]]$ computes dfs.

4.6 Diameter

With the bottom-up sequence from XOR-Linked traversal, we compute the diameter of the tree using DP.

Algorithm 9: Tree Diameter

Input: Number of nodes N, Parent array p, Degree array d
Output: Integer representing the tree diameter

Initialize $max\_depth \leftarrow [0, \ldots, 0]$ of size N;
Initialize $diameter \leftarrow [0, \ldots, 0]$ of size N;
foreach *edge* (u, v) *processed via XOR-Linked traversal* **do**
    `/* Path through` v `connecting` u`'s deepest leaf and previously processed children's deepest leaf */`
    $path\_through\_v \leftarrow max\_depth[v] + max\_depth[u] + 1$;
    $diameter[v] \leftarrow \max(\{diameter[v], diameter[u], path\_through\_v\})$;
    $max\_depth[v] \leftarrow \max(max\_depth[v], max\_depth[u] + 1)$;
return $\max_{i \in [0, N-1]} diameter[i]$;

The algorithm maintains two variables for each vertex v:

- $max\_depth[v]$: The length of the longest path from v down to a leaf in its subtree already processed.
- $diameter[v]$: The maximum diameter completely contained within the subtree rooted at v already processed.

Because the traversal processes children before their parents, when we evaluate the edge between child u and parent v, $max\_depth[u]$ and $diameter[u]$ are already computed. We update v's state by checking if joining u's deepest path to v's currently known deepest path creates a new maximum diameter for v's subtree. Finally, we update v's maximum depth to include u.

4.7 Shallowest Decomposition Tree

The Shallowest Decomposition Tree possesses the same properties as a Centroid Decomposition but can be constructed in $O(n)$ time with optimal depth.

Definition 4.3 (Valid Tree Labeling)**.** *A valid tree labeling is an assignment of integer labels $\ell : V \to \mathbb{N}$ such that for any two distinct vertices u, v sharing the same label ($\ell(u) = \ell(v)$), the simple path between them contains at least one vertex w with a strictly greater label ($\ell(w) > \ell(u)$).*

The depth of the resulting decomposition tree is exactly the maximum label assigned.

1. The DP We compute the optimal labeling in $O(n)$ time using a bottom-up DP. For each node u, we maintain a bitmask $\mathcal{F}(u)$ representing the set of forbidden labels. The k-th bit of $\mathcal{F}(u)$ is 1 if assigning label k to u would violate the labeling condition with its descendants.

The optimal label $\ell(u)$ is the smallest non-forbidden integer, corresponding to the index of the lowest unset bit in $\mathcal{F}(u)$.

When a child v passes its constraints up to its parent u, the newly assigned label $\ell(v)$ acts as a "shield." Any forbidden labels strictly smaller than $\ell(v)$ are shielded and are no longer forbidden for u. However, $\ell(v)$ itself becomes forbidden for u, and any constraints strictly greater than $\ell(v)$ pass through unaffected.

It is easy to see that the propagation is exactly:

$$\mathcal{P}(v) = \mathcal{F}(v) + 1$$

When u aggregates the propagated masks $\mathcal{P}(v)$ from all its children, it must resolve collisions:

1. **Union:** If a label is forbidden by any child, it is forbidden for u. Let $\mathcal{O}_{\text{once}} = \bigvee \mathcal{P}(v)$.
2. **Collision:** If a label k is forbidden by two or more distinct children, u must act as a shield for them by taking a label strictly greater than k. Let $\mathcal{O}_{\text{twice}}$ be the mask of bits set by at least two children.

If $\mathcal{O}_{\text{twice}} > 0$, let $k = \lfloor \log_2(\mathcal{O}_{\text{twice}}) \rfloor$ be its highest set bit. To force $\ell(u) > k$, we saturate all bits up to k by applying a mask $M_k = (1 \ll (k+1)) - 1$. The final forbidden state for u is exactly:

$$\mathcal{F}(u) = \mathcal{O}_{\text{once}} \vee M_k$$

From this, we evaluate $\ell(u) = \text{trailing\_zeros}(\mathcal{F}(u) + 1)$.

2. Decomposition Chains Let $\text{anc}(x)$ denote the lowest ancestor of x in the original rooted tree such that $\ell(\text{anc}(x)) > \ell(x)$. In the decomposition tree, u becomes the direct parent of all vertices x where $\text{anc}(x) = u$.

Two vertices x and y sharing the same $\text{anc}(x) = u$ must reside in the subtree of the same child v of u. Furthermore, they must have distinct labels (if they shared a label, a higher label would exist between them, violating the premise that u is their lowest higher-labeled ancestor).

Thus, the set of vertices in v's subtree that attach to u form a strict total order: $x_1 > x_2 > \cdots > x_k$. When u is removed during decomposition, x_1 becomes the highest-labeled root of the newly isolated component, x_2 becomes its child, and so on. This vertical parent-child line in the decomposition tree is called a Decomposition Chain.

The labels present in the chain from child v to parent u are exactly the labels exposed in v's subtree that are strictly less than $\ell(u)$. Because $\mathcal{P}(v)$ encodes all exposed labels, we can extract this chain mask:

$$\text{ChainMask}(v \to u) = \mathcal{P}(v) \wedge ((1 \ll \ell(u)) - 1)$$

2. Global Chain Consider the set of vertices $S \subseteq V$ that have no higher-labeled ancestor in the original rooted tree. These vertices represent the sequence of maximum labels in the decomposition process.

By the same proof as before, the labels in S are distinct, let them be $x_1 > x_2 > \cdots > x_k$.

In the decomposition tree, the vertex x_1 (which possesses the maximum label in the entire tree) naturally becomes the root, r_{decomp}. When x_1 is removed during decomposition, x_2 becomes the highest-labeled root of its newly isolated component, thus a child of x_1. Thus the vertices in S form a chain.

3. Vertex Retrieval Knowing the labels in a chain is insufficient; we must retrieve the vertices in $O(1)$ time.

We maintain a global array of stacks, allocating one stack per possible label. During the DFS, immediately after computing $\ell(u)$, we push u onto stacks$[\ell(u)]$.

Because multiple subtrees share this global stack, we must manage the traversal order.

- **Forward:** In the DFS, we iterate through the children of u in forward order $(v_1, v_2, \ldots, v_m)$. Their vertices are recursively evaluated and pushed onto the stacks in that order. The vertices belonging to the final child v_m reside at the top of the stacks.
- **Reversed:** To extract the chains to form edges, we iterate through the children of u in reverse $(v_m, \ldots, v_1)$. Because we process the last child v_m first, the vertices defined by its ChainMask are guaranteed to be exactly at the top of their respective stacks.

For a given ChainMask, we iterate through its set bits in decreasing order (extracting the highest bit $L = \lfloor \log_2(\text{mask}) \rfloor$ first). This ensures that higher-labeled vertices become ancestors of lower-labeled vertices in the decomposition tree. We pop the vertex w from stacks$[L]$, add the directed edge $u \to w$, and update $u \leftarrow w$ to continue forming the chain downwards.

Algorithm 10: Shallowest Decomposition Tree

```
Function ExtractChain(mask, u):
    while mask ≠ 0 do
        L ← ⌊log2(mask)⌋;
        mask ← mask ⊕ (1 ≪ L);
        if stacks[L] ≠ ∅ then
            v ← stacks[L].pop();
            Add edge u → v to decomposition tree;
            u ← v;
Function DFS(u, p):
    once ← 0; twice ← 0;
    foreach v ∈ adj[u] do
        if v ≠ p then
            DFS(v, u);
            prop ← forbidden[v] + 1;
            twice ← twice ∨ (once ∧ prop);
            once ← once ∨ prop;
    if twice = 0 then M ← 0;
    else
        k ← ⌊log2(twice)⌋;
        M ← (1 ≪ (k + 1)) − 1;
    forbidden[u] ← once ∨ M;
    label_u ← trailing_zeros(forbidden[u] + 1);
    stacks[label_u].push(u);
    foreach v ∈ reversed(adj[u]) do
        if v ≠ p then
            extract_mask ← (forbidden[v] + 1) ∧ ((1 ≪ label_u) − 1);
            ExtractChain(extract_mask, u);
Function ShallowestDecomp(adj, root):
    Initialize forbidden array to 0 for all nodes;
    Initialize stacks as an array of empty stacks;
    DFS(root, null);
    max_label ← ⌊log2(forbidden[root] + 1)⌋;
    decomp_root ← stacks[max_label].pop();
    final_mask ← (forbidden[root] + 1) ∧ ((1 ≪ max_label) − 1);
    ExtractChain(final_mask, decomp_root);
    return decomp_root;
```

4.8 Lazy Anti-Monopoly Tree

The Lazy AM-Tree maintains an incremental MSF in $O(\log n)$ amortized time per insertion. It maintains a Transformed MST (T-MST), a rooted tree that minimizes depth while preserving the results of Path-Max queries.

TW-Transformations and Promotion A fundamental property of T-MSTs is the *TW-Transformation*: given two adjacent edges (x, y, w_1) and (y, z, w_2) with $w_1 \geq w_2$, replacing the edge (x, y, w_1) with (x, z, w_1) preserves all Path-Max queries in the tree.

The Promote(x) operation leverages this to move a node x up the tree while preserving Path-Max queries. Let $y = parent[x]$ and $z = parent[y]$, with edge weights $w_1 = weight(x \to y)$ and $w_2 = weight(y \to z)$.

- **Shortcut** ($w_1 > w_2$): x is directly promoted to become z's child with weight w_1. Node y becomes a sibling of x.
- **Rotate** ($w_1 < w_2$ or y is root): y is pushed down to become a child of x with weight w_1. If y is not the root, x is attached to z with weight w_2.

Stitch-Based Insertion Instead of finding the heaviest edge on the cycle, the Stitch-based algorithm moves the new edge (u, v, w) to its correct position using TW-transformations.

If $weight(u \to parent[u]) < w$, the edge (u, v, w) is TW-transformed to $(parent[u], v, w)$ and recursively moved upwards. Once both $weight[u \to parent[u]] > w$ and $weight[v \to parent[v]] > w$, we attach the smaller subtree to the larger one (e.g., u becomes a child of v with weight w). This displaces u's old parent edge (u, u', w'). Because $w' > w$, we apply another TW-transformation, moving the displaced edge to (u', v, w'), and recursively route it.

If the original edge closed a cycle, the relocation forces the two endpoints to eventually meet at their LCA ($u = v$). At this point, we discard the edge.

Algorithm 11: Lazy AM-Tree

```
1  Function Promote(x):
2      y ← parent[x];
3      z ← parent[y];
4      w1 ← weight(x → y); w2 ← weight(y → z);
5      if z ≠ null and w1 > w2 then
           // Shortcut:  x bypasses y to become its sibling
6          size[y] ← size[y] − size[x];
7          parent[x] ← z;
8          weight(x → parent) ← w1;
9      else
           // Rotate:  y is pushed down as a child of x
10         size[y] ← size[y] − size[x];
11         size[x] ← size[x] + size[y];
12         parent[x] ← z;
13         weight(x → parent) ← w2;
14         parent[y] ← x;
15         weight(y → parent) ← w1;

16 Function UpwardCalibrate(x):
17     while parent[x] ≠ null do
18         y ← parent[x];
19         if size[x] ≤ (2/3) size[y] then x ← y;
20         else Promote(x);

21 Function LinkByStitch(u, v, w):
22     if u = v or u = null or v = null then return;
23     if parent[u] ≠ null and w > weight(u → parent) then LinkByStitch(parent[u], v, w) ;
24     else if parent[v] ≠ null and w > weight(v → parent) then LinkByStitch(u, parent[v], w) ;
25     else
26         if size[u] > size[v] then Swap u ↔ v;
27         u_old_parent ← parent[u];
28         w_old ← weight(u → parent);
29         parent[u] ← v;
30         weight(u → parent) ← w;
31         size[v] ← size[v] + size[u];
           // Route the displaced heavier edge
32         LinkByStitch(u_old_parent, v, w_old);

33 Function LazyInsert(u, v, w):
34     UpwardCalibrate(u);
35     UpwardCalibrate(v);
36     LinkByStitch(u, v, w);
```

Correctness and Complexity

Theorem 4.4. *The Lazy AM-Tree maintains a valid T-MST and supports insertions in $O(\log n)$ amortized time.*

Proof. **Correctness:** Every operation in LinkByStitch is a valid TW-transformation. By continually replacing edges with equivalent higher-reaching edges, the sequence preserves Path-Max queries. If u and v were previously connected, the endpoints of the heaviest cycle-edge move to their LCA, where $u = v$ triggers the base case that drops the redundant edge. Therefore, the tree remains valid.
Complexity: We use the potential function $\Phi = \sum_{i=1}^{n} \log(size[i])$.

1. **Calibrate:** The Promote function is only called when x is a heavy child ($size[x] > \frac{2}{3} size[y]$). Under this condition, rotating x upwards decreases y's size by at least a factor of $1/3$, meaning Φ decreases by an amount strictly greater than the $O(1)$ actual cost of the rotation. Therefore, Promote has an amortized cost of zero. Because the loop only advances upward when the AM-rule holds, size increases by a factor of $3/2$ per advancement, limiting iterations to $O(\log n)$.

2. **Link:** The LinkByStitch recursion traverses upward, performing $O(d(u) + d(v))$ actual operations. Structural size changes only occur when the smaller subtree is attached to the larger one. Because the smaller subtree's size

at most doubles, the potential Φ increases by at most 1 per topological attachment. Across the $O(d(u) + d(v))$ affected nodes, the total potential increase is bounded by $O(\log n)$.

Thus the total amortized cost per insertion is $O(\log n)$. □

5 Data Structures

5.1 RMQ

We want to find the index of the minimum element within an arbitrary subarray $A[l \dots r]$. We can achieve $O(1)$ query time with $O(n)$ preprocessing time and space.

1. Blocks We partition the array into blocks of size $B = \Theta(\log n)$. For each index $i \in [0, n-1]$, we maintain a bitmask $mask[i]$ of length up to B. This mask represents a strictly increasing subsequence of values ending at i, extending leftward up to the block boundary or at most B elements.

Specifically, the k-th bit (0-indexed) of $mask[i]$ is set to 1 if and only if $A[i-k]$ is the minimum element in the suffix $A[i-k \dots i]$. As we iterate i from 0 to $n-1$, we compute $mask[i]$ iteratively:

1. Shift the previous mask left by 1: $mask_{\text{curr}} \leftarrow mask_{i-1} \ll 1$.
2. While the current element $A[i]$ is strictly less than the element corresponding to the lowest set bit in $mask_{\text{curr}}$, unset that bit.
3. Set the 0-th bit to 1, representing the element $A[i]$ itself.

2. Sparse Table We construct a Sparse Table over the $\lfloor \frac{n}{B} \rfloor$ blocks. Instead of storing all elements, the 0-th level of the Sparse Table only stores the overall minimum of each block. The minimum of the j-th block (which ends at index $i = j \cdot B + B - 1$) is found using its mask: $mask[i]$ contains suffix minimums, so its highest set bit is the minimum of the entire block. Because there are only n/B blocks, the Sparse Table takes $O((n/B)\log(n/B)) = O(n)$ time and space.

3. Query Resolution To answer a query for the range $[l, r]$:

- **Case 1: The range spans at most B elements.** The range is small enough that the entire answer is encoded within $mask[r]$. We truncate the mask to only consider the last $r - l + 1$ elements by doing a bitwise AND with $(1 \ll (r - l + 1)) - 1$. The index of the highest set bit in this truncated mask corresponds to the minimum in $[l, r]$.
- **Case 2: The range spans multiple blocks.** We decompose the range into three parts:
  1. A partial block on the left: from l to the end of its block.
  2. A set of completely covered intermediate blocks.
  3. A partial block on the right: from the start of its block to r.

  The minimum of the left partial block is found using the highest set bit of the mask at the end of that block. The minimum of the right partial block is found using the highest set bit of $mask[r]$. The minimum of the completely covered intermediate blocks is queried in $O(1)$ time from the Sparse Table. We simply take the minimum of these three candidates.

Algorithm 12: RMQ

```
Function Init(A[0 ... n − 1]):
    B ← Θ(log n);
    mask ← array of size n;
    curr_mask ← 0;
    for i ← 0 to n − 1 do
        curr_mask ← curr_mask ≪ 1;
        while curr_mask ≠ 0 and A[i] < A[i − trailing_zeros(curr_mask)] do
            curr_mask ← curr_mask & (curr_mask − 1) ;                    // Pops the lowest set bit
        curr_mask ← curr_mask | 1;
        mask[i] ← curr_mask;
    // Build Sparse Table over Blocks
    num_blocks ← ⌊n/B⌋;
    ST ← 2D array of size log(num_blocks) × num_blocks;
    for j ← 0 to num_blocks − 1 do
        end_idx ← j · B + B − 1;
        ST[0][j] ← end_idx − ⌊log2(mask[end_idx])⌋;
    for k ← 1 to ⌊log2(num_blocks)⌋ do
        for j ← 0 to num_blocks − (1 ≪ k) do
            ST[k][j] ← min(ST[k − 1][j], ST[k − 1][j + (1 ≪ (k − 1))]);

Function Query(l, r):
    if r − l + 1 ≤ B then
        trunc_mask ← mask[r] & ((1 ≪ (r − l + 1)) − 1);
        return r − ⌊log2(trunc_mask)⌋;
    // Left partial block
    left_end ← (l/B) · B + B − 1;
    ans ← left_end − ⌊log2(mask[left_end])⌋;
    // Right partial block
    ans ← min(ans, r − ⌊log2(mask[r])⌋);
    // Intermediate full blocks
    x ← l/B + 1;
    y ← r/B − 1;
    if x ≤ y then
        k ← ⌊log2(y − x + 1)⌋;
        mid_ans ← min(ST[k][x], ST[k][y − (1 ≪ k) + 1]);
        ans ← min(ans, mid_ans);
    return ans;
```

5.2 Concave Min-Plus Convolution

5.2.1 Description

Given an array a of length n (which is concave, meaning its forward differences are monotonically decreasing) and an arbitrary array b of length m, the concave min-plus convolution computes:

$$c[p] = \min_{0 \le x < n,\, 0 \le y < m,\, x+y=p} (a[x] + b[y])$$

We give an $O(m \log n + n)$ solution by chunking the array b and using a modified Li Chao Tree.

The algorithm processes b in chunks of size up to n. For a chunk starting at index s, we compute its contributions to a window of length $2n$ in the output array c using two passes:

- **Forward Pass:** Evaluates pairs where the relative index in a is greater than or equal to the relative index in b. It inserts functions $f_k(i) = a[i-k] + b[s+k]$ into a Li Chao tree and queries them for $i \ge k$.
- **Reverse Pass:** Symmetrically handles the remaining cross-terms where the relative index in a is strictly less than the relative index in b, sweeping backwards.

5.2.2 Crossings

Lemma 5.1 (Single Crossing Property)**.** *For a concave array a, define the extended functions for $k \in [0, n)$:*

$$f_k(i) = \begin{cases} -\infty - k & \text{if } i < k \\ a[i-k] + b[s+k] & \text{if } i \geq k \end{cases}$$

For any $j < k$, the difference $D(i) = f_k(i) - f_j(i)$ transitions from negative to positive at most once.

Proof. We evaluate the difference $D(i)$ across three domains:

1. **For $i < j$:** Both are artificial. $D(i) = (-\infty - k) - (-\infty - j) = j - k < 0$.
2. **For $j \leq i < k$:** f_j is finite, but f_k is artificial. $D(i) = -\infty < 0$.
3. **For $i \geq k$:** Both are finite. $D(i) = a[i-k] - a[i-j] + C$. Because a is concave, its slope is decreasing. Since $i - k < i - j$, the difference $a[i-k] - a[i-j]$ strictly monotonically increases with i.

Because $D(i)$ is strictly negative for all $i < k$ and monotonically increases for all $i \geq k$, the functions f_j and f_k intersect at most once globally. □

5.2.3 Boundary Conditions

While Theorem 5.1 proves the functions cross at most once, their shapes are highly irregular due to the jump at $i = k$.

A standard Li Chao Tree assumes that if a function is better at the left endpoint L and worse at the midpoint M, the single continuous crossing must lie within $[L, M]$, and recursively routes the function leftward.

A function f_k might be artificially better at L (because it's about $-\infty$), worse at M, but then become the true minimum on the right side of the interval.

For a node spanning the discrete interval $[L, R)$, let z be the current resident function and x be the newly inserted function.

- If $x(L) \geq z(L)$ and $x(R-1) \geq z(R-1)$, then x is dominated at both strict boundaries. By Lemma 5.1, it must be dominated everywhere inside the interval. We safely discard x.
- If $x(L) < z(L)$ and $x(R-1) < z(R-1)$, then x dominates z everywhere in the interval. We replace z with x and discard z.
- Otherwise, a crossing exists within the interval. We swap x and z if necessary to maintain the best function at the midpoint, and recursively push the loser to the appropriate child.

5.3 DSU

We prove the $O(\alpha(n))$ amortized bound for DSU using a potential function. The Ackermann functions are here $A_k : \mathbb{N} \to \mathbb{N}$ defined recursively as $A_0(l) = l + 1$ and $A_{k+1}(l) = A_k^{(l+1)}(1)$. We define $SIZE^{max}(p)$ as the maximum number of descendants a node p ever has, which guarantees $SIZE^{max}(PARENT(p)) \geq 2 \cdot SIZE^{max}(p)$ for any non-root node.

Let the rank of a node be $r(p) = \lfloor \log SIZE^{max}(p) \rfloor$. This rank satisfies $r(PARENT(p)) \geq r(p) + 1$. We define the potential function for any non-root node p as:

$$\Phi(p) := \sum_{k=0}^{\alpha(n)} \sum_{l=1}^{r(p)} [A_k^{(l+1)}(r(p)) > r(PARENT(p))]$$

This potential function satisfies the necessary properties for amortized analysis:

- **Monotonicity:** Since $SIZE^{max}(PARENT(p))$ is nondecreasing over time, the potential $\Phi(p)$ is nonincreasing for any non-root node p.
- **Boundedness:** The maximum potential is bounded by definition, $\Phi(p) \leq (\alpha(n) + 1)r(p)$.

Consider a path compression step along the find-path $p_1 \to p_2 \to \cdots \to p_m$, where $p_m = r$ is the root. For any node p_i $(1 \leq i < m)$, let k_i be the largest $k \geq 0$ such that $A_k(r(p_i)) \leq r(p_{i+1})$. We know $k_i \leq \alpha(n)$ because $A_{\alpha(n)+1}(1) > n \geq SIZE^{max}(p_{i+1}) \geq r(p_{i+1})$. Next, let l_i be the largest $l \geq 1$ for which $A_{k_i}^{(l)}(r(p_i)) \leq r(p_{i+1})$. We are guaranteed $l_i \leq r(p_i)$.

For any fixed $0 \leq k \leq \alpha(n)$, consider all nodes p_i along the path where $k_i = k$. All such nodes, except the one closest to the root, will have their potential decrease by at least 1. Consider any such node p_i that is not closest to the root; its old parent p_{i+1} satisfies:

$$A_k^{(l_i)}(r(p_i)) \leq r(p_{i+1}) < A_k^{(l_i+1)}(r(p_i))$$

Because p_i is not the closest to the root among nodes with $k_j = k$, there exists some $j > i$ such that $k_j = k$, meaning $r(p_{j+1}) \geq A_k(r(p_j))$. Because ranks monotonically increase along the path, $r(p_m) \geq r(p_{j+1})$ and $r(p_j) \geq r(p_{i+1})$. Thus:

$$r(p_m) \geq A_k(r(p_{i+1})) \geq A_k^{(l_i+1)}(r(p_i))$$

When path compression updates p_i's parent from p_{i+1} directly to p_m, the indicator condition $A_k^{(l_i+1)}(r(p_i)) > r(PARENT(p_i))$ changes from true (1) to false (0). Therefore, for each $0 \leq k \leq \alpha(n)$, there is at most one node where the potential fails to decrease. This bounds the amortized cost per find step to $O(\alpha(n))$.

5.4 DSU with Potential

The DSU with Potential, maintains disjoint sets while tracking relative differences (potentials) between elements in the same set. The potentials must belong to a group G.

For any node x, the array $\mathtt{p}[x]$ maintains the potential difference from x to its current parent in the DSU tree, which we denote as $\Delta(x, \text{parent}(x))$.

1. Path Compression When querying the root of a node x, the algorithm recursively compresses the path. Suppose the recursive call finds the root R and the potential from the parent to the root, $\Delta(\text{parent}(x), R)$.

To update x to point directly to R, we compose the potential from x to its parent with the potential from its parent to R:

$$\Delta(x, R) = \Delta(x, \text{parent}(x)) \circ \Delta(\text{parent}(x), R)$$

2. Querying Potential To find the potential $\Delta(a, b)$ from node a to node b, we first verify they belong to the same set. Let R be their common root. The `find` operation gives us $v_a = \Delta(a, R)$ and $v_b = \Delta(b, R)$.

The path from a to b can be constructed by traversing from a up to R, and then down from R to b. Thus, the combined potential is:

$$\Delta(a, b) = \Delta(a, R) \circ \Delta(R, b) = v_a \circ v_b^{-1}$$

3. Edge Insertion When merging the sets containing a and b by adding a directed edge from a to b with a given potential P, we require that $\Delta(a, b) = P$.

Let R_a and R_b be the respective roots of a and b. Suppose we link R_a as a child of R_b. We need to compute the required weight $W = \Delta(R_a, R_b)$ for this new edge.

We know the full path $a \to R_a \to R_b \to b$ must equal the target potential P:

$$\Delta(a, R_a) \circ \Delta(R_a, R_b) \circ \Delta(R_b, b) = P$$

Substituting our known variables $v_a = \Delta(a, R_a)$, $W = \Delta(R_a, R_b)$, and $v_b^{-1} = \Delta(R_b, b)$:

$$v_a \circ W \circ v_b^{-1} = P$$

Thus:

$$W = v_a^{-1} \circ P \circ v_b$$

If the union-by-size heuristic determined that R_b should become a child of R_a, the weight of the reversed edge is the inverse W^{-1}.

5.5 Persistent Queue

A fully persistent queue allows us to push and pop elements from any historical state of the queue. By modeling the sequence of pushes as a tree, we can implement it by using Algorithm 6.

1. State Each state in the persistent queue is represented as a node in a tree. A node u stores:

- v: The value inserted at this state.
- p: The parent node (the state of the queue before this element was pushed).
- depth: The depth of the node in the tree (the total number of pushes to reach this state).
- jmp: The jump pointer used for ancestor queries.
- fr: A pointer to the node representing the current front of the queue.

2. Push Operation When pushing a value onto a previous state p, we append a new child node u. The front pointer fr is inherited from p

We compute the new jump pointer $\text{jmp}[u]$ in $O(1)$ time using the combination rule from Algorithm 6. We check:

$$\text{depth}[p] - \text{depth}[\text{jmp}[p]] = \text{depth}[\text{jmp}[p]] - \text{depth}[\text{jmp}[\text{jmp}[p]]]$$

If they are equal, we merge them: $\text{jmp}[u] \leftarrow \text{jmp}[\text{jmp}[p]]$. Otherwise, we reset the jump length: $\text{jmp}[u] \leftarrow p$.

3. Pop Operation To pop an element from state p, we must advance the front pointer fr to the next element in the queue. Because the queue's elements form a path from fr down to the current tail p, the next element to become the front is the child of fr that lies on the path to p.

This is a Level Ancestor query: we must find the ancestor of p whose depth is exactly $\text{depth}[fr] + 1$. We can find this in $O(\log n)$ time and then create a new state that copies p, but updates its fr pointer to this new ancestor.

4. Front Operation Querying the front of the queue simply requires reading the value stored at the node pointed to by fr.

Algorithm 13: Persistent Queue

```
1  Function Push(v, p):
2      if p = NULL then
3          return QueueNode(v, parent: NULL, fr: nxt, jmp: nxt, depth: 0);
4      fr ← nodes[p].fr;
5      p_j ← nodes[p].jmp;
6      p_jj ← nodes[p_j].jmp;
7      if depth[p] − depth[p_j] = depth[p_j] − depth[p_jj] then
8          jmp ← p_jj;
9      else
10         jmp ← p;
11     depth ← nodes[p].depth + 1;
12     return QueueNode(v, p, fr, jmp, depth);

13 Function LevelAncestor(p, target_depth):
14     while depth[p] > target_depth do
15         if depth[jmp[p]] ≥ target_depth then
16             p ← jmp[p];
17         else
18             p ← parent[p];
19     return p;

20 Function Pop(p):
21     new_node ← nodes[p].clone();
22     target_depth ← nodes[nodes[p].fr].depth + 1;
23     new_node.fr ← LevelAncestor(p, target_depth);
24     return new_node;
```

5.6 Online Suffix Max

To answer suffix maximum queries $\max(A[i \dots n-1])$ in a setting where elements are appended, we can combine a monotonic stack with a DSU.

1. Monotonic Stack When a new element x is appended at index p, it invalidates any previous elements smaller than it. We maintain a stack of indices corresponding to a strictly decreasing subsequence of the array. As x is pushed, we pop all indices j from the stack where $A[j] \leq x$.

2. DSU Any index whose suffix maximum was previously $A[j]$ now shares a suffix maximum of x. For every popped index j, we union its set into p's set. Finally, we push p onto the stack.

3. Queries To find the maximum in the suffix starting at index i, we simply query the DSU for the root of i.

6 Formal Power Series

6.1 Inverse

To compute the inverse $F = A^{-1}$, we want to find the root of the equation $\Phi(F) = F^{-1} - A = 0$.

Taking the derivative with respect to F gives $\Phi'(F) = -F^{-2}$. Plugging this into the Newton iteration formula $F_{k+1} = F_k - \frac{\Phi(F_k)}{\Phi'(F_k)}$ yields:

$$F_{k+1} = F_k - \frac{F_k^{-1} - A}{-F_k^{-2}} = F_k + F_k(1 - AF_k)$$

$$F_{k+1} = F_k(2 - AF_k) \pmod{x^{2^{k+1}}}$$

Because F_k is already the inverse modulo x^k, we know that $AF_k \equiv 1 \pmod{x^k}$. We can express this product as $AF_k = 1 + x^k E$, where E represents the upper coefficients of the product.

Substituting this into our Newton iteration:

$$F_{k+1} = F_k - F_k(AF_k - 1) = F_k - F_k(x^k E)$$

$$F_{k+1} = F_k - x^k(F_k E \pmod{x^k})$$

This confirms that the lower k coefficients of F_{k+1} are exactly F_k. To compute the upper k coefficients, we only need to extract E and multiply it by F_k.

To avoid the cost of negating the result array at every step, the algorithm maintains the negative inverse $R_k = -F_k$. Evaluating the product yields $AR_k = -1 - x^k E$. By shifting the coefficients right by k, we isolate $-E$. Multiplying this by R_k yields $(-E)(-F_k) = EF_k$.

Because $F_{k+1} = F_k - x^k(EF_k)$, we have $-F_{k+1} = -F_k + x^k(EF_k)$, meaning R_{k+1} is formed by writing EF_k into the upper half of R_k.

6.2 Logarithm

To compute $F = \ln(A)$, we take the formal derivative:

$$F' = \frac{A'}{A}$$

Then we just formally integrate:

$$F = \int \frac{A'}{A}\, dx$$

6.3 Square Root

To compute the square root $F = \sqrt{A}$, we want to solve $\Phi(F) = F^2 - A = 0$.

The derivative is $\Phi'(F) = 2F$. The standard Newton iteration becomes:

$$F_{k+1} = F_k - \frac{F_k^2 - A}{2F_k} \pmod{x^{2^{k+1}}}$$

We simultaneously maintain the inverse $G_k \equiv F_k^{-1} \pmod{x^{2^k}}$.

1. Updating the Square Root We can rewrite the square root iteration using G_k:

$$F_{k+1} = F_k - \frac{1}{2}G_k(F_k^2 - A) \pmod{x^{2^{k+1}}}$$

Because $F_k^2 \equiv A \pmod{x^{2^k}}$ by definition of the previous step, the lower 2^k coefficients of $(F_k^2 - A)$ are guaranteed to be zero. Let Δ represent the non-zero upper coefficients of this error term. We can compute F_{k+1} strictly by evaluating:

$$F_{k+1} = F_k - x^{2^k}\left(\frac{1}{2}G_k\Delta \pmod{x^{2^k}}\right)$$

2. Updating the Inverse Once F_{k+1} is computed, we must update our maintained inverse so that $G_{k+1} \equiv F_{k+1}^{-1} \pmod{x^{2^{k+1}}}$. We apply one step of the standard Newton iteration for polynomial inversion, substituting our newly found F_{k+1}:

$$G_{k+1} = G_k(2 - F_{k+1}G_k) \pmod{x^{2^{k+1}}}$$

$$G_{k+1} = G_k - G_k(F_{k+1}G_k - 1) \pmod{x^{2^{k+1}}}$$

Just like the square root update, the lower 2^k coefficients of $(F_{k+1}G_k - 1)$ are guaranteed to vanish. We extract the upper coefficients and multiply them by G_k to find the upper half of G_{k+1}.

6.4 Inverse Square Root

To compute the inverse square root $F = 1/\sqrt{A}$, we want to solve $\Phi(F) = F^{-2} - A = 0$.

The derivative is $\Phi'(F) = -2F^{-3}$. The iteration becomes:

$$F_{k+1} = F_k - \frac{F_k^{-2} - A}{-2F_k^{-3}} = F_k + \frac{F_k}{2}(1 - AF_k^2)$$

$$F_{k+1} = \frac{F_k}{2}(3 - AF_k^2) \pmod{x^{2^{k+1}}}$$

6.5 Exponential

The newton iteration for exponential $F = \exp(A)$ is $F_{2k} \equiv F_k(1 + A_{2k} - \ln F_k) \pmod{x^{2k}}$.

We define the differential operator $\theta = x\frac{d}{dx}$. Because $F_k \equiv \exp(A_k) \pmod{x^k}$, we have $\theta F_k \equiv (\theta A_k)\exp(A_k) \equiv (\theta A_k)F_k \pmod{x^{k-1}}$. Let the residual be V:

$$V = (\theta A_k)F_k - \theta F_k$$

We have that $V \equiv 0 \pmod{x^{k-1}}$. Furthermore, θF_k has degree at most $k-1$. Therefore:

$$V_{\geq k} = \big((\theta A_k)F_k\big)_{\geq k}$$

We will use $\ln F_k = \int F_k' F_k^{-1}$. Let $G_{2k} := F_k^{-1} \pmod{x^{2k}}$. Rearranging the residual definition yields $\theta F_k = (\theta A_k)F_k - V$. Multiplying both sides by G_{2k} gives:

$$(\theta F_k)G_{2k} = (\theta A_k)F_k G_{2k} - VG_{2k}$$

Since $F_k G_{2k} \equiv 1 \pmod{x^{2k}}$, we have:

$$(\theta F_k)G_{2k} \equiv \theta A_k - VG_{2k} \pmod{x^{2k-1}}$$

Consider $VG_{2k} \pmod{x^{2k}}$. V has no terms below degree $k-1$. If we multiply V by any term in G_{2k} of degree k or higher, the resulting degree will be at least $2k-1$, thus we only need the first k terms of the inverse, which is G_k, which we already computed.

$$VG_{2k} \equiv VG_k \pmod{x^{2k-1}}$$

Substituting this back yields:

$$(\theta F_k)G_{2k} \equiv \theta A_k - VG_k \pmod{x^{2k-1}}$$

Applying the inverse differential operator θ^{-1} (equivalent to $F \mapsto \int \frac{F}{x}$) to both sides gives:

$$\ln F_k \equiv A_k - \theta^{-1}(VG_k) \pmod{x^{2k}}$$

Substituting this into the Newton iteration yields the update rule:

$$F_{2k} \equiv F_k + F_k\left(A_{2k} - A_k + \theta^{-1}(VG_k)\right) \pmod{x^{2k}}$$

Algorithm 14: FPS Exponential

Input: Series $A(x)$, target precision N
Output: Series $F(x) \equiv \exp(A(x)) \pmod{x^N}$

```
Function ExpOptimized(A, N):
    F ← 1; G ← 1 ;                                   // G maintains F^{-1}
    k ← 1;
    while k < N do
        V ← ((θA_k)F)_{≥k} (mod x^{2k});
        W ← VG (mod x^{2k});
        C ← A_{2k} − A_k + θ^{-1}(W);
        F ← F + F · C (mod x^{2k});
        G ← G(2 − FG) (mod x^{2k});
        k ← 2k;
    return F;
```

6.6 Power

1. Logarithm and Exponential We have the identity:

$$G = F^k = \exp(k \ln F)$$

To use this identity formally, the constant term of F must be 1 so that $\ln F$ is well-defined. If $F(x) = c_0 + c_1 x + \dots$ where $c_0 \neq 1$, we can factor out the constant term:

$$F^k = c_0^k \exp\left(k \ln\left(\frac{F}{c_0}\right)\right)$$

Note: If the lowest non-zero term of F is $c_m x^m$, we factor out x^{mk}, apply the algorithm to the shifted series, and then shift the result back by mk.

2. Recurrence We can derive a recurrence by differentiating $G = F^k$.

Taking the formal derivative of both sides gives:

$$G' = kF^{k-1}F'$$

Multiplying both sides by F we get:

$$FG' = kGF'$$

Let $F(x) = \sum c_n x^n$ and $G(x) = \sum q_n x^n$. We can extract the coefficient of x^{i-1} on both sides of the equation. On the left side (FG'):

$$[x^{i-1}](FG') = \sum_{j=0}^{i} c_j(i-j)q_{i-j} = c_0 i q_i + \sum_{j=1}^{i} c_j(i-j)q_{i-j}$$

On the right side (kGF'):

$$[x^{i-1}](kGF') = k\sum_{j=0}^{i} q_{i-j}(jc_j) = k\sum_{j=1}^{i} jc_j q_{i-j}$$

Equating the two sides and isolating the q_i term yields:

$$c_0 i q_i = \sum_{j=1}^{i} kjc_j q_{i-j} - \sum_{j=1}^{i}(i-j)c_j q_{i-j}$$

$$q_i = \frac{1}{ic_0}\sum_{j=1}^{i}\Big(kj - (i-j)\Big)c_j q_{i-j}$$

We start with the base case $q_0 = c_0^k$. If $\deg F = d$, the inner summation only runs up to $\min(i, d)$, so the time complexity is $O(Nd)$.

6.7 Division

Given $A(x)$ and $B(x)$, we want to find find the unique quotient $Q(x)$ and remainder $R(x)$ such that:

$$A(x) = Q(x)B(x) + R(x)$$

where $\deg(R) < \deg(B) = m$.

We substitute $x \leftarrow 1/x$ into the division identity and multiply both sides by x^n:

$$x^n A(1/x) = x^{n-m}Q(1/x) \cdot x^m B(1/x) + x^{n-m+1} \cdot x^{m-1}R(1/x)$$

Using the reversal definition, this perfectly rewrites to:

$$A_{\text{rev}}(x) = Q_{\text{rev}}(x)B_{\text{rev}}(x) + x^{d+1}R_{\text{rev}}(x)$$

Taking the equation modulo x^{d+1}:

$$A_{\text{rev}}(x) \equiv Q_{\text{rev}}(x)B_{\text{rev}}(x) \pmod{x^{d+1}}$$

Since $B(x)$ has degree m, its leading coefficient is non-zero. Therefore, the constant term of $B_{\text{rev}}(x)$ is non-zero, guaranteeing that $B_{\text{rev}}(x)$ is invertible. Thus:

$$Q_{\text{rev}}(x) \equiv A_{\text{rev}}(x)B_{\text{rev}}^{-1}(x) \pmod{x^{d+1}}$$

To extract the quotient, we invert $B_{\text{rev}}(x)$ modulo x^{d+1} in $O(n \log n)$. We then multiply it by $A_{\text{rev}}(x)$, truncate to degree d, and reverse the output.

We then compute the remainder as $R(x) = A(x) - Q(x)B(x)$.

Algorithm 15: Division

Function `DivMod`$(A(x), B(x))$**:**
- $n \leftarrow \deg(A)$;
- $m \leftarrow \deg(B)$;
- $d \leftarrow n - m$;
- **if** $d < 0$ **then return** $(0, A(x))$;
- **if** $\min(d, m) \leq$ *THRESHOLD* **then**
  - **return** `NaiveDivMod`$(A(x), B(x))$;
- $A_{\text{rev}}(x) \leftarrow \text{Reverse}(A(x)) \pmod{x^{d+1}}$;
- $B_{\text{rev}}(x) \leftarrow \text{Reverse}(B(x)) \pmod{x^{d+1}}$;
- $B_{\text{inv}}(x) \leftarrow \text{PolyInv}(B_{\text{rev}}(x), d+1)$;
- $Q_{\text{rev}}(x) \leftarrow (A_{\text{rev}}(x) \cdot B_{\text{inv}}(x)) \pmod{x^{d+1}}$;
- $Q(x) \leftarrow \text{Reverse}(Q_{\text{rev}}(x), d)$;
- $R(x) \leftarrow A(x) - (Q(x) \cdot B(x))$;
- **return** $(Q(x), R(x))$;

6.8 Shift

The Taylor shift operation transforms a polynomial $P(x)$ into $P(x + c)$ for some scalar c in $O(n \log n)$ time.

Derivation Let $P(x) = \sum_{i=0}^{n} p_i x^i$. We want to compute the coefficients of $P(x + c)$:

$$P(x + c) = \sum_{i=0}^{n} p_i (x + c)^i$$

We expand $(x + c)^i$:

$$P(x + c) = \sum_{i=0}^{n} p_i \sum_{j=0}^{i} \binom{i}{j} c^{i-j} x^j$$

Expand the binomial coefficient $\binom{i}{j} = \frac{i!}{j!(i-j)!}$ and swap the order of summation to group by the powers of x:

$$P(x + c) = \sum_{j=0}^{n} \frac{x^j}{j!} \sum_{i=j}^{n} (p_i i!) \frac{c^{i-j}}{(i-j)!}$$

Notice that the inner summation evaluates the coefficient of x^j in the shifted polynomial. Let q_j denote this inner sum:

$$q_j = \sum_{i=j}^{n} (p_i i!) \frac{c^{i-j}}{(i-j)!}$$

This is a cross-correlation (or transposed convolution) because the indices in the product subtract to j. To evaluate this using NTT, we convert it into a convolution by reversing one of the sequences.

Let A be the sequence defined by $A_i = p_i i!$, and let A^{rev} be its reversal, so $A^{\text{rev}}_k = A_{n-k}$.

Let B be the sequence defined by $B_k = \frac{c^k}{k!}$.

Substitute $i = n - k$. The limits $i = j \ldots n$ become $k = 0 \ldots n - j$. The indices in our sum become:

$$q_j = \sum_{k=0}^{n-j} A_{n-(j+k)} B_k = \sum_{k=0}^{n-j} A^{\text{rev}}_{n-j-k} B_k$$

Thus q_j is the coefficient of x^{n-j} in the product polynomial $C(x) = A^{\text{rev}}(x)B(x)$.

Once $C(x)$ is computed, we simply read the coefficients backwards and divide by $j!$ to construct the final shifted polynomial.

Algorithm 16: Shift

Input: Polynomial $P(x)$ of degree n, scalar c
Output: Polynomial $P(x+c)$

Function `TaylorShift(`P, c`)`**:**
  $A^{\text{rev}} \leftarrow$ array of size $n+1$;
  for $i \leftarrow 0$ **to** n **do**
    $A^{\text{rev}}[n-i] \leftarrow P[i] \times i!$;
  $B \leftarrow$ array of size $n+1$;
  $c_{pow} \leftarrow 1$;
  for $i \leftarrow 0$ **to** n **do**
    $B[i] \leftarrow \frac{c_{pow}}{i!}$;
    $c_{pow} \leftarrow c_{pow} \times c$;
  $C(x) \leftarrow A^{\text{rev}}(x) \times B(x)$;
  $R \leftarrow$ array of size $n+1$;
  for $j \leftarrow 0$ **to** n **do**
    $R[j] \leftarrow \frac{C[n-j]}{j!}$;
  return $R(x)$;

6.9 Bostan-Mori

The Bostan-Mori algorithm is a computes the N-th coefficient of a rational function $\frac{U(x)}{V(x)}$ in $O(d \log d \log N)$ time, where d is the degree of the denominator.

1. The Bostan-Mori Algorithm To compute $[x^N]\frac{U(x)}{V(x)}$, we use the fact that $V(x)V(-x)$ is an even polynomial, meaning it can be written as $E(x^2)$. We multiply the numerator and denominator by $V(-x)$:

$$\frac{U(x)}{V(x)} = \frac{U(x)V(-x)}{V(x)V(-x)} = \frac{U(x)V(-x)}{E(x^2)}$$

Since the denominator now only contains even powers of x, the parity of the non-zero terms in the expansion is determined by the numerator. If N is even, we only care about the even terms of the numerator. If N is odd, we only care about the odd terms. We can therefore discard half of the terms:

$$[x^N]\frac{U(x)}{V(x)} = \begin{cases} [x^{N/2}]\frac{(U(x)V(-x))_{\text{even}}}{E(x)} & \text{if } N \text{ is even} \\ [x^{(N-1)/2}]\frac{(U(x)V(-x))_{\text{odd}}}{E(x)} & \text{if } N \text{ is odd} \end{cases}$$

By iterating this process, N is halved at each step until $N = 0$, requiring $O(\log N)$ polynomial multiplications of size $O(d)$ as we double and then halve the degree.

Computing $U(x)U(-x)$ We can compute $V(x)$ such that $V(x^2) = U(x)U(-x)$ faster assuming we have access to the NTT of $U(x)$.

In the frequency domain, the array represents the evaluations of $U(x)$ at the $2N$-th roots of unity, $\omega_0, \omega_1, \ldots, \omega_{2N-1}$. Because $\omega_{k+N} = -\omega_k$, evaluating the polynomial at these antipodal points yields $U(\omega_k)$ and $U(-\omega_k)$.

Let $A = U(\omega_k)$ and $B = U(-\omega_k)$. The product of these two evaluations is:

$$A \times B = U(\omega_k)U(-\omega_k) = V(\omega_k^2)$$

ω_k^2 is an N-th root of unity. This means computing the products yields the evaluation of $V(y)$ at the N-th roots of unity.

Bit Reversal:
In bit-reversed order the antipodal roots of unity are next to each other: specifically, the values $U(\omega_k)$ and $U(-\omega_k)$ are stored as pairs $[a, b]$ in the array.

Therefore, we can iterate through the array in chunks of 2 and multiply the adjacent elements.

Algorithm 17: Bostan-Mori

Input: Polynomials $U(x), V(x)$ of degree d, integer N
Output: $[x^N]\frac{U(x)}{V(x)}$

Function `BostanMori(`U, V, N`):`
  while $N > 0$ **do**
    $U_{next}(x) \leftarrow U(x)V(-x)$;
    $V_{next}(x) \leftarrow V(x)V(-x)$;
    if $N \pmod 2 = 0$ **then**
      $U(x) \leftarrow (U_{next})_{\text{even}}$; `// Take even terms:` $x^{2k} \to x^k$
    else
      $U(x) \leftarrow (U_{next})_{\text{odd}}$; `// Take odd terms:` $x^{2k+1} \to x^k$
    $V(x) \leftarrow (V_{next})_{\text{even}}$;
    $N \leftarrow \lfloor N/2 \rfloor$;
  return $U[0]/V[0]$;

6.10 Power Projection

The power projection problem asks us to compute the sequence of inner products $V_k = \langle W, P^k \rangle$ for $k = 0 \dots n-1$. Under the middle product pairing, this is equivalent to extracting the $(n-1)$-th coefficient:

$$V_k = [x^{n-1}]\big(W^R(x)P(x)^k\big)$$

We can construct the generating function for the sequence V:

$$V(y) = \sum_{k=0}^{n-1} V_k y^k = \sum_{k=0}^{n-1} [x^{n-1}] W^R(x)P(x)^k y^k = [x^{n-1}] \frac{W^R(x)}{1 - yP(x)}$$

This formulation reduces the problem to extracting the x^{n-1} coefficient of a bivariate rational function $\frac{A(x,y)}{B(x,y)}$, where initially $A(x,y) = W^R(x)$ and $B(x,y) = 1 - yP(x)$. We can compute this in $O(n \log^2 n)$ time by executing the Bostan-Mori algorithm on the variable x, treating the coefficients as polynomials in y.

1. Handling $P(0) \neq 0$ If $P(0) = c \neq 0$, then $[x^{n-1}] \sum_{k=0}^{\infty} y^k P(x)^k$ is not necessarily equal to $[x^{n-1}] \sum_{k=0}^{n-1} y^k P(x)^k$. To circumvent this, we project onto $\tilde{P}(x) = P(x) - c$, which safely has $\tilde{P}(0) = 0$. Let $\tilde{V}_k$ be this shifted projection sequence. We recover V_k using the binomial expansion:

$$V_k = \langle W, (c + \tilde{P})^k \rangle = \sum_{j=0}^{k} \binom{k}{j} c^{k-j} \tilde{V}_j$$

Dividing by $k!$ yields $\frac{V_k}{k!} = \sum_{j=0}^{k} \frac{c^{k-j}}{(k-j)!} \frac{\tilde{V}_j}{j!}$, which is a convolution of $\exp(cx)$ and the Borel transform of $\tilde{V}$. This shift takes $O(n \log n)$ time.

2. Bostan-Mori on Bivariate Polynomials Assume $P(0) = 0$. We pad our x-degree bound to the nearest power of two, $m = 2^L \geq n$. Because m is a power of 2, the index we are extracting $(m-1)$ has a binary representation consisting entirely of 1s $(\underbrace{11\dots1}_{L})$. In the Bostan-Mori algorithm, we thus extract the odd part of the numerator at every step.

Let $A(x,y) = \sum_{i=0}^{m-1} p_i(y)x^i$ and $B(x,y) = \sum_{i=0}^{m-1} q_i(y)x^i$. Initially, $p_i(y) = w_{m-1-i}$ and $q_i(y) = -P_i y$ (with $q_0(y) = 1$). At each step, we compute:

$$A_{new}(x,y) = \mathcal{O}_x\big(A(x,y)B(-x,y)\big)$$
$$B_{new}(x,y) = \mathcal{E}_x\big(B(x,y)B(-x,y)\big)$$

Because the coefficients are polynomials in y, we can compute the NTT on each row and column to compute the convolution, and we can avoid duplicating the polynomials and apply various constant factor optimizations.

Complexity At each step, we perform $O(m)$ NTTs of size k over y, which takes $O(m \cdot k \log k)$ time. We then perform $O(k)$ NTTs of size m over x, taking $O(k \cdot m \log m)$ time. Because the invariant $m \cdot k = O(n)$ is preserved at every step, the time spent per step simplifies to:

$$O(n \log k + n \log m) = O(n \log(k \cdot m)) = O(n \log n)$$

Since m halves at each step, there are $\log_2(n)$ steps. Thus, the overall complexity is $O(n \log^2 n)$.

Algorithm 18: Power Projection

Function `PowerProj`$(W(x), P(x), n)$**:**
- **if** $n = 0$ **then return** 0;
- $c \leftarrow P[0]$;
- **if** $c \neq 0$ **then**
  - $\tilde{P}(x) \leftarrow P(x) - c$;
  - $\tilde{V}(y) \leftarrow$ `PowerProj`$(W(x), \tilde{P}(x), n)$;
  - **for** $j \leftarrow 0$**to**$n - 1$ **do**
    - $\tilde{V}_{\text{borel}}[j] \leftarrow \tilde{V}[j] \cdot (j!)^{-1} \pmod{M}$;
  - $E(y) \leftarrow \sum_{i=0}^{n-1} c^i \cdot (i!)^{-1} y^i$;
  - $V_{\text{borel}}(y) \leftarrow (\tilde{V}_{\text{borel}}(y) \cdot E(y)) \pmod{y^n}$;
  - **for** $k \leftarrow 0$**to**$n - 1$ **do**
    - $V[k] \leftarrow V_{\text{borel}}[k] \cdot (k!) \pmod{M}$;
  - **return** $V(y)$;
- $m \leftarrow$ next_power_of_2(n);
- $k \leftarrow 1$;
- $A(x, y) \leftarrow$ Pad_to_Degree(Reverse$(W(x)), m - 1)$;
- $B(x, y) \leftarrow 1 - yP(x)$;
- **while** $m > 1$ **do**
  - $B_{\text{neg}}(x, y) \leftarrow B(-x, y)$;
  - $A_{\text{next}}(x, y) \leftarrow \mathcal{O}_x\big(A(x, y) \cdot B_{\text{neg}}(x, y)\big)$;
  - $B_{\text{next}}(x, y) \leftarrow \mathcal{E}_x\big(B(x, y) \cdot B_{\text{neg}}(x, y)\big)$;
  - $A(x, y) \leftarrow A_{\text{next}}(x, y)$;
  - $B(x, y) \leftarrow B_{\text{next}}(x, y)$;
  - $m \leftarrow m/2$;
  - $k \leftarrow 2k$;
- **return** $A(0, y) \pmod{y^n}$;

6.11 Transposition Principle

The transposition principle (as used in algorithms on formal power series) states that if a linear map $f : \mathbb{K}^n \to \mathbb{K}^m$ can be computed efficiently, then so can its transposed linear map $f^T : \mathbb{K}^m \to \mathbb{K}^n$.

Let V and W be vector spaces over $\mathbb{K}$, and let $\mathcal{L} : V \to W$ be a linear operator, we consider a bilinear form $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{K}$. The transposed operator $\mathcal{L}^T : W \to V$ is defined as the operator that satisfies the identity:

$$\langle \mathcal{L}(u), v \rangle_W = \langle u, \mathcal{L}^T(v) \rangle_V$$

By taking an algorithm and breaking it down into a composition of linear operators $\mathcal{L} = \mathcal{L}_k \circ \cdots \circ \mathcal{L}_1$, we can find the adjoint of the algorithm by composing the adjoints of the steps in reverse order: $\mathcal{L}^T = \mathcal{L}_1^T \circ \cdots \circ \mathcal{L}_k^T$.

We primarily use two pairings:

The Dot-Product: We can define our pairing over polynomials of degree $< k$ as the dot-product.

$$\langle A, B \rangle_k = \sum_{i=0}^{k-1} a_i b_i$$

The Middle Product: We can also use the middle product pairing. For $U, A \in \mathbb{K}[x]_{<d}$:

$$\langle U, A \rangle = [x^{d-1}]\big(U(x)A(x)\big) = \sum_{i=0}^{d-1} u_i a_{d-1-i}$$

6.12 Transposed Multiplication

Dot Product Pairing

Lemma 6.1. *Let $\mathcal{M}_P(Q) = (P(x)Q(x)) \bmod x^k$ denote multiplication by a fixed polynomial $P(x)$. Its transposed operator $\mathcal{M}_P^T$, known as the Middle Product or Transposed Multiplication (denoted $\circ$), is given by:*

$$\mathcal{M}_P^T(A) = A \circ P = \sum_{j=0}^{k-1} x^j \sum_{\ell=0}^{k-1-j} a_{j+\ell} p_\ell$$

Proof. We seek the adjoint such that $\langle \mathcal{M}_P(Q), A\rangle_k = \langle Q, \mathcal{M}_P^T(A)\rangle_k$. Expanding the left side:

$$\langle P \cdot Q \bmod x^k, A\rangle_k = \sum_{i=0}^{k-1} a_i \sum_{j=0}^{i} q_j p_{i-j}$$

We swap the order of summation to isolate the coefficients of Q:

$$\sum_{j=0}^{k-1} q_j \sum_{i=j}^{k-1} a_i p_{i-j}$$

Let $\ell = i - j$. The inner sum becomes $\sum_{\ell=0}^{k-1-j} a_{j+\ell} p_\ell$. This represents the j-th coefficient of the required transposed state, matching the definition of the middle product $A \circ P$. □

Middle Product Pairing

Lemma 6.2. *Let the pairing over $\mathbb{K}[x]_{<k}$ be defined by: $\langle A, B\rangle_k = [x^{k-1}]\big(A(x)B(x)\big)$. The adjoint of polynomial multiplication by a fixed polynomial $P(x)$ corresponds to multiplication by $P(x)$ followed by a degree right-shift.*

Specifically, if the forward operator maps an input of degree $< n$ to an output space of degree $< N$, the transposed operator applied to a state $Z(x)$ is given by:

$$\mathcal{M}_P^T(Z(x)) = \left\lfloor \frac{P(x)Z(x)}{x^{N-n}} \right\rfloor \pmod{x^n}$$

Proof. Let $\mathcal{M}_P : \mathbb{K}[x]_{<n} \to \mathbb{K}[x]_{<N}$ be the forward operator defined by $U(x) \mapsto U(x)P(x)$. We seek the transposed state $A(x) = \mathcal{M}_P^T(Z(x)) \in \mathbb{K}[x]_{<n}$ given an incoming transposed state $Z(x) \in \mathbb{K}[x]_{<N}$. By the definition of the adjoint, the following must hold for any polynomial $U(x) \in \mathbb{K}[x]_{<n}$:

$$\langle \mathcal{M}_P(U), Z\rangle_N = \langle U, A\rangle_n$$

Expanding both sides using the definition of our pairing yields:

$$[x^{N-1}]\big(U(x)P(x)Z(x)\big) = [x^{n-1}]\big(U(x)A(x)\big)$$

We factor x^{N-n} out of the left-hand expression:

$$[x^{n-1}]\left(U(x)\frac{P(x)Z(x)}{x^{N-n}}\right) = [x^{n-1}]\big(U(x)A(x)\big)$$

Since $U(x)$ is arbitrary, the coefficients of the factor must be identical on both sides for degrees 0 through $n-1$.

Any fractional power or power $\geq n$ will result in a negative or overly large degree when multiplied by $U(x)$, so they can be discarded using floor and modulo.

Therefore, the transposed state $A(x)$ is the range of valid degrees:

$$A(x) = \left\lfloor \frac{P(x)Z(x)}{x^{N-n}} \right\rfloor \pmod{x^n}$$

□

6.13 Transposed NTT and INTT

If an implementation does not do bit reversal then the transforms are self-adjoint.

Let n be a power of 2, and $\mathrm{rev}(k)$ as the bit-reversal of k up to the $\log_2(n)$-th digit. Consider bit-reversed FFT and bit-reversed IFFT:

- **Bit-reversed FFT:** Input a and output b defined by:

$$b_i = \sum_j \omega^{-\mathrm{rev}(i)\cdot j} a_j$$

- **Bit-reversed IFFT:** Input b and output a defined by:

$$a_i = \frac{1}{n} \sum_j \omega^{i\cdot \mathrm{rev}(j)} b_j$$

Their transposes are:

- **Transposed Bit-reversed FFT:** Input b and output a defined by:

$$a_i = \sum_j \omega^{-i \cdot \mathrm{rev}(j)} b_j$$

- **Transposed Bit-reversed IFFT:** Input a and output b defined by:

$$b_i = \frac{1}{n} \sum_j \omega^{\mathrm{rev}(i) \cdot j} a_j$$

"Bit reversed FFT" and "transposed bit reversed IFFT", and "bit reversed IFFT" and "transposed bit reversed FFT", are essentially the same.

The operation of reversing the second to last elements sends the k-th element to the $(n-k)$-th position, and $\omega^{n-k} = \omega^{-k}$, so the exponent is negated. Therefore:

- The **transposed bit-reversed FFT** is: bit-reversed IFFT, then reversing all but the first element, then scaling it by n.
- The **transposed bit-reversed IFFT** is: reversing all but the first element, then bit-reversed FFT, then scaling it by $1/n$.

Transposed Multiplication Using these exact transposed operators, we can compute transposed multiplication:

$$\mathrm{Pad}_n(Z) \xrightarrow{\texttt{intt\_t}} \cdot \odot \mathrm{NTT}(V) \xrightarrow{\texttt{ntt\_t}} \mathrm{Truncate}_{m-k+1}$$

The difference between doing this and computing the middle product by reversing is that we compute less coefficients of the output and thus it's faster if less are needed.

6.14 Transposed Bostan-Mori

The Bostan-Mori algorithm evaluates the linear functional $U(x) \mapsto [x^N]\frac{U(x)}{V(x)}$ for $U \in \mathbb{K}[x]_{<d}$. The transposed algorithm computes a polynomial $A(x) \in \mathbb{K}[x]_{<d}$ under a chosen inner product.

We use the middle product pairing. The function `TransposedProj` computes the polynomial $A(x)$ such that for any $U(x)$ of degree $< d$:

$$[x^{d-1}]U(x)A(x) = [x^N]\frac{U(x)}{V(x)}$$

1. Recovering the Remainder To justify the correctness of the algorithm, we prove that computing this transposed state $A(x)$ is equivalent to computing $x^N \pmod{P(x)}$.

Theorem 6.3. *Let $P(x) = V^R(x)$. If $R(x) = x^N \pmod{P(x)}$, then $A(x) \equiv \frac{R^R(x)}{V(x)} \pmod{x^d}$.*

Proof. By polynomial division, $x^N = Q(x)P(x) + R(x)$. Substituting $x \leftarrow 1/x$ yields:

$$x^{-N} = Q(1/x)P(1/x) + R(1/x)$$

Notice that $V(x) = x^d P(1/x) \implies P(1/x)/V(x) = x^{-d}$. We divide our equation by $V(x)$ and multiply by $U(x)$:

$$x^{-N}\frac{U(x)}{V(x)} = x^{-d}Q(1/x)U(x) + \frac{R(1/x)U(x)}{V(x)}$$

We extract the $[x^0]$ coefficient from both sides. On the left, $[x^0]x^{-N}\frac{U(x)}{V(x)} = [x^N]\frac{U(x)}{V(x)}$.
On the right side, the first term vanishes: $Q(1/x)$ has maximum degree 0, and $U(x)$ has maximum degree $d-1$, so $x^{-d}Q(1/x)U(x)$ has strictly negative powers of x, yielding an $[x^0]$ coefficient of 0. We are left with:

$$[x^N]\frac{U(x)}{V(x)} = [x^0]\frac{R(1/x)U(x)}{V(x)}$$

Since $\deg(R) < d$, its reversal is $R^R(x) = x^{d-1}R(1/x)$. Substituting $R(1/x) = x^{-(d-1)}R^R(x)$ gives:

$$[x^0]x^{-(d-1)}\frac{R^R(x)U(x)}{V(x)} = [x^{d-1}]U(x)\frac{R^R(x)}{V(x)}$$

By equating this with our pairing definition $[x^{d-1}]U(x)A(x)$, it follows immediately that:

$$A(x) \equiv \frac{R^R(x)}{V(x)} \pmod{x^d} \implies R^R(x) \equiv A(x)V(x) \pmod{x^d}$$

□

2. Transposing the Recurrence We now derive the recurrence for the state polynomial $A(x)$ by transposing the two linear operators of the forward step: polynomial multiplication by $V(-x)$, and the extraction of the Even/Odd terms.

Lemma 6.4. *Let $\mathcal{E}$ and $\mathcal{O}$ denote the operators that extract the even and odd terms of a polynomial, respectively. Their adjoints under the middle product pairing correspond to upsampling (interleaving zeroes), with a parity shift that swaps them: $\mathcal{E}^T(A) = xA(x^2)$ and $\mathcal{O}^T(A) = A(x^2)$.*

Proof. Suppose N is even, meaning the forward step applies $U_{next} = \mathcal{E}(W)$. We seek a polynomial $Z(x)$ such that $\langle \mathcal{E}(W), A\rangle_{d-1} = \langle W, Z\rangle_{2d-1}$.

Expanding the left side yields:

$$\langle \mathcal{E}(W), A\rangle_{d-1} = [x^{d-1}]\mathcal{E}(W(x))A(x) = \sum_{i=0}^{d-1} w_{2i}a_{d-1-i}$$

Consider $Z(x) = xA(x^2)$. Because its non-zero terms are strictly at odd powers, its middle product with $W(x)$ at degree $2d-1$ filters out the odd coefficients of W:

$$[x^{2d-1}]W(x)Z(x) = \sum_k w_k[x^{2d-1-k}]xA(x^2)$$

For the Z coefficient to be non-zero, the power $2d-1-k$ must be odd, which strictly forces k to be even. Let $k = 2i$. The corresponding coefficient is $[x^{2d-2-2i}]A(x^2) = a_{d-1-i}$. Thus, the summation matches the left side, proving that $\mathcal{E}^T(A) = xA(x^2)$.

By identical logic, if N is odd, the forward step uses $U_{next} = \mathcal{O}(W)$. The required adjoint $Z(x)$ does not need the $+1$ degree shift to align the terms, yielding $\mathcal{O}^T(A) = A(x^2)$. □

3. Base Case ($N = 0$) When $N = 0$, the forward algorithm evaluates to $\frac{U(0)}{V(0)}$. We need $[x^{d-1}]U(x)A(x) = \frac{U(0)}{V(0)}$. This holds if and only if $A(x) = \frac{1}{V(0)}x^{d-1}$.

Algorithm 19: Transposed Bostan-Mori

```
Function TransposedProj(N, Q(x), d):
    if N = 0 then
        R(x) ← (1/Q[0]) x^{d-1};
        return R(x);
    Q_next(x) ← (Q(x)Q(−x))_even;
    p(x) ← TransposedProj(⌊N/2⌋, Q_next(x), d);
    if N (mod 2) = 0 then
        p_ext(x) ← x · p(x^2);
    else
        p_ext(x) ← p(x^2);
    p_new(x) ← p_ext(x) · Q(−x);
    p_ret(x) ← (p_new(x) ≫ d) (mod x^d) ;        // Keep terms x^d ... x^{2d-1}
    return p_ret(x);

Function XiMod(P(x), N):
    d ← deg(P);
    if N < d then return x^N;
    Q(x) ← Reverse(P(x));
    A(x) ← TransposedProj(N, Q, d);
    R^R(x) ← (A(x) · Q(x)) (mod x^d);
    return Reverse(R^R(x));
```

6.15 Composition

The problem of computing $C(x) = F(P(x)) \pmod{x^n}$ can be solved in $O(n \log^2 n)$ by transposing the algorithm for Power Projection.

Let $\mathbf{M}_P$ be an $m \times m$ transformation matrix where the k-th column represents the coefficients of $P(x)^k \pmod{x^m}$.

- **Composition:** Evaluating $F(P(x))$ equates to taking a linear combination of the powers of $P(x)$ scaled by the coefficients of F. This is $\mathbf{c} = \mathbf{M}_P\mathbf{f}$.
- **Power Projection:** Transposing this map asks us to evaluate $\mathbf{v} = \mathbf{M}_P^T\mathbf{w}$. This corresponds to taking the dot product of a vector $\mathbf{w}$ with every column of $\mathbf{M}_P$, yielding the sequence $V_k = [x^{m-1}]W^R(x)P(x)^k$.

1. Handling $P(0) \neq 0$ If $P(0) = c \neq 0$, we must shift the polynomial to $\tilde{P}(x) = P(x) - c$ so that $\tilde{P}(0) = 0$.

We simultaneously shift the polynomial $F(x)$ using a Taylor shift to compute $F_{\text{shift}}(x) = F(x + c)$. The original composition is recovered by evaluating $F_{\text{shift}}(\tilde{P}(x))$.

2. Transposed Odd Extraction Because we pad the polynomials to a power of two n, our target coefficient is exactly $[x^{n-1}]$. Since $n - 1$ in binary is composed entirely of 1s, the forward Power Projection must extract the odd terms at every recursive layer.

Let $\mathcal{O}_x$ denote the extraction of odd terms: $\mathcal{O}_x(U(x)) = \frac{U(x)-U(-x)}{2x}$. In the forward pass, a state $A(x)$ is multiplied by the denominator $Q_s(-x)$ and projected via:

$$A_{next}(x^2) = \mathcal{O}_x\big(A(x)Q_s(-x)\big) = \frac{A(x)Q_s(-x) - A(-x)Q_s(x)}{2x}$$

We execute the adjoint directly in the frequency domain. Evaluating this odd extraction at a root of unity ω combines the two frequencies ω and $-\omega$:

$$A_{next}(\omega^2) = \frac{1}{2\omega}\Big(A(\omega)Q_s(-\omega) - A(-\omega)Q_s(\omega)\Big)$$

This is a linear map from the tuple $(A(\omega), A(-\omega))$ to $A_{next}(\omega^2)$. The adjoint operator is:

$$A^*(\omega) = P(\omega^2) \cdot \frac{1}{2\omega} Q_s(-\omega)$$
$$A^*(-\omega) = P(\omega^2) \cdot \frac{-1}{2\omega} Q_s(\omega)$$

In a bit-reversed NTT array, the frequencies ω and $-\omega$ sit at adjacent indices $2i$ and $2i + 1$. Therefore, we get $P(\omega^2)$ from index i, compute $c = P[i] \cdot \frac{1}{2}\omega^{-1}$, and write:

$$A^*[2i] = c \cdot Q_s[2i + 1]$$
$$A^*[2i + 1] = -c \cdot Q_s[2i]$$

Algorithm 20: Composition

Function `Compose`$(F(x), P(x), m)$**:**
  if $m = 0$ **then return** 0;
  $c \leftarrow P[0]$;
  if $c \neq 0$ **then**
    $\tilde{P}(x) \leftarrow P(x) - c$;
    $F_{\text{shift}}(x) \leftarrow$ `TaylorShift`$(F(x), c)$;
    return `Compose`$(F_{\text{shift}}(x), \tilde{P}(x), m)$;
  $n \leftarrow$ next_power_of_2(m);
  Pad $F(x)$ and $P(x)$ with zeroes to degree $n - 1$;
  $V \leftarrow$ BitReversedInverseRoots$(2n)$;
  $Q(x) \leftarrow 1 - yP(x)$;
  $A(x) \leftarrow$ `TransposedPowerProj`$(n, 1, Q(x), F(x), V)$;
  return Truncate$_{m-1}$(Reverse$(A(x))$);

Function `TransposedPowerProj`$(n, k, Q(x), F(x), V)$**:**
  if $n = 1$ **then**
    $A \leftarrow$ array of size $2k$ initialized to 0;
    $F_{\text{rev}} \leftarrow$ Reverse(F);
    for $i \leftarrow 0$ **to** $k - 1$ **do**
      $A[2i] \leftarrow F_{\text{rev}}[i]$;
    return A;
  $Q_{\text{freq}} \leftarrow \text{NTT}_x(Q)$;
  $Q_{\text{next}} \leftarrow \text{INTT}_x\big(\mathcal{E}_x(Q_{\text{freq}} \cdot Q_{\text{freq}}(\text{negated}))\big)$;
  $P \leftarrow$ `TransposedPowerProj`$(n/2, 2k, Q_{\text{next}}, F, V)$;
  $P_{\text{freq}} \leftarrow \text{INTT}_x^T(P)$;
  $A_{\text{freq}} \leftarrow$ array of size $2nk$;
  for $j \leftarrow 0$ **to** $k - 1$ **do**
    for $i \leftarrow 0$ **to** $\frac{n}{2} - 1$ **do**
      $idx \leftarrow j \cdot n + i$;
      `// V[i] represents the` ω^{-1} `factor`
      $c \leftarrow P_{\text{freq}}[idx] \cdot \frac{1}{2} \cdot V[i]$;
      $A_{\text{freq}}[2 \cdot idx] \leftarrow c \cdot Q_{\text{freq}}[2 \cdot idx + 1]$;
      $A_{\text{freq}}[2 \cdot idx + 1] \leftarrow -c \cdot Q_{\text{freq}}[2 \cdot idx]$;
  $A_{\text{final}} \leftarrow \text{NTT}_x^T(A_{\text{freq}})$;
  return A_{final};

6.16 Lagrange Inversion

Lemma 6.5 (Lagrange Inversion)**.** *For formal power series f, Φ, and H, assume $f(x) = x\Phi(f(x))$. Then:*

$$[x^n]H(f(x)) = \frac{1}{n}[x^{n-1}]H'(x)\Phi(x)^n$$

Proof. By the linearity of the coefficient extraction operator, we can assume without loss of generality that $H(x) = x^k$. We then need to show:

$$[x^n]f(x)^k = \frac{1}{n}[x^{n-1}]kx^{k-1}\Phi(x)^n = \frac{k}{n}[x^{n-k}]\Phi(x)^n$$

We proceed by induction on n.

Base Cases: The cases $n = 0$ and $k = 0$ are trivial. If $k > n$, the claim is trivially true: since $f(0) = 0$, the lowest non-zero term of $f(x)^k$ has degree k, making $[x^n]f(x)^k = 0$. Since $n - k < 0$, the right-hand side is also exactly 0. If $k = n$, then $n[x^n]A^k(x) = n[x^0]\Phi^n(A(x)) = n(c_0)^n = k[z^{n-k}]\Phi^n(x)$.

Inductive Step: Assume the theorem holds for indices strictly less than n. Using the relation $f(x) = x\Phi(f(x))$, we pull out the x^k:

$$\begin{aligned}[x^n]f(x)^k &= [x^n]x^k\Phi(f(x))^k \\ &= [x^{n-k}]\Phi(f(x))^k\end{aligned}$$

We now apply the induction hypothesis to evaluate the $(n - k)$-th coefficient of the composite function $\Phi(f(x))^k$.

Treating $\Phi(x)^k$ as our new $H(x)$, we have:

$$\begin{aligned}[x^{n-k}]\Phi(f(x))^k &= \frac{1}{n-k}[x^{n-k-1}]\left(\Phi(x)^k\right)'\Phi(x)^{n-k} \\ &= \frac{1}{n-k}[x^{n-k-1}]\left(k\Phi'(x)\Phi(x)^{k-1}\right)\Phi(x)^{n-k} \\ &= \frac{k}{n-k}[x^{n-k-1}]\Phi'(x)\Phi(x)^{n-1}\end{aligned}$$

We rewrite $\Phi'(x)\Phi(x)^{n-1}$ using the chain rule in reverse:

$$\begin{aligned}\frac{k}{n-k}[x^{n-k-1}]\Phi'(x)\Phi(x)^{n-1} &= \frac{k}{n-k}[x^{n-k-1}]\frac{1}{n}\left(\Phi(x)^n\right)' \\ &= \frac{1}{n}\frac{k}{n-k}[x^{n-k-1}]\left(\Phi(x)^n\right)'\end{aligned}$$

Finally, using the identity $[x^{m-1}]P'(x) = m[x^m]P(x)$ with $m = n-k$, we evaluate:

$$\begin{aligned}\frac{1}{n}\frac{k}{n-k}[x^{n-k-1}]\left(\Phi(x)^n\right)' &= \frac{1}{n}\frac{k}{n-k}(n-k)[x^{n-k}]\Phi(x)^n \\ &= \frac{k}{n}[x^{n-k}]\Phi(x)^n\end{aligned}$$

□

Lemma 6.6 (Lagrange Inversion for Compositional Inverse)**.** *Let $F(x) = \sum_{r\geq 1} f_r x^r$ be a formal power series where f_1 is invertible, and let $F^{\langle -1\rangle}(x)$ denote its compositional inverse such that $F(F^{\langle -1\rangle}(x)) = x$. For any formal power series $H(x)$ and any integer $n \geq 1$:*

$$[x^n]H(F^{\langle -1\rangle}(x)) = \frac{1}{n}[x^{n-1}]H'(x)\left(\frac{x}{F(x)}\right)^n$$

Proof. To apply Lemma 6.5, we define $\Phi(x) := \frac{x}{F(x)}$. Because $f_1 \neq 0$, $\Phi(x)$ is a well-defined formal power series with a non-zero constant term $\Phi(0) = 1/f_1$.

Let $A(x) := F^{\langle -1\rangle}(x)$. By the definition of the compositional inverse, we know that $F(A(x)) = x$.

We evaluate our function Φ at $A(x)$:

$$\Phi(A(x)) = \frac{A(x)}{F(A(x))} = \frac{A(x)}{x}$$

Multiplying both sides by x, we recover the necessary functional equation:

$$A(x) = x\Phi(A(x))$$

This satisfies the hypothesis of Lemma 6.5 with $f(x) = A(x)$. We can therefore apply it to recover the coefficients of the composite function $H(A(x))$:

$$[x^n]H(A(x)) = \frac{1}{n}[x^{n-1}]H'(x)\Phi(x)^n$$

Substituting $A(x) = F^{\langle -1\rangle}(x)$ and our definition of $\Phi(x) = \frac{x}{F(x)}$ back into the relation yields:

$$[x^n]H(F^{\langle -1\rangle}(x)) = \frac{1}{n}[x^{n-1}]H'(x)\left(\frac{x}{F(x)}\right)^n$$

□

6.17 Compositional Inverse

Let $F(x)$ be a formal power series such that $F(0) = 0$ and $F'(0) \neq 0$. The compositional inverse of $F(x)$ is the unique formal power series $G(x)$ satisfying $F(G(x)) = x$.

1. Reduction to $F'(0) = 1$ We can assume without loss of generality that $F'(0) = 1$. If $F'(0) = c \neq 1$, we can scale the series by defining $\tilde{F}(x) = c^{-1}F(x)$. Once we find its compositional inverse $\tilde{G}(x)$ such that $\tilde{F}(\tilde{G}(x)) = x$, we have $c^{-1}F(\tilde{G}(x)) = x$, meaning $F(\tilde{G}(x)) = cx$. By substituting $x \mapsto c^{-1}x$, we obtain $F(\tilde{G}(c^{-1}x)) = x$, which gives $G(x) = \tilde{G}(c^{-1}x)$.

2. Lagrange Inversion Assume $F(x) = x + f_2x^2 + \dots$, and let $G(x)$ be its compositional inverse. By Lemma 6.6, we can relate the powers of $F(x)$ to the coefficients of $G(x)$. Specifically, for any k:

$$[x^n]F(x)^k = \frac{k}{n}[x^{n-k}]\left(\frac{x}{G(x)}\right)^n$$

Let us define an auxiliary series $H(x) = \frac{x}{G(x)}$. Our goal is to compute the polynomial $H(x)^n \pmod{x^n}$. Let its coefficients be $H(x)^n = \sum_{j=0}^{n-1} u_j x^j$. From the inversion formula, setting $j = n - k$ (so $k = n - j$), we can isolate u_j:

$$u_j = [x^j]H(x)^n = \frac{n}{n-j}[x^n]F(x)^{n-j}$$

3. Power Projection If we define the target vector $W(x) = x^n$, computing the power projection yields the sequence $v_k = [x^n]F(x)^k$ for $k = 0 \dots n$ in $O(n \log^2 n)$ time.

Once we have the sequence v_k, we can construct $H(x)^n$ in linear time by scaling and reversing the sequence:

$$u_j = \frac{n}{n-j}v_{n-j}$$

Finally, recovering $G(x)$ requires taking the inverse power of $H(x)^n$:

$$H(x)^{-1} = (H(x)^n)^{-1/n}$$

Since $H(x) = x/G(x)$, we recover $G(x) = x \cdot H(x)^{-1}$.

Algorithm 21: Compositional Inverse

```
Function CompInverse(F(x), n):
    if n = 0 then return 0;
    c ← F[1];
    F_monic(x) ← c^{-1} F(x);
    W(x) ← x^n;
    // V[k] = [x^n] F_monic(x)^k  for k ∈ [0, n]
    V ← PowerProjection(F_monic(x), W(x), n);
    A ← empty array of size n;
    for k ← 0 to n − 1 do
        A[k] ← n · V[k + 1] · (k + 1)^{-1};
    H_n(x) ← Reverse(A) ;                          // H_n[j] = A[n − 1 − j]
    P ← −n^{-1} (mod M);
    H_inv(x) ← PolyPow(H_n(x), P, n);
    G̃(x) ← x · H_inv(x);
    for k ← 1 to n do
        G[k] ← G̃[k] · c^{-k};
    return G(x);
```

6.18 Multipoint Evaluation

Multipoint evaluation computes the values of a polynomial $F(x) \in \mathbb{K}[x]_{<n}$ at n distinct points $\alpha_1, \dots, \alpha_n$. This linear map corresponds to multiplying the Vandermonde matrix $V(\alpha)$ by the coefficient vector of $F(x)$. We construct an algorithm for the transposed problem, and then transpose it.

The transposed problem multiplies the transposed Vandermonde matrix $V(\alpha)^T$ by a vector $c = (c_1, \dots, c_n)^T$. The j-th entry of the output is $\sum_{i=1}^{n} c_i\alpha_i^j$, which represents the first n coefficients of the rational function power series:

$$R(x) = \sum_{i=1}^{n} \frac{c_i}{1 - \alpha_i x} \pmod{x^n}$$

1. The Algorithm for the Transposed Problem To compute $R(x)$ efficiently, we use a bottom-up divide-and-conquer approach on a subproduct tree. For any node u representing a subset of points S_u, we maintain a numerator $N_u(x)$ and a denominator $D_u(x) = \prod_{i \in S_u}(1 - \alpha_i x)$.

At the leaves, $N_{\{i\}}(x) = c_i$ and $D_{\{i\}}(x) = 1 - \alpha_i x$. For an internal node u with children L and R, we combine them as:

$$N_u(x) = N_L(x)D_R(x) + N_R(x)D_L(x), \quad D_u(x) = D_L(x)D_R(x)$$

At the root, the final output is obtained by computing $N_{root}(x) \cdot D_{root}(x)^{-1} \pmod{x^n}$.

Since $D_u(x)$ depends only on the fixed evaluation points α_i, it is treated as a constant in the linear map.

2. Transposing the Recurrence We use the standard dot-product. In the forward transposed problem, the root step is $S(x) = N_{root}(x) \cdot D_{root}(x)^{-1} \pmod{x^n}$. By applying the lemma, if the input to the multipoint evaluation is $F(x)$ (which acts as the dual to $S(x)$), the transposed state at the root is:

$$T_{root}(x) = F(x) \circ D_{root}^{-1}(x)$$

During the upward combination phase, a node u computes $N_u = N_L D_R + N_R D_L$. Because the inner product is linear, transposing this addition distributes the middle product to the children. The transposed states passed down from u to L and R are:

$$T_L(x) = T_u(x) \circ D_R(x) \quad \text{and} \quad T_R(x) = T_u(x) \circ D_L(x)$$

At the base case (the leaves), the forward step initializes $N_{\{i\}} = c_i$. The transposed operator simply extracts the degree-0 coefficient of the leaf state. Thus, the evaluated value $F(\alpha_i)$ is identically $[x^0]T_{\{i\}}(x)$.

Algorithm 22: Multipoint Evaluation

```
Function FastMultipointEval(F(x), α_1, ..., α_n):
    D ← PrecomputeSubproductTree(α_1, ..., α_n);
    // D_root(x) = ∏(1 − α_i x)
    I_root(x) ← D_root(x)^{-1} (mod x^n);
    T_root(x) ← TransposedMul(F(x), I_root(x));
    Y ← array of size n;
    EvalTopDown(root, T_root(x), D, Y);
    return Y;

Function EvalTopDown(u, T_u(x), D, Y):
    if u is a leaf for point α_i then
        Y[i] ← T_u[0];
        return;
    T_L(x) ← TransposedMul(T_u(x), D_{R.child}(x));
    T_R(x) ← TransposedMul(T_u(x), D_{L.child}(x));
    EvalTopDown(L.child, T_L(x), D, Y);
    EvalTopDown(R.child, T_R(x), D, Y);
```

6.19 Chirp-Z

When the evaluation points form a geometric progression $\alpha_j = z^j$ for $j = 0, \ldots, k-1$, we can compute the multipoint evaluation in $O(n \log n)$.

To evaluate $F(x) = \sum_{i=0}^{d-1} p_i x^i$ at z^j, we substitute $x = z^j$:

$$F(z^j) = \sum_{i=0}^{d-1} p_i z^{ij}$$

We apply the identity $ij = \binom{i+j}{2} - \binom{i}{2} - \binom{j}{2}$:

$$F(z^j) = \sum_{i=0}^{d-1} p_i z^{\binom{i+j}{2}} z^{-\binom{i}{2}} z^{-\binom{j}{2}} = z^{-\binom{j}{2}} \sum_{i=0}^{d-1} \left(p_i z^{-\binom{i}{2}}\right) z^{\binom{i+j}{2}}$$

Notice that the inner summation is the Middle Product of $F_{scaled}(x)$ with a sequence $B(x)$:

$$F_{scaled}(x) = \sum_{i=0}^{d-1} \left(p_i z^{-\binom{i}{2}}\right) x^i, \quad B(x) = \sum_{m=0}^{k+d-1} z^{\binom{m}{2}} x^m$$

The evaluation over all k points is given by $Y = (F_{scaled}(x) \circ B(x)) \pmod{x^k}$, followed by a scaling by $z^{-\binom{j}{2}}$.

Algorithm 23: Chirp-Z

Function `ScaleQBinomial`$(F(x), a)$**:**
  `// Multiplies the` i`-th coefficient by` $a^{\binom{i}{2}}$ `in` $O(n)$
  $d \leftarrow \deg(F)$;
  $cur \leftarrow 1$; $total \leftarrow 1$;
  for $i \leftarrow 0$ **to** d **do**
    $F[i] \leftarrow (F[i] \cdot total) \pmod M$;
    $total \leftarrow (total \cdot cur) \pmod M$;
    $cur \leftarrow (cur \cdot a) \pmod M$;
  return $F(x)$;

Function `ChirpZ`$(F(x), z, k)$**:**
  $d \leftarrow \deg(F)$;
  $z_{inv} \leftarrow z^{-1} \pmod M$;
  $B(x) \leftarrow$ Polynomial of $(k + d)$ ones;
  $B(x) \leftarrow$ `ScaleQBinomial`$(B(x), z)$;
  $F_{scaled}(x) \leftarrow$ `ScaleQBinomial`$(F(x), z_{inv})$;
  $Y(x) \leftarrow$ `TransposedMul`$(B(x), F_{scaled}(x)) \pmod{x^k}$;
  $Y(x) \leftarrow$ `ScaleQBinomial`$(Y(x), z_{inv})$;
  return $Y(x)$;

6.20 Interpolation

Polynomial interpolation finds the unique polynomial $F(x) \in \mathbb{K}[x]_{<n}$ such that $F(\alpha_i) = y_i$ for n distinct points $\alpha_1, \ldots, \alpha_n$. By the Lagrange interpolation formula:

$$F(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j}$$

Let $D(x) = \prod_{i=1}^{n}(x - \alpha_i)$. We have $D'(\alpha_i) = \prod_{j \neq i}(\alpha_i - \alpha_j)$. Thus, the formula is:

$$F(x) = \sum_{i=1}^{n} \frac{y_i}{D'(\alpha_i)} \frac{D(x)}{x - \alpha_i}$$

Thus we need to solve two problems:

1. Compute the values $v_i = D'(\alpha_i)$ for all $i = 1, \ldots, n$.
2. Compute the coefficients $c_i = y_i / v_i$, and evaluate $\sum_{i=1}^{n} c_i \prod_{j \neq i}(x - \alpha_j)$.

1. The Subproduct Tree Both tasks rely on a subproduct tree over the points α_i. For any node u corresponding to a subset of points S_u, we consider $D_u(x) = \prod_{i \in S_u}(x - \alpha_i)$ and $D_u^R(x) = \prod_{i \in S_u}(1 - \alpha_i x)$.

- **Downward Pass:** To evaluate $D'(x)$ at all α_i, we initialize the root state:

$$T_{root}(x) = D'_{root}(x) \circ (D^R_{root}(x))^{-1} \pmod{x^n}$$

  At any internal node u, we compute:

$$T_L(x) = T_u(x) \circ D_R^R(x) \quad \text{and} \quad T_R(x) = T_u(x) \circ D_L^R(x)$$

- **Leaves:** When the recursion reaches a leaf representing α_i, we have: $[x^0]T_{\{i\}}(x) = D'(\alpha_i)$. We compute the $c_i = y_i / D'(\alpha_i)$, and return $N_{\{i\}}(x) = c_i$.
- **Combine:** We combine $N_L(x)$ and $N_R(x)$ to form N_u:

$$N_u(x) = N_L(x) D_R(x) + N_R(x) D_L(x)$$

  At the root, $N_{root}(x) = F(x)$.

Algorithm 24: Interpolation

```
Function FastInterpolation(α_1, ..., α_n, y_1, ..., y_n):
    D, D^R ← PrecomputeSubproductTree(α_1, ..., α_n);
    I_root(x) ← (D^R[root](x))^{-1} (mod x^n);
    Q_root(x) ← (d/dx) D[root](x);
    T_root(x) ← TransposedMul(Q_root(x), I_root(x));
    return InterpRec(root, T_root(x), D, D^R, y);

Function InterpRec(u, T_u(x), D, D^R, y):
    if u is a leaf for point α_i then
        v_i ← T_u[0];
        return (y_i · v_i^{-1}) (mod M);
    T_L(x) ← TransposedMul(T_u(x), D^R[R.child](x));
    T_R(x) ← TransposedMul(T_u(x), D^R[L.child](x));
    N_L(x) ← InterpRec(L.child, T_L(x), D, D^R, y);
    N_R(x) ← InterpRec(R.child, T_R(x), D, D^R, y);
    N_u(x) ← N_L(x) · D[R.child](x) + N_R(x) · D[L.child](x);
    return N_u(x);
```

6.21 Inverse Chirp-Z

The Inverse Chirp-Z algorithm reconstructs a polynomial $F(x) \in \mathbb{K}[x]_{<k}$ from its evaluations $y_i = F(z^i)$ at a geometric progression for $i = 0, \ldots, k-1$ in $O(k \log k)$.

The Lagrange interpolation formula gives:

$$F(x) = \sum_{i=0}^{k-1} \frac{y_i}{D'(z^i)} \prod_{j \neq i} (x - z^j)$$

where $D(x) = \prod_{j=0}^{k-1}(x - z^j)$. We consider the evaluation $D'(z^i) = \prod_{j \neq i}(z^i - z^j)$.

For $j < i$, we factor out z^i and substitute $m = i - j$:

$$\prod_{j=0}^{i-1}(z^i - z^j) = \prod_{j=0}^{i-1} z^i(1 - z^{j-i}) = z^{i^2} \prod_{m=1}^{i}(1 - z^{-m})$$

For $j > i$, we again factor out z^i and substitute $m = j - i$:

$$\prod_{j=i+1}^{k-1}(z^i - z^j) = \prod_{j=i+1}^{k-1} z^i(1 - z^{j-i}) = z^{i(k-1-i)} \prod_{m=1}^{k-1-i}(1 - z^m)$$

We multiply these together ($i^2 + ik - i - i^2 = i(k-1)$), yielding:

$$D'(z^i) = z^{i(k-1)} \left(\prod_{m=1}^{i}(1 - z^{-m}) \right) \left(\prod_{m=1}^{k-1-i}(1 - z^m) \right)$$

Thus, the weight $w_i = 1/D'(z^i)$ is a product of $z^{-i(k-1)}$, a prefix product of $(1 - z^{-m})^{-1}$, and a suffix product of $(1 - z^m)^{-1}$.We define $y'_i = y_i w_i$.

Consider the reversal of the interpolating polynomial, $F^R(x) = x^{k-1}F(x^{-1})$. Substituting x^{-1} yields:

$$F^R(x) = \sum_{i=0}^{k-1} y'_i x^{k-1} \prod_{j \neq i}(x^{-1} - z^j) = \sum_{i=0}^{k-1} y'_i \prod_{j \neq i}(1 - z^j x)$$

Notice that this reversed polynomial is exactly the numerator of a partial fraction decomposition. Defining $Q(x) = \prod_{j=0}^{k-1}(1 - z^j x)$, we obtain:

$$\frac{F^R(x)}{Q(x)} = \sum_{i=0}^{k-1} \frac{y'_i}{1 - z^i x}$$

We expand the rational function:

$$\sum_{i=0}^{k-1} \frac{y'_i}{1 - z^i x} = \sum_{i=0}^{k-1} y'_i \sum_{m=0}^{\infty} (z^i x)^m = \sum_{m=0}^{\infty} \left(\sum_{i=0}^{k-1} y'_i z^{im} \right) x^m$$

Let $S_m = \sum_{i=0}^{k-1} y'_i z^{im}$. S_m is the evaluation of $Y(x) = \sum_{i=0}^{k-1} y'_i x^i$ at the point z^m. Thus, the first k terms, $S(x) = \sum_{m=0}^{k-1} S_m x^m$, can be computed by applying a Chirp-Z transform to $Y(x)$.

Because $\deg(F^R) < k$, we can then compute:

$$F^R(x) \equiv S(x)Q(x) \pmod{x^k}$$

We thus need to compute $Q(x) = \prod_{j=0}^{k-1}(1 - z^j x)$. We apply Lemma 6.14, which provides:

$$Q(x) = \prod_{j=0}^{k-1}(1 - z^j x) = \sum_{i=0}^{k}(-1)^i z^{\binom{i}{2}} \binom{k}{i}_z x^i$$

where $\binom{k}{i}_z$ denotes the q-binomial coefficient. This can be computed in $O(k)$.

Algorithm 25: Inverse Chirp-Z

Function `PrefProdInverses`(a, k)**:**
- `// Returns array` P `where` $P[m] = \prod_{j=1}^{m}(1 - a^j)^{-1}$
- $P \leftarrow$ array of size k with $P[0] = 1$;
- $a_j \leftarrow a$;
- **for** $m \leftarrow 1$ **to** $k - 1$ **do**
  - $P[m] \leftarrow P[m-1] \cdot (1 - a_j)^{-1} \pmod{M}$;
  - $a_j \leftarrow (a_j \cdot a) \pmod{M}$;
- **return** P;

Function `InverseChirpZ`$(y_0 \dots y_{k-1}, z, k)$**:**
- $P_{pos} \leftarrow$ `PrefProdInverses`(z, k);
- $P_{neg} \leftarrow$ `PrefProdInverses`(z^{-1}, k);
- $z_k \leftarrow z^{-(k-1)} \pmod{M}$;
- $z_{ki} \leftarrow 1$;
- $Y(x) \leftarrow 0$;
- **for** $i \leftarrow 0$ **to** $k - 1$ **do**
  - $w_i \leftarrow z_{ki} \cdot P_{neg}[i] \cdot P_{pos}[(k-1) - i] \pmod{M}$;
  - $Y[i] \leftarrow y_i \cdot w_i \pmod{M}$;
  - $z_{ki} \leftarrow z_{ki} \cdot z_k \pmod{M}$;
- $S(x) \leftarrow$ `ChirpZ`$(Y(x), z, k)$;
- $Q(x) \leftarrow \sum_{i=0}^{k-1} \left((-1)^i z^{\binom{i}{2}} \binom{k}{i}_z\right) x^i$;
- $F^R(x) \leftarrow (S(x) \cdot Q(x)) \pmod{x^k}$;
- **return** Reverse$(F^R(x), k)$;

6.22 GCD

HGCD: Given polynomials $A, B \in \mathbb{K}[x]$ with $\deg(A) > \deg(B) \geq 0$, the Half-GCD algorithm returns a regular matrix R such that:

$$\begin{pmatrix} A' \\ B' \end{pmatrix} = R \begin{pmatrix} A \\ B \end{pmatrix}$$

where the degrees of the output straddle half the degree of A. Specifically:

$$\deg(A') \geq \lceil \deg(A)/2 \rceil > \deg(B')$$

The HGCD Algorithm The idea is that some prefix of the quotient sequence is determined only by the high degree coefficients. The algorithm considers a threshold $m = \lceil \deg(A)/2 \rceil$. It calls itself on the upper halves of the polynomials shifted right by m. After applying the returned matrix R_1, the resulting polynomials A' and B' will straddle $\approx 3m/2$. Then we perform one Euclidean division step to decrease the degree below m, then make a second recursive call.

Algorithm 26: Half-GCD

Function `HalfGCD`(A, B)**:**
- $n \leftarrow \deg(A)$;
- $m \leftarrow \lceil n/2 \rceil$;
- **if** $\deg(B) < m$ **then**
  - **return** $E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$;
- $A_0 \leftarrow A \operatorname{div} x^m$;
- $B_0 \leftarrow B \operatorname{div} x^m$;
- $R_1 \leftarrow$ `HalfGCD`(A_0, B_0);
- $\begin{pmatrix} A' \\ B' \end{pmatrix} \leftarrow R_1 \begin{pmatrix} A \\ B \end{pmatrix}$;
- **if** $\deg(B') < m$ **then**
  - **return** R_1;
- $Q \leftarrow A' \operatorname{div} B'$;
- $R \leftarrow A' \bmod B'$;
- $C \leftarrow B'$;
- $D \leftarrow R$;
- $k \leftarrow 2m - \deg(C)$;
- $C_0 \leftarrow C \operatorname{div} x^k$;
- $D_0 \leftarrow D \operatorname{div} x^k$;
- $R_2 \leftarrow$ `HalfGCD`(C_0, D_0);
- **return** $R_2 \cdot \begin{pmatrix} 0 & 1 \\ 1 & -Q \end{pmatrix} \cdot R_1$;

Correctness

Definition 6.7 (Regular Matrix)**.** *A matrix associated with a quotient Q is defined as $E_Q = \begin{pmatrix} Q & 1 \\ 1 & 0 \end{pmatrix}$. A matrix W is called regular if it is the product of elementary matrices associated a Euclidean remainder sequence: $W = E_{Q_1} E_{Q_2} \cdots E_{Q_k}$, where $\deg(Q_i) \geq 1$ for all i.*

Lemma 6.8 (Ordering Property of Regular Matrices)**.** *Let $W = \begin{pmatrix} P & Q \\ S & T \end{pmatrix}$ be a regular matrix constructed from quotients $Q_1, \ldots, Q_k$ with $\deg(Q_i) \geq 1$. The degrees of the entries satisfy:*

$$\deg(P) = \sum_{i=1}^{k} \deg(Q_i)$$

$$\deg(P) \geq \max(\deg(Q), \deg(S)) \geq \min(\deg(Q), \deg(S)) \geq \deg(T)$$

Furthermore, $\deg(P) > \deg(Q)$ and $\deg(P) > \deg(S)$.

Proof. We proceed by induction on k. Base case $k = 1$: $W = \begin{pmatrix} Q_1 & 1 \\ 1 & 0 \end{pmatrix}$. Here $\deg(P) = \deg(Q_1) \geq 1$, $\deg(Q) = \deg(S) = 0$, and $\deg(T) = -\infty$.
Inductive step: Assume $W_k = \begin{pmatrix} P_k & Q_k \\ S_k & T_k \end{pmatrix}$ satisfies the property. For $W_{k+1} = W_k \begin{pmatrix} Q_{k+1} & 1 \\ 1 & 0 \end{pmatrix}$, we have:

$$W_{k+1} = \begin{pmatrix} P_k Q_{k+1} + Q_k & P_k \\ S_k Q_{k+1} + T_k & S_k \end{pmatrix}$$

Because $\deg(P_k) > \deg(Q_k)$ by the inductive hypothesis, the degree of the new top-left entry is exactly $\deg(P_k) + \deg(Q_{k+1})$. Since $\deg(Q_{k+1}) \geq 1$, this strictly dominates the degree of P_k, preserving the property. □

We prove that applying a transformation matrix computed from truncated top-coefficients correctly reduces the full polynomials.

Lemma 6.9 (Parallel Reduction Criteria)**.** *Let $A, B \in \mathbb{K}[x]$ be polynomials, and let $m \geq 1$ be a truncation threshold. Define $A_0 = A \operatorname{div} x^m$ and $B_0 = B \operatorname{div} x^m$.*

Let R_1 be the matrix returned by $\mathrm{HGCD}(A_0, B_0)$ such that $\begin{pmatrix} A_0' \\ B_0' \end{pmatrix} = R_1 \begin{pmatrix} A_0 \\ B_0 \end{pmatrix}$. If $\deg(A_0') \geq \lceil \deg(A_0)/2 \rceil > \deg(B_0')$, then applying R_1 to the full polynomials (A, B) yields (A', B') satisfying:

$$\deg(A') = \deg(A_0') + m \quad \text{and} \quad \deg(A') > \deg(B')$$

Proof. We can decompose the original polynomials as $A = A_0x^m + A_1$ and $B = B_0x^m + B_1$, where the residues strictly satisfy $\deg(A_1) < m$ and $\deg(B_1) < m$.

Let $R_1^{-1} = W = \begin{pmatrix} P & Q \\ S & T \end{pmatrix}$. We have $A_0 = PA_0' + QB_0'$. Since $\deg(A_0') > \deg(B_0')$ and $\deg(P) > \deg(Q)$ (by Lemma 6.7), the degree of the product PA_0' dominates QB_0'. Therefore:

$$\deg(A_0) = \deg(P) + \deg(A_0') \implies \deg(P) = \deg(A_0) - \deg(A_0')$$

Now we apply the forward reduction matrix R_1 to the full polynomials. Since $\det(W) = \delta = \pm 1$, we have $R_1 = \delta \begin{pmatrix} T & -Q \\ -S & P \end{pmatrix}$. By linearity:

$$\begin{pmatrix} A' \\ B' \end{pmatrix} = R_1 \begin{pmatrix} A_0x^m + A_1 \\ B_0x^m + B_1 \end{pmatrix} = \begin{pmatrix} A_0'x^m \\ B_0'x^m \end{pmatrix} + \delta \begin{pmatrix} TA_1 - QB_1 \\ -SA_1 + PB_1 \end{pmatrix}$$

Let A_1' and B_1' be the error terms. We upper-bound their degrees:

$$\begin{aligned} \deg(A_1') &\le \max(\deg(T) + \deg(A_1), \deg(Q) + \deg(B_1)) \\ &\le \max(\deg(T), \deg(Q)) + m - 1 \\ &\le \deg(P) + m - 1 \quad \text{(by Lemma 6.7)} \end{aligned}$$

Substitute $\deg(P) = \deg(A_0) - \deg(A_0')$ into the bound:

$$\deg(A_1') \le \deg(A_0) - \deg(A_0') + m - 1$$

For A' to be determined by the leading term $A_0'x^m$, we require $\deg(A_0'x^m) > \deg(A_1')$. We evaluate this:

$$\deg(A_0') + m > \deg(A_0) - \deg(A_0') + m - 1$$

$$2\deg(A_0') \ge \deg(A_0) \implies \deg(A_0') \ge \lceil \deg(A_0)/2 \rceil$$

Which we assume, so $\deg(A') = \deg(A_0') + m$.

By the same logic, we upper-bound the error B_1':

$$\begin{aligned} \deg(B_1') &\le \max(\deg(S) + \deg(A_1), \deg(P) + \deg(B_1)) \\ &\le \deg(P) + m - 1 = \deg(A_0) - \deg(A_0') + m - 1 \end{aligned}$$

Since $B' = B_0'x^m + B_1'$, its degree is bounded by:

$$\deg(B') \le \max(\deg(B_0') + m,\ \deg(A_0) - \deg(A_0') + m - 1)$$

We compare this against $\deg(A') = \deg(A_0') + m$. Since $\deg(B_0') \le \deg(A_0') - 1$, the first term inside the max is strictly less than $\deg(A')$. Since $2\deg(A_0') \ge \deg(A_0)$, we have $\deg(A_0') \ge \deg(A_0) - \deg(A_0')$. Therefore, the second term $\deg(A_0) - \deg(A_0') + m - 1 \le \deg(A_0') + m - 1$, which is also strictly less than $\deg(A')$. Thus, $\deg(A') > \deg(B')$. □

Lemma 6.10 (HGCD Straddling)**.** *Let $A, B \in \mathbb{K}[x]$ be polynomials with $\deg(A) > \deg(B) \ge 0$. The* HGCD(A, B) *algorithm returns a matrix R mapping $(A, B) \to (C', D')$ such that the final degrees straddle $m = \lceil \deg(A)/2 \rceil$:*

$$\deg(C') \ge m > \deg(D')$$

Proof. We induct on the degree of A.

Let $m = \lceil \deg(A)/2 \rceil$. The first recursive call HGCD(A_0, B_0) returns R_1. By the inductive hypothesis, R_1 correctly straddles $\lceil \deg(A_0)/2 \rceil$, satisfying the premise of Lemma 6.9. Applying R_1 yields A', B' with $\deg(A') \ge m > \deg(B_0')$.

Case 1: If $\deg(B') < m$, the algorithm exits at Step 2. Since $\deg(A') \ge m$, the straddling condition $\deg(A') \ge m > \deg(B')$ is satisfied.

Case 2: If $\deg(B') \ge m$, the algorithm proceeds to Step 3. It computes $C = B'$ and $D = A' \bmod B'$. We have $\deg(C) > \deg(D)$ and because we did not exit, we know $\deg(C) \ge m$. To find the second threshold, we define $k = 2m - \deg(C)$. We must verify that $k \ge 0$ and that $\deg(C) \ge k$.

From Lemma 6.9, $\deg(C) = \deg(B') \le \max(\deg(B_0') + m, \deg(A_0) - \deg(A_0') + m - 1)$. Using $\deg(A_0') \ge \lceil \deg(A_0)/2 \rceil$, this evaluates to $\deg(C) < \frac{3}{2}m$. Since $\deg(C) < 2m$, we have $k > 0$. Furthermore, because $\deg(C) \ge m$, it follows that $2\deg(C) \ge 2m$, which implies $\deg(C) \ge 2m - \deg(C) = k$.

In Step 4, we make the second recursive call HGCD(C_0, D_0) where $C_0 = C \operatorname{div} x^k$ and $D_0 = D \operatorname{div} x^k$. By definition, $\deg(C_0) = \deg(C) - k$. Substitute k:

$$\deg(C_0) = \deg(C) - (2m - \deg(C)) = 2\deg(C) - 2m$$

The inductive hypothesis on this second call guarantees that the returned matrix R_2 maps $(C_0, D_0) \to (C_0', D_0')$ such that:

$$\deg(C_0') \geq \lceil \deg(C_0)/2 \rceil = \deg(C) - m \quad \text{and} \quad \deg(C_0') > \deg(D_0')$$

This satisfies the premises of Lemma 6.9 for (C, D) with threshold k. Applying R_2 to the full (C, D) yields the final polynomials (C', D').

We now calculate the bounds of the final degrees. By Lemma 6.9:

$$\deg(C') = \deg(C_0') + k \geq (\deg(C) - m) + (2m - \deg(C)) = m$$

For the remainder D', the lemma bounds its degree by:

$$\deg(D') \leq \max(\deg(D_0') + k,\ \deg(C_0) - \deg(C_0') + k - 1)$$

We bound both terms. Since the inductive hypothesis forces $\deg(D_0') \leq \lceil \deg(C_0)/2 \rceil - 1 = \deg(C) - m - 1$, we have:

$$\deg(D_0') + k \leq (\deg(C) - m - 1) + (2m - \deg(C)) = m - 1 < m$$

We use the lower bound $\deg(C_0') \geq \deg(C) - m$, which implies $-\deg(C_0') \leq m - \deg(C)$:

$$\begin{aligned} \deg(C_0) - \deg(C_0') + k - 1 &\leq (2\deg(C) - 2m) + (m - \deg(C)) + (2m - \deg(C)) - 1 \\ &= m - 1 < m \end{aligned}$$

We conclude $\deg(D') < m$. Therefore, the final degrees satisfy $\deg(C') \geq m > \deg(D')$. □

Full GCD Algorithm To compute the complete Extended GCD of A and B, we interleave HGCD with a Euclidean division step.

Algorithm 27: GCD

Function `GCD`(A, B)**:**
- $M_{\text{full}} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$;
- **while** $B \neq 0$ **do**
  - $Q \leftarrow A \operatorname{div} B$;
  - $R \leftarrow A \bmod B$;
  - $M_{\text{step}} \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & -Q \end{pmatrix}$;
  - $A \leftarrow B$; $\quad B \leftarrow R$;
  - $M_{\text{full}} \leftarrow M_{\text{step}} \cdot M_{\text{full}}$;
  - $R_{\text{half}} \leftarrow$ `HalfGCD`(A, B);
  - $\begin{pmatrix} A \\ B \end{pmatrix} \leftarrow R_{\text{half}} \begin{pmatrix} A \\ B \end{pmatrix}$;
  - $M_{\text{full}} \leftarrow R_{\text{half}} \cdot M_{\text{full}}$;
- **return** A, M_{full};

6.23 Minimum Linear Recurrence

We can find the minimum linear recurrence of a sequence in $O(n \log^2 n)$ time.

Given a sequence $s_0, s_1, \ldots, s_{n-1}$, we define its generating polynomial:

$$S(x) = \sum_{i=0}^{n-1} s_i x^i$$

The sequence satisfies a linear recurrence of order d if there exist coefficients $c_0, c_1, \ldots, c_d$ (with $c_0 = 1$) such that for all $k \geq d$, $\sum_{j=0}^{d} c_j s_{k-j} = 0$.

If we define the characteristic polynomial of this recurrence as $C(x) = \sum_{j=0}^{d} c_j x^j$, the recurrence relation says that the higher-order terms of the product $S(x)C(x)$ are 0. Specifically, the product is some error polynomial $P(x)$ of degree $< d$. Thus:

$$S(x)C(x) \equiv P(x) \pmod{x^n}$$

where $\deg(C) = d$, $C(0) = 1$, and $\deg(P) < d$. Our goal is to find $C(x)$ while minimizing d.

Reversal To avoid dealing with the requirement $C(0) = 1$, we can reverse the sequence and ensure that it's monic.

Suppose $C(x)$ defines a linear recurrence of order d for $S(x)$. This means $\deg(C) = d$ and we can write:

$$S(x)C(x) = P(x) + x^n R(x)$$

where $\deg(P) < d$. Substituting x with x^{-1} and multiplying both sides by x^{n-1+d} yields:

$$x^{n-1}S(x^{-1}) \cdot x^d C(x^{-1}) = x^{n-1+d}P(x^{-1}) + x^{d-1}R(x^{-1})$$

We define the reversed polynomials as follows:

$$\begin{aligned}\tilde{S}(x) &= x^{n-1}S(x^{-1}) \\ \tilde{C}(x) &= x^d C(x^{-1}) \\ \tilde{P}(x) &= x^{d-1}R(x^{-1}) \\ \tilde{R}(x) &= x^{n-1+d}P(x^{-1})\end{aligned}$$

Because $\deg(S) < n$ and $\deg(C) = d$, the highest degree of $S(x)C(x)$ is strictly less than $n + d$, meaning $\deg(R) \leq d - 1$. Thus, $\tilde{R}(x^{-1})$ contains only non-positive powers of x down to $x^{-(d-1)}$, ensuring $\tilde{P}(x)$ is a polynomial with $\deg(\tilde{P}) \leq d - 1$.

Conversely, because $\deg(P) \leq d-1$, the smallest power of x in $P(x^{-1})$ is $-(d-1)$. The lowest power of x in $\tilde{R}(x)$ is therefore $n - 1 + d - (d - 1) = n$. This guarantees that $\tilde{R}(x) \equiv 0 \pmod{x^n}$.

This gives us the congruence:

$$\tilde{S}(x)\tilde{C}(x) = \tilde{P}(x) + \tilde{R}(x) \equiv \tilde{P}(x) \pmod{x^n}$$

Therefore, if $\deg(P) < \deg(C)$, it holds that $\deg(\tilde{P}) < \deg(\tilde{C})$. To find the minimum recurrence for $S(x)$, we compute the minimum recurrence for its reverse $\tilde{S}(x)$ and then reverse the resulting polynomial.

Degree Invariant We consider applying the Extended Euclidean Algorithm to $r_{-1} = x^n$ and $r_0 = \tilde{S}(x)$. This generates a sequence of quotients A_k, remainders r_k, and denominators Q_k defined by:

$$\begin{aligned} r_k &= r_{k-2} - A_k r_{k-1} \quad \text{where} \quad A_k = r_{k-2} \operatorname{div} r_{k-1} \\ Q_k &= A_k Q_{k-1} + Q_{k-2} \quad \text{with} \quad Q_{-1} = 0,\ Q_0 = 1 \end{aligned}$$

Lemma 6.11 (Convergent Congruence)**.** *For all $k \geq 0$, the Euclidean convergent denominator $Q_k(x)$ satisfies:*

$$Q_k(x)\tilde{S}(x) \equiv (-1)^k r_k(x) \pmod{x^n}$$

Proof. We proceed by induction on k. Recall the initial conditions: $r_{-1} = x^n$, $r_0 = \tilde{S}(x)$, $Q_{-1} = 0$, and $Q_0 = 1$.

Base Cases ($k = 0, 1$):
For $k = 0$, $Q_0\tilde{S}(x) = 1 \cdot \tilde{S}(x) = r_0(x) = (-1)^0 r_0(x)$, which holds exactly.
For $k = 1$, the first division step gives $r_1 = r_{-1} - A_1 r_0$. Taking this modulo x^n (since $r_{-1} = x^n \equiv 0$) yields $r_1 \equiv -A_1\tilde{S}(x) \pmod{x^n}$. Since $Q_1 = A_1Q_0 + Q_{-1} = A_1$, we have:

$$Q_1\tilde{S}(x) = A_1\tilde{S}(x) \equiv -r_1(x) = (-1)^1 r_1(x) \pmod{x^n}$$

Inductive Step:
Assume the congruence holds for $k - 2$ and $k - 1$. Multiply the recurrence $Q_k = A_kQ_{k-1} + Q_{k-2}$ by $\tilde{S}(x)$:

$$\begin{aligned} Q_k\tilde{S}(x) &= A_k\big(Q_{k-1}\tilde{S}(x)\big) + \big(Q_{k-2}\tilde{S}(x)\big) \\ &\equiv A_k\big((-1)^{k-1}r_{k-1}(x)\big) + \big((-1)^{k-2}r_{k-2}(x)\big) \pmod{x^n} \end{aligned}$$

Factor out $(-1)^k$, noting that $(-1)^{k-1} = -(-1)^k$ and $(-1)^{k-2} = (-1)^k$:

$$Q_k\tilde{S}(x) \equiv (-1)^k\Big(- A_k r_{k-1}(x) + r_{k-2}(x)\Big) \pmod{x^n}$$

By the remainder recurrence $r_k = r_{k-2} - A_k r_{k-1}$, the term inside the parentheses is exactly $r_k(x)$. Thus:

$$Q_k\tilde{S}(x) \equiv (-1)^k r_k(x) \pmod{x^n}$$

□

Lemma 6.12 (Degree Invariant)**.** *For all $k \geq 0$, the sum of the degree of the k-th convergent and the $(k-1)$-th remainder is exactly n:*

$$\deg(Q_k) + \deg(r_{k-1}) = n$$

Proof. We proceed by induction on k. Base case ($k = 1$): Since $A_1 = r_{-1} \operatorname{div} r_0$, we have $\deg(A_1) = \deg(r_{-1}) - \deg(r_0) = n - \deg(r_0)$. By definition, $Q_1 = A_1 Q_0 + Q_{-1} = A_1(1) + 0 = A_1$. Thus, $\deg(Q_1) = n - \deg(r_0)$, so $\deg(Q_1) + \deg(r_0) = n$.

Inductive step: Assume $\deg(Q_{k-1}) + \deg(r_{k-2}) = n$. By the division algorithm, $A_k = r_{k-2} \operatorname{div} r_{k-1}$, so $\deg(A_k) = \deg(r_{k-2}) - \deg(r_{k-1})$. For the convergent, $\deg(Q_k) = \deg(A_k Q_{k-1} + Q_{k-2})$. Since $\deg(A_k) \geq 1$, the degree strictly grows, meaning $\deg(Q_k) = \deg(A_k) + \deg(Q_{k-1})$. Substituting $\deg(A_k)$:

$$\deg(Q_k) = (\deg(r_{k-2}) - \deg(r_{k-1})) + \deg(Q_{k-1})$$

Rearranging terms yields:

$$\deg(Q_k) + \deg(r_{k-1}) = \deg(Q_{k-1}) + \deg(r_{k-2}) = n$$

□

Because $\deg(r_k) < \deg(r_{k-1})$, this invariant implies that $\deg(Q_k) + \deg(r_k) < n$ for all k.

Minimality A linear recurrence requires: $\deg(r_k) < \deg(Q_k)$. Because $\deg(r_k)$ strictly decreases and $\deg(Q_k)$ strictly increases, there is a unique crossover point.

Lemma 6.13 (Minimal Recurrence Uniqueness)**.** *Let k be the first step in the Euclidean sequence where $\deg(r_k) < \deg(Q_k)$. Then $Q_k(x)$ is the unique minimal-degree characteristic polynomial satisfying the linear recurrence.*

Proof. Because k is the first step where the degrees cross over, the preceding step must not have crossed over. Therefore, the $(k-1)$-th remainder and convergent satisfy:

$$\deg(r_{k-1}) \geq \deg(Q_{k-1})$$

Assume for contradiction there exists another valid pair (Q^*, r^*) satisfying $Q^* \tilde{S} \equiv r^* \pmod{x^n}$ with $\deg(r^*) < \deg(Q^*)$ and strictly smaller degree: $\deg(Q^*) < \deg(Q_k)$.

We cross-multiply this valid pair with the $(k-1)$-th Euclidean congruence $Q_{k-1}\tilde{S} \equiv \pm r_{k-1} \pmod{x^n}$:

$$\begin{aligned} Q^*(\pm r_{k-1}) &\equiv Q^*(Q_{k-1}\tilde{S}) \pmod{x^n} \\ Q_{k-1} r^* &\equiv Q_{k-1}(Q^*\tilde{S}) \pmod{x^n} \end{aligned}$$

Subtracting these yields $\pm Q^* r_{k-1} - Q_{k-1} r^* \equiv 0 \pmod{x^n}$.

Now, we bound the maximum degree of this difference. By the degree invariant, we know $\deg(r_{k-1}) = n - \deg(Q_k)$.

1. $\deg(Q^* r_{k-1}) = \deg(Q^*) + \deg(r_{k-1}) = \deg(Q^*) + n - \deg(Q_k)$.
   Because we assumed $\deg(Q^*) < \deg(Q_k)$, this evaluates to strictly less than n.

2. $\deg(Q_{k-1} r^*) = \deg(Q_{k-1}) + \deg(r^*)$.
   Since $\deg(Q_{k-1}) \leq \deg(r_{k-1}) = n - \deg(Q_k)$, and $\deg(r^*) < \deg(Q^*) \leq \deg(Q_k) - 1$, the sum is strictly bounded by $(n - \deg(Q_k)) + (\deg(Q_k) - 1) = n - 1 < n$.

Because both cross-terms have a degree strictly less than n, their difference must also have a degree strictly less than n. However, the difference is congruent to $0 \pmod{x^n}$, thus it's equal to 0.

Thus, we obtain:

$$Q^* r_{k-1} = \pm Q_{k-1} r^*$$

Taking the degree of both sides gives:

$$\deg(Q^*) + \deg(r_{k-1}) = \deg(Q_{k-1}) + \deg(r^*)$$

Rearranging the terms to group by step yields:

$$\deg(r_{k-1}) - \deg(Q_{k-1}) = \deg(r^*) - \deg(Q^*)$$

Since k is the first crossover step, we know the left-hand side is non-negative (≥ 0), thus so is the right-hand side, meaning $\deg(r^*) \geq \deg(Q^*)$, which is a contradiction. □

We can find this step k without expanding the polynomials by using only the degrees of the quotients A_i. Substitute $\deg(Q_k) = n - \deg(r_{k-1})$ into the minimality condition $\deg(r_k) < \deg(Q_k)$:

$$\deg(r_k) < n - \deg(r_{k-1}) \implies \deg(r_{k-1}) + \deg(r_k) < n$$

We define a tracking variable $\delta_k = \deg(r_{k-1}) + \deg(r_k) - n$. The optimal crossover point occurs at the first k where $\delta_k < 0$.

To compute δ_k iteratively, we start with $\delta_0 = \deg(r_{-1}) + \deg(r_0) - n = \deg(r_0)$. At each step, we update the variable:

$$\begin{aligned}\delta_k - \delta_{k-1} &= (\deg(r_{k-1}) + \deg(r_k) - n) - (\deg(r_{k-2}) + \deg(r_{k-1}) - n) \\ &= \deg(r_k) - \deg(r_{k-2})\end{aligned}$$

Since $r_{k-2} = A_k r_{k-1} + r_k$ and $r_{k-1} = A_{k+1} r_k + r_{k+1}$, we have $\deg(r_{k-2}) - \deg(r_{k-1}) = \deg(A_k)$ and $\deg(r_{k-1}) - \deg(r_k) = \deg(A_{k+1})$. Summing these gives $\deg(r_{k-2}) - \deg(r_k) = \deg(A_k) + \deg(A_{k+1})$. Therefore, the update rule is:

$$\delta_k = \delta_{k-1} - \deg(A_k) - \deg(A_{k+1})$$

Recovery To recover the characteristic polynomial, we need to compute the convergent matrix:

$$\mathcal{M}_{0,k} = \prod_{i=0}^{k-1} \begin{pmatrix} A_i & 1 \\ 1 & 0 \end{pmatrix}$$

The top-left entry of this product is the convergent denominator $Q_k(x)$. We use divide and conquer, splitting the sequence based on the sum of degrees in the left and right halves.

Algorithm 28: Minimum Linear Recurrence

Function `EvaluateConvergent`(l, r, A_{quots})**:**
- **if** $r - l \leq 1$ **then**
  - **return** $\begin{pmatrix} A_{\text{quots}}[l] & 1 \\ 1 & 0 \end{pmatrix}$;
- $S \leftarrow \sum_{i=l}^{r-1} \deg(A_{\text{quots}}[i])$;
- $k \leftarrow l + 1$;
- $C \leftarrow \deg(A_{\text{quots}}[l])$;
- **while** $k < r - 1$ ***and*** $2 \cdot C \leq S$ **do**
  - $C \leftarrow C + \deg(A_{\text{quots}}[k])$;
  - $k \leftarrow k + 1$;
- $\mathcal{M}_{\text{left}} \leftarrow$ `EvaluateConvergent`(l, k, A_{quots});
- $\mathcal{M}_{\text{right}} \leftarrow$ `EvaluateConvergent`(k, r, A_{quots});
- **return** $\mathcal{M}_{\text{left}} \cdot \mathcal{M}_{\text{right}}$;

Function `MinRecurrence`$(S(x), n)$**:**
- $r_{-1}(x) \leftarrow x^n$;
- $r_0(x) \leftarrow \text{Reverse}(S(x) \pmod{x^n})$;
- **if** $r_0(x) = 0$ **then return** 1;
- $(g, \ldots, A_{\text{quots}}, \ldots) \leftarrow$ `FullGCD`(r_{-1}, r_0);
- Append 0 to A_{quots};
- $pref \leftarrow 0$;
- $\delta \leftarrow n - \deg(A_{\text{quots}}[0])$;
- **while** $\delta \geq 0$ **do**
  - $\delta \leftarrow \delta - (\deg(A_{\text{quots}}[pref]) + \deg(A_{\text{quots}}[pref + 1]))$;
  - $pref \leftarrow pref + 1$;
- $\mathcal{M} \leftarrow$ `EvaluateConvergent`$(0, pref, A_{\text{quots}})$;
- $Q(x) \leftarrow \mathcal{M}_{0,0}$;
- $Q_{\text{rev}}(x) \leftarrow \text{Reverse}(Q(x))$;
- $c \leftarrow Q_{\text{rev}}[0]$;
- **return** $Q_{\text{rev}}(x) \cdot c^{-1}$;

6.24 Modular Inversion

Given a polynomial $A(x)$ and a modulus polynomial $M(x)$, the modular inversion problem asks to find a polynomial $A^{-1}(x)$ such that:

$$A(x)A^{-1}(x) \equiv 1 \pmod{M(x)}$$

This can be computed in $O(n \log^2 n)$ time using Half-GCD.

1. Bézout's Identity By running Extended GCD on $M(x)$ and $A(x)$, we obtain the greatest common divisor $g(x) = \gcd(M(x), A(x))$ alongside the transition matrix T that satisfies Bézout's identity:

$$\begin{pmatrix} T_{11}(x) & T_{12}(x) \\ T_{21}(x) & T_{22}(x) \end{pmatrix} \begin{pmatrix} M(x) \\ A(x) \end{pmatrix} = \begin{pmatrix} g(x) \\ 0 \end{pmatrix}$$

Expanding the top row of the matrix multiplication yields:

$$T_{11}(x)M(x) + T_{12}(x)A(x) = g(x)$$

Taking this equation modulo $M(x)$, the first term vanishes, leaving:

$$T_{12}(x)A(x) \equiv g(x) \pmod{M(x)}$$

2. Normalization An inverse exists if and only if $\deg(g(x)) = 0$ so it's a unit in $\mathbb{K}[x]$, we then divide both sides by $g(x)$ and obtain the inverse.

$$\left(T_{12}(x) \cdot g_0^{-1}\right) A(x) \equiv 1 \pmod{M(x)}$$

Algorithm 29: Modular Inversion

```
Function InverseMod(A(x), M(x)):
    if A(x) = 0 then return None;
    // Compute Extended GCD and Transformation Matrix T
    (g(x), ..., T) ← FullGCD(M(x), A(x));
    // Check for coprimality
    if deg(g) ≠ 0 then
        return None ;                                   // No inverse exists
    // T_12 is the coefficient corresponding to A(x)
    U(x) ← T_12(x);
    g_0 ← g[0];
    return U(x) · g_0^{-1} (mod M(x));
```

6.25 Root Finding

The Berlekamp-Rabin algorithm isolates roots of $f(x)$ over $\mathbb{F}_p$ by first extracting distinct linear factors via $f_{\text{linear}}(x) = \gcd(f(x), x^p - x)$. It then divides the roots among two parts using random shifts.

1. Pairwise Separation For any two distinct roots α, β, a random shift $r \in \mathbb{F}_p$ separates them if the test polynomial $(x + r)^{(p-1)/2} \pmod{p}$ yields different values for them. This occurs when:

$$-1 = \left(\frac{\alpha + r}{p}\right)\left(\frac{\beta + r}{p}\right) = \left(\frac{1 + (\beta - \alpha)(\alpha + r)^{-1}}{p}\right)$$

Letting $x = \alpha + r$ and $c = \beta - \alpha \neq 0$, the inner term $1 + cx^{-1}$ maps to all $\frac{p-1}{2}$ non-residues in $\mathbb{F}_p$ as x varies. Thus, a random shift separates any specific pair of roots with probability at least $\frac{p-1}{2p}$. Consequently, the probability two roots remain together is bounded by $\rho \leq \frac{p+1}{2p} \leq \frac{2}{3}$ (for $p \geq 3$).

2. Recursion Depth Let $D_k(\alpha)$ be the degree of the polynomial containing α at recursion depth k. Let $D_k(\alpha)$ be the degree of the polynomial containing α at recursion depth k. At the root of the recursion tree, we have $D_0(\alpha) = n$.

Consider a single step from depth k to $k + 1$. Suppose that at depth k, the polynomial containing α has degree d. We define an indicator random variable for each $\beta \in S$ where S is the set of other $d - 1$ roots in the polynomial:

$$I_\beta = \begin{cases} 1 & \text{if } \beta \text{ remains in the same factor as } \alpha \\ 0 & \text{if } \beta \text{ is separated from } \alpha \end{cases}$$

The degree of the polynomial containing α at the next depth is:

$$D_{k+1}(\alpha) = 1 + \sum_{\beta \in S} I_\beta$$

From the above analysis, we know that the probability any specific β fails to separate from α is bounded by $\rho \leq \frac{p+1}{2p} \leq \frac{2}{3}$, and $\mathbb{E}[I_\beta] \leq \rho$.

The expected degree at depth $k + 1$ is:

$$\mathbb{E}[D_{k+1}(\alpha) \mid D_k(\alpha) = d] = 1 + \sum_{\beta \in S} \mathbb{E}[I_\beta] \leq 1 + (d - 1)\rho$$

Thus:

$$\mathbb{E}[D_{k+1}(\alpha) - 1 \mid D_k(\alpha) = d] \leq \rho(d - 1)$$

$$\mathbb{E}[D_{k+1}(\alpha) - 1] \leq \rho \cdot \mathbb{E}[D_k(\alpha) - 1]$$

$$\mathbb{E}[D_k(\alpha) - 1] \leq \rho^k (n-1)$$

By Markov's Inequality, the probability α is not yet isolated is $\leq \rho^k n$. Union Bounding we have:

$$P(L > k) \leq n^2 \rho^k$$

For $k = C \log n$ with a sufficiently large constant C, this probability is $< 1/n$.

3. Complexity

1. **Powering:** Computing $x^p \pmod{f(x)}$ or $(x+r)^{\frac{p-1}{2}} \pmod{f(x)}$ requires $O(n \log n \log p)$ time.
2. **Recursion:** At any depth k, splitting a polynomial of degree d takes $O(d \log d \log p)$ time. Thus the work across all disjoint polynomials at depth k is bounded by $O(n \log n \log p)$.

Summing the $O(n \log n \log p)$ work across the $O(\log n)$ levels yields an overall bound of $O(n \log^2 n \log p)$.

Algorithm 30: Root Finding

```
Function FindRoots(f(x), p):
    if deg(f) < 1 then return ∅;
    A(x) ← x^p (mod f(x));
    A(x) ← A(x) − x;
    f_linear(x) ← gcd(f(x), A(x));
    R ← empty list;
    FindRootsRecursive(f_linear(x), p, R);
    Sort R and return R;

Function FindRootsRecursive(f(x), p, R):
    d ← deg(f);
    if d = 0 then return;
    if d = 1 then
        r ← −f[0] · f[1]^{−1} (mod p);
        Add r to R;
        return;
    while true do
        Pick a random r ∈ [0, p − 1];
        q(x) ← x + r;
        A(x) ← q(x)^{(p−1)/2} (mod f(x));
        // If A(x) ≡ 1 or A(x) ≡ 0, we made no progress
        if A(x) is a constant polynomial then
            continue;
        A(x) ← A(x) − 1;
        g_1(x) ← gcd(f(x), A(x));
        if 0 < deg(g_1) < deg(f) then
            g_2(x) ← f(x)/g_1(x);
            FindRootsRecursive(g_1(x), p, R);
            FindRootsRecursive(g_2(x), p, R);
            break;
```

6.26 Q-Analogs

Lemma 6.14 (q-Binomial Theorem)**.** *For any positive integer n and indeterminates x and q,*

$$\prod_{i=0}^{n-1} (1 + xq^i) = \sum_{k=0}^{n} q^{\binom{k}{2}} \binom{n}{k}_q x^k$$

Proof. We evaluate the coefficient of x^k on the left-hand side of the equation.

To form a term with exactly x^k in the expansion of $\prod_{i=0}^{n-1}(1+xq^i)$, we must choose the "$xq^i$" term from exactly k of the n binomial factors, and the "1" from the remaining $n-k$ factors. Let the indices of the k chosen factors be $i_1, i_2, \ldots, i_k$, ordered such that:

$$0 \leq i_1 < i_2 < \cdots < i_k \leq n-1$$

The product of these chosen terms is $x^k q^{i_1+i_2+\cdots+i_k}$. Therefore, the total coefficient of x^k is:

$$[x^k]\prod_{i=0}^{n-1}(1+xq^i) = \sum_{0\leq i_1<i_2<\cdots<i_k\leq n-1} q^{i_1+i_2+\cdots+i_k}$$

We shift the sequence to remove this strict inequality by defining a new sequence $p_m = i_m-(m-1)$ for $m = 1, \ldots, k$. Our bounds are now:

$$0 \leq p_1 \leq p_2 \leq \cdots \leq p_k \leq n-1-(k-1) \implies 0 \leq p_1 \leq p_2 \leq \cdots \leq p_k \leq n-k$$

This is exactly a partition that fits inside a $k \times (n-k)$ rectangle.

Let us rewrite the sum of the original indices i_m in terms of the partition parts p_m:

$$\sum_{m=1}^{k} i_m = \sum_{m=1}^{k} \big(p_m + (m-1)\big) = \sum_{m=1}^{k} p_m + \sum_{m=1}^{k}(m-1)$$

The first sum is the total size of the partition, denoted $|\lambda|$. The second sum is $\binom{k}{2}$.

Substituting this back into our coefficient sum:

$$[x^k]\prod_{i=0}^{n-1}(1+xq^i) = \sum_{\lambda\subseteq k\times(n-k)} q^{|\lambda|+\binom{k}{2}} = q^{\binom{k}{2}} \sum_{\lambda\subseteq k\times(n-k)} q^{|\lambda|}$$

Thus, the sum simplifies to:

$$[x^k]\prod_{i=0}^{n-1}(1+xq^i) = q^{\binom{k}{2}}\binom{n}{k}_q$$

□

Lemma 6.15 (Durfee Square Identity)**.** *For indeterminates x and q,*

$$\prod_{i=1}^{\infty}\frac{1}{1-xq^i} = \sum_{k=0}^{\infty}\frac{x^k q^{k^2}}{(1-q)(1-q^2)\cdots(1-q^k)\cdot(1-xq)(1-xq^2)\cdots(1-xq^k)}$$

Proof. Consider the left-hand side. Letting the number of parts be $\ell(\lambda)$:

$$\prod_{i=1}^{\infty}\frac{1}{1-xq^i} = \sum_{\lambda} x^{\ell(\lambda)}q^{|\lambda|}$$

Consider partitions with a Durfee square of size k, removing it splits the partition into three parts:

1. A $k \times k$ block of cells, its weight is $x^k q^{k^2}$.
2. A partition to the right of the Durfee square. It can have at most k rows ($\ell(\mu) \leq k$) and does not create any new parts. Thus the contribution is:
$$\frac{1}{(1-q)(1-q^2)\cdots(1-q^k)}$$
3. A partition residing below the Durfee square. Its largest part can be at most k and every row adds a new part to λ. The contribution is thus:
$$\frac{1}{(1-xq)(1-xq^2)\cdots(1-xq^k)}$$

We have that $|\lambda| = k^2 + |\mu| + |\nu|$ and $\ell(\lambda) = k + \ell(\nu)$, thus we have the result. □

6.27 Jacobi Triple Product

Lemma 6.16.

$$[y^0]\prod_{j=1}^{\infty}(1+x^{2j-1}y^2)(1+x^{2j-1}y^{-2}) = \sum_{m=0}^{\infty}\frac{x^{2m^2}}{\prod_{i=1}^{m}(1-x^{2i})^2}$$

Proof. We must select m factors of y^2 and m factors of y^{-2} for some $m \geq 0$. Since these selections are independent, the constant term is $\sum_{m=0}^{\infty} G_m(x)^2$, where $G_m(x)$ is the generating function for the sum of m distinct positive odd integers.

Any sequence of m strictly increasing odd integers can be decomposed into $(1, 3, 5, \ldots, 2m-1)$, which sums to m^2, plus an increasing sequence of m non-negative even integers $(2p_1, 2p_2, \ldots, 2p_m)$.

This sequence corresponds to a partition into at most m even parts, generated by $\prod_{i=1}^{m}(1-x^{2i})^{-1}$. Thus:

$$G_m(x) = x^{m^2}\prod_{i=1}^{m}\frac{1}{1-x^{2i}}$$

□

Theorem 6.17 (Jacobi Triple Product)**.** *For complex numbers x and y with $|x| < 1$ and $y \neq 0$:*

$$\prod_{m=1}^{\infty}(1-x^{2m})(1+x^{2m-1}y^2)(1+x^{2m-1}y^{-2}) = \sum_{n=-\infty}^{\infty}x^{n^2}y^{2n}$$

Proof. Let $f_x(y)$ be defined as the infinite product on the left-hand side:

$$f_x(y) = \prod_{m=1}^{\infty}(1-x^{2m})(1+x^{2m-1}y^2)(1+x^{2m-1}y^{-2})$$

Substituting xy for y we have:

$$f_x(xy) = \prod_{m=1}^{\infty}(1-x^{2m})(1+x^{2m+1}y^2)(1+x^{2m-3}y^{-2})$$

Comparing this to the original $f_x(y)$, the product containing y^2 has lost its $m=1$ term, $(1+xy^2)$, while the product containing y^{-2} has gained an $m=0$ term, $(1+x^{-1}y^{-2})$. Thus:

$$\begin{aligned} f_x(xy) &= \frac{1+x^{-1}y^{-2}}{1+xy^2}f_x(y) \\ &= \frac{x^{-1}y^{-2}(xy^2+1)}{1+xy^2}f_x(y) \\ &= x^{-1}y^{-2}f_x(y) \end{aligned}$$

Because $f_x(y)$ is meromorphic for $|y| > 0$, it admits a Laurent series expansion in terms of y:

$$f_x(y) = \sum_{n=-\infty}^{\infty}c_n(x)y^{2n}$$

Substituting this series into our functional equation $xf_x(xy) = y^{-2}f_x(y)$, we obtain:

$$\sum_{n=-\infty}^{\infty}c_n(x)x^{2n+1}y^{2n} = y^{-2}\sum_{n=-\infty}^{\infty}c_n(x)y^{2n}$$

By shifting the index $n \to n+1$ on the right-hand side, we have:

$$\sum_{n=-\infty}^{\infty}c_n(x)x^{2n+1}y^{2n} = \sum_{n=-\infty}^{\infty}c_{n+1}(x)y^{2n}$$

Equating the coefficients of y^{2n} yields the recurrence relation:

$$c_{n+1}(x) = c_n(x)x^{2n+1}$$

This gives:

$$c_n(x) = c_0(x)x^{1+3+5+\cdots+(2n-1)} = c_0(x)x^{n^2}$$

Thus:

$$f_x(y) = c_0(x) \sum_{n=-\infty}^{\infty} x^{n^2} y^{2n}$$

We must show that $c_0(x) = 1$. This is the y^0 coefficient of the original infinite product. From Lemma 6.16 and 6.15 this is:

$$c_0(x) = \left(\prod_{m=1}^{\infty} (1 - x^{2m}) \right) \times \left(\prod_{m=1}^{\infty} \frac{1}{1 - x^{2m}} \right) = 1$$

□

6.28 MacMahon's Master Theorem

Theorem 6.18. *Let $A = (a_{ij})$ be an $n \times n$ matrix, and let $X = \operatorname{diag}(x_1, \ldots, x_n)$ be a diagonal matrix of formal variables. For any tuple $k = (k_1, \ldots, k_n) \in \mathbb{N}^n$, let $x^k = x_1^{k_1} \cdots x_n^{k_n}$. Then*

$$[x^k] \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} x_j \right)^{k_i} = [x^k] \frac{1}{\det(I - AX)}.$$

Proof. Let $R = \mathbb{C}[a_{11}, \ldots, a_{nn}]$ be the polynomial ring representing the generic matrix A. We work in the ring of formal power series $R[[x_1, \ldots, x_n]]$.

Let V be a free module of rank n over R with basis $v_1, \ldots, v_n$. Let $M = AX$, meaning $M_{ij} = a_{ij} x_j$. We define a linear operator $T : V \to V$ by its action on the standard basis:

$$T(v_i) = \sum_{j=1}^{n} M_{ij} v_j.$$

The operator T extends naturally to the symmetric algebra $\operatorname{Sym}(V) \cong R[v_1, \ldots, v_n]$. The subspace $S^m(V)$ of homogeneous polynomials of degree m has a basis of monomials $v^k = v_1^{k_1} \cdots v_n^{k_n}$ where $|k| = \sum_{i=1}^{n} k_i = m$. The action of T on this basis is:

$$T(v^k) = \prod_{i=1}^{n} T(v_i)^{k_i} = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} x_j v_j \right)^{k_i}.$$

The trace of T acting on $S^m(V)$ is the sum of its diagonal matrix entries in this standard basis:

$$\operatorname{Tr}(S^m(T)) = \sum_{|k|=m} [v^k] T(v^k) = \sum_{|k|=m} [v^k] \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} x_j v_j \right)^{k_i}.$$

In the polynomial expansion of $\prod_{i=1}^{n} (\sum_{j=1}^{n} a_{ij} x_j v_j)^{k_i}$, the variables x_j and v_j are strictly coupled; every term chosen from the inner sum contributes exactly one x_j and one v_j. Therefore, the formal coefficient of v^k is exactly x^k multiplied by the scalar coefficient we would obtain if we evaluated at $v_j = 1$. Thus:

$$[v^k] T(v^k) = x^k \cdot [x^k] \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} x_j \right)^{k_i}.$$

Summing this trace over all degrees m yields a formal power series:

$$\sum_{m=0}^{\infty} \operatorname{Tr}(S^m(T)) = \sum_{k \in \mathbb{N}^n} x^k \left([x^k] \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} x_j \right)^{k_i} \right).$$

We must now prove that this sum evaluates to $\det(I - AX)^{-1}$. Let W be an $n \times n$ matrix of independent formal variables w_{ij}. We will establish the generic trace identity $\sum_{m=0}^{\infty} \operatorname{Tr}(S^m(W)) = \det(I - W)^{-1}$ in $\mathbb{C}[[w_{11}, \ldots, w_{nn}]]$.

By comparing the homogeneous components of degree m, we define a sequence of polynomials:

$$F_m(W) = \operatorname{Tr}(S^m(W)) - [\text{degree } m \text{ of } \det(I - W)^{-1}].$$

We must prove $F_m(W)$ is identically zero in $\mathbb{C}[w_{11}, \ldots, w_{nn}]$.

Let $\Delta(W)$ be the discriminant of the characteristic polynomial of W with respect to λ. Because there exist matrices with distinct eigenvalues, Δ is a non-zero polynomial.

Let $C \in \mathbb{C}^{n\times n}$ be any specific complex matrix such that $\Delta(C) \neq 0$. This non-vanishing discriminant guarantees that C has n distinct roots, meaning C is diagonalizable. Let its eigenvalues be $\lambda_1, \ldots, \lambda_n$, with a basis of eigenvectors $u_1, \ldots, u_n$.

Because C acts diagonally on V, the induced basis of monomials $u^k = u_1^{k_1} \cdots u_n^{k_n}$ is an eigenbasis for $S^m(C)$, with corresponding eigenvalues $\lambda_1^{k_1} \cdots \lambda_n^{k_n}$. Computing the trace in this eigenbasis yields:

$$\sum_{m=0}^{\infty} \operatorname{Tr}(S^m(C)) = \sum_{m=0}^{\infty} \sum_{|k|=m} \lambda_1^{k_1} \cdots \lambda_n^{k_n} = \prod_{i=1}^{n} \left(\sum_{k_i=0}^{\infty} \lambda_i^{k_i} \right).$$

By formally evaluating the independent geometric series, we obtain:

$$\prod_{i=1}^{n} \frac{1}{1-\lambda_i} = \frac{1}{\det(I-C)}.$$

Thus, $F_m(C) = 0$ for all strictly diagonalizable matrices C. We have established that the polynomial F_m evaluates to zero wherever the non-zero polynomial Δ is non-zero, thus $F_m\Delta = 0$, so $F_m = 0$ as $\Delta \neq 0$ and we're over $\mathbb{C}$.

By substituting the variables $w_{ij} = a_{ij}x_j$ (which corresponds to $W = AX$) back into the identity for W, we obtain:

$$\sum_{m=0}^{\infty} \operatorname{Tr}(S^m(T)) = \frac{1}{\det(I-AX)}.$$

□

7 Set Power Series

7.1 Subset Convolution

Given two functions $f, g : 2^V \to \mathbb{F}$ over a set V of size n, their subset convolution $h = f * g$ is defined as:

$$h(S) = \sum_{A \subseteq S} f(A)g(S \setminus A)$$

Computing this naively requires $O(3^n)$ time. We can optimize this to $O(n^2 2^n)$ using the Zeta and Möbius transforms.

Define the ranked representations $\hat{f}$ and $\hat{g}$:

$$\hat{f}_k(S) = \begin{cases} f(S) & \text{if } |S| = k \\ 0 & \text{otherwise} \end{cases}$$

The Zeta transform computes: $\mathcal{Z}(f)(S) = \sum_{A \subseteq S} f(A)$. We have that point-wise multiplication after the Zeta transform computes:

$$\mathcal{Z}^{-1}\big(\mathcal{Z}(u) \cdot \mathcal{Z}(v)\big)(S) = \sum_{A \cup B = S} u(A)v(B)$$

By performing a standard linear convolution over the rank dimension and a point-wise multiplication in the Zeta domain, we compute a new ranked function $\hat{h}$:

$$\hat{h}_i(S) = \mathcal{Z}^{-1}\left(\sum_{j=0}^{i} \mathcal{Z}(\hat{f}_j) \cdot \mathcal{Z}(\hat{g}_{i-j})\right)(S) = \sum_{A \cup B = S} \hat{f}_j(A)\hat{g}_{i-j}(B)$$

To extract the final answer for a set S, we evaluate the rank slice corresponding to its cardinality: $h(S) = \hat{h}_{|S|}(S)$. By substituting $i = |S|$, the non-zero terms mandate that $|A| = j$ and $|B| = i - j$, which means $|A| + |B| = i = |S|$ so $A \cap B = \emptyset$.

Complexity

- **Time:** The Forward and Inverse Zeta Transforms process $n + 1$ arrays of size 2^n, taking $O(n \cdot n2^n)$ time. The ranked point-wise multiplication takes $O(n^2)$ for each of the 2^n sets. Total Time: $O(n^2 2^n)$.
- **Space:** Allocating the ranked arrays requires $(n + 1) \times 2^n$ memory. Total Space: $O(n2^n)$.

Algorithm 31: Subset Convolution

```
Function ZetaTransform(A, n):
    for i ← 0 to n − 1 do
        for S ← 0 to 2^n − 1 do
            if (S & (1 ≪ i)) ≠ 0 then
                A[S] ← A[S] + A[S ⊕ (1 ≪ i)];
Function MobiusTransform(A, n):
    for i ← 0 to n − 1 do
        for S ← 0 to 2^n − 1 do
            if (S & (1 ≪ i)) ≠ 0 then
                A[S] ← A[S] − A[S ⊕ (1 ≪ i)];
Function SubsetConvolution(f, g, n):
    Initialize f̂, ĝ, ĥ as arrays of size (n + 1) × 2^n filled with 0;
    for S ← 0 to 2^n − 1 do
        f̂[|S|][S] ← f[S];
        ĝ[|S|][S] ← g[S];
    for k ← 0 to n do
        ZetaTransform(f̂[k], n);
        ZetaTransform(ĝ[k], n);
    for i ← 0 to n do
        for j ← 0 to i do
            for S ← 0 to 2^n − 1 do
                ĥ[i][S] ← ĥ[i][S] + f̂[j][S] · ĝ[i − j][S];
    for k ← 0 to n do
        MobiusTransform(ĥ[k], n);
    Initialize h as an array of size 2^n;
    for S ← 0 to 2^n − 1 do
        h[S] ← ĥ[|S|][S];
    return h;
```

7.2 Transposed Subset Convolution

Given two functions $f, g : 2^V \to \mathbb{F}$ over a set V of size n, the Transposed Subset Convolution is equivalent to Superset Convolution, and computes h defined as:

$$h(S) = \sum_{Y \supseteq S} f(Y)g(Y \setminus S)$$

We reverse f to get $\tilde{f}$, then compute the subset convolution with g, then reverse the output.

To see why this yields the superset convolution, we expand the definition of $h(S)$ from the three steps above by substituting $K = U \setminus S$:

$$h(S) = \tilde{h}(U \setminus S) = \sum_{X \subseteq U \setminus S} f(U \setminus X)g((U \setminus S) \setminus X)$$

Letting $Y = U \setminus X$, the summation condition $X \subseteq U \setminus S$ becomes $S \subseteq Y$. The sum becomes:

$$h(S) = \sum_{S \subseteq Y} f(Y)g(Y \setminus S)$$

7.3 Inversion and Quotient

Given a subset power series $g : 2^V \to \mathbb{F}$ where $g(\emptyset) \neq 0$, its inverse $h = g^{-1}$ is the unique series satisfying $h * g = \varepsilon$, where $\varepsilon(\emptyset) = 1$ and $\varepsilon(S) = 0$ for all $S \neq \emptyset$. More generally, the quotient $h = f/g$ is the series satisfying $h * g = f$.

We can compute the quotient layer-by-layer (by rank) in $O(n^2 2^n)$ time by a DP directly in the Ranked Zeta Transform domain.

Expanding the definition of the subset convolution evaluated at a specific set X, we isolate the contribution of $h(X)$:

$$f(X) = \sum_{A \subseteq X} h(A)g(X \setminus A) = h(X)g(\emptyset) + \sum_{A \subsetneq X} h(A)g(X \setminus A)$$

Let $E(X) = \sum_{A \subsetneq X} h(A)g(X \setminus A)$. If we process the sets in increasing order of cardinality $l = |X|$, all required values of $h(A)$ for $|A| < l$ will have already been computed.

This allows us to inductively construct the ranked components $\hat{h}_l$ of the quotient. For a fixed rank l, the transformed accumulation of the smaller ranks is:

$$\mathcal{Z}(\hat{E}_l) = \sum_{i=1}^{l} \mathcal{Z}(\hat{h}_{l-i}) \cdot \mathcal{Z}(\hat{g}_i)$$

We recover E in the subset domain by applying the inverse Zeta (Möbius) transform. Then, for every set X with $|X| = l$, we extract the value of $h(X)$:

$$h(X) = \big(f(X) - E(X)\big) \cdot g(\emptyset)^{-1}$$

Finally, we apply the forward Zeta transform to $\hat{h}_l$ so that its transformed values $\mathcal{Z}(\hat{h}_l)$ are available for computing the subsequent ranks $l + 1, \ldots, n$.

Algorithm The computation proceeds iteratively over the rank dimension:

1. **Initialization:** Compute the inverse scalar $g(\emptyset)^{-1}$. Initialize $(n+1) \times 2^n$ arrays $\hat{g}$ and $\hat{h}$. Fill $\hat{g}$ by placing $g(S)$ into $\hat{g}_{|S|}[S]$ and apply the Fast Zeta Transform to every rank slice of $\hat{g}$.
2. **Layered DP:** For each rank l from 0 to n:
   (a) **Accumulate:** For each element S, compute $\hat{E}_l[S] = \sum_{i=1}^{l} \hat{h}_{l-i}[S] \cdot \hat{g}_i[S]$.
   (b) **Möbius Inversion:** Apply the Fast Möbius Transform to $\hat{E}_l$.
   (c) **Extract:** For every set X where $|X| = l$, compute $\hat{h}_l[X] = \big(f(X) - \hat{E}_l[X]\big) \cdot g(\emptyset)^{-1}$.
   (d) **Zeta Transform:** Apply the Fast Zeta Transform to $\hat{h}_l$.

For SPS inversion, this identical algorithm is used by setting $f(\emptyset) = 1$ and $f(X) = 0$ for all $|X| > 0$.

Algorithm 32: SPS Quotient

```
Function SPS_Quotient(f, g, n):
    g_inv ← g[0]^{-1};
    Initialize ĝ, ĥ as arrays of size (n + 1) × 2^n filled with 0;
    for S ← 0 to 2^n − 1 do
        ĝ[|S|][S] ← g[S];
    for k ← 0 to n do
        FastZetaTransform(ĝ[k], n);
    for l ← 0 to n do
        for i ← 1 to l do
            for S ← 0 to 2^n − 1 do
                ĥ[l][S] ← ĥ[l][S] + ĥ[l − i][S] · ĝ[i][S];
        FastMobiusTransform(ĥ[l], n);
        for X ← 0 to 2^n − 1 do
            if |X| == l then
                ĥ[l][X] ← (f[X] − ĥ[l][X]) · g_inv;
            else
                ĥ[l][X] ← 0 ;                    // Clear side-effects from the Möbius transform
        FastZetaTransform(ĥ[l], n);
    Initialize h as an array of size 2^n;
    for S ← 0 to 2^n − 1 do
        h[S] ← ĥ[|S|][S];
    return h;
```

7.4 Exponential and Logarithm

Given a subset power series $f : 2^V \to \mathbb{F}$, we can compute its exponential $h = \exp(f)$ and logarithm $h = \log(f)$ in $O(n^2 2^n)$ time. For the exponential to exist, we require $f(\emptyset) = 0$. For the logarithm to exist, we require $f(\emptyset) = 1$.

The coefficient of the partial derivative $\frac{\partial}{\partial x_i}$ on a subset S (where $i \notin S$) yields the coefficient of $S \cup \{i\}$ in the original series.

By the chain rule, the exponential $h = \exp(f)$ satisfies the differential equation:

$$\frac{\partial h}{\partial x_i} = h * \frac{\partial f}{\partial x_i}$$

We build the series h iteratively, adding elements from 0 to $n-1$. Let $h_{<i}$ denote the series restricted to subsets drawn from $\{0, \ldots, i-1\}$. When we introduce the i-th element, we want to compute the slice of h for subsets where i is the maximum element. We denote this slice as $h_{\text{slice}(i)}$.

By restricting the equation above to the domain of the first i elements, we have:

$$h_{\text{slice}(i)} = h_{<i} * f_{\text{slice}(i)}$$

This is a subset convolution of size 2^i. Since $f(\emptyset) = 0$, the i-th half of the output is determined by the i-th half of f and the already-computed lower half of h.

Similarly, for the logarithm $h = \log(f)$, the chain rule dictates:

$$\frac{\partial h}{\partial x_i} = \frac{\partial f}{\partial x_i} * f^{-1} \implies \frac{\partial f}{\partial x_i} = \frac{\partial h}{\partial x_i} * f$$

Restricting this to the new element i, we obtain:

$$f_{\text{slice}(i)} = h_{\text{slice}(i)} * f_{<i}$$

This means the new slice of the logarithm can be directly computed using the SPS quotient:

$$h_{\text{slice}(i)} = f_{\text{slice}(i)} / f_{<i}$$

Algorithm The algorithm iteratively computes the exact slices of size 2^i for $i = 0$ to $n-1$.

- **Exp:** We initialize $h[\emptyset] = 1$. At step i, we extract the previously computed half $h[0 \ldots 2^i - 1]$ and multiply it by $f[2^i \ldots 2^{i+1} - 1]$ using a Subset Convolution of size 2^i. We write the output into $h[2^i \ldots 2^{i+1} - 1]$.
- **Log:** We initialize $h[\emptyset] = 0$. At step i, we compute the Subset Quotient of $f[2^i \ldots 2^{i+1} - 1]$ divided by $f[0 \ldots 2^i - 1]$ and write the output into $h[2^i \ldots 2^{i+1} - 1]$.

Algorithm 33: SPS Exponential and Logarithm

```
Function SPS_Exp(f, n):
    if f[0] ≠ 0 then return Error;
    Initialize h as an array of size 2^n filled with 0;
    h[0] ← 1;
    for i ← 0 to n − 1 do
        h[2^i ... 2^{i+1} − 1] ← h[0 ... 2^i − 1] * f[2^i ... 2^{i+1} − 1];
    return h;

Function SPS_Log(f, n):
    if f[0] == 0 then return Error;
    Initialize h as an array of size 2^n filled with 0;
    for i ← 0 to n − 1 do
        h[2^i ... 2^{i+1} − 1] ← f[2^i ... 2^{i+1} − 1] * f[0 ... 2^i − 1]^{−1};
    return h;
```

Complexity For both operations, at the i-th iteration, we perform a subset convolution (or quotient) on arrays of length 2^i, which requires exactly $O(i^2 2^i)$ time. The total complexity is the sum:

$$T(n) = \sum_{i=0}^{n-1} O(i^2 2^i) = O(n^2 2^n)$$

7.5 Composition

Given a formal power series $F(x) = \sum_{k=0}^{m-1} f_k x^k$ and a subset power series $G : 2^V \to \mathbb{F}$ over n variables, their composition $H(S) = F(G(S))$ can be computed in $O(n^2 2^n)$ time.

1. Handling $G(\emptyset) \neq 0$ Subset composition requires $G(\emptyset) = 0$ to converge. We first apply the Inverse Borel transform to F, which maps $f_i \to i!f_i = F^{(i)}(0)$.

If the constant term is non-zero (let $a = G(\emptyset)$), we must shift the power series. We have $F^{(i)}(a) = \sum_{j \geq i} F^{(j)}(0) \frac{a^{j-i}}{(j-i)!}$. This shift is computed as a semi-correlation of the transformed coefficients with $\exp(ax)$. Once shifted, we set $G(\emptyset) = 0$.

2. Taylor Expansion We evaluate $F(G)$ by introducing the elements of the set one by one. Let $G_{<k}$ denote the subset power series restricted to the first k elements, and let $G_{\text{slice}(k)}$ be the slice of G where the k-th element is the strict maximum (with x_k factored out). We can iteratively decompose G:

$$G_{\leq k} = G_{<k} + x_k G_{\text{slice}(k)}$$

We use the Taylor expansion of F evaluated at $A + B$, where $A = G_{<k}$ and $B = x_k G_{\text{slice}(k)}$:

$$F(A + B) = F(A) + B * F'(A) + \frac{B * B}{2!} * F''(A) + \dots$$

In subset convolution, elements must be disjoint to yield a non-zero product and every term in B contains the element x_k, thus we only need to consider the first derivative:

$$F(G_{\leq k}) = F(G_{<k}) + x_k G_{\text{slice}(k)} * F'(G_{<k})$$

Notice that computing $F(G_{\leq k})$ requires $F'(G_{<k})$. Consequently, computing $F'(G_{<k})$ will require $F''(G_{<k-1})$, etc. The transition for evaluating the i-th derivative is:

$$F^{(i)}(G_{\leq k}) = F^{(i)}(G_{<k}) + x_k G_{\text{slice}(k)} * F^{(i+1)}(G_{<k})$$

3. DP Optimization We maintain a single working array C of length 2^n. We iterate the derivative order i downwards from n to 0. For a fixed i, our goal is to transform the state of C from representing $F^{(i+1)}(G)$ into $F^{(i)}(G)$.

We iterate the introduced variable index j downwards. At the start of step j, the lower prefix $C[0 \dots 2^j - 1]$ holds the evaluation of $F^{(i+1)}(G_{<j})$. We compute its subset convolution with $G_{\text{slice}(j)}$ and write the result into the upper half $C[2^j \dots 2^{j+1} - 1]$.

Because we iterate j downwards, we compute the upper halves first, preventing the overwriting of the smaller prefixes. We then set the base case (the empty set evaluated at the i-th derivative) by assigning $C[\emptyset] = f_i$. This is correct as our coefficients were mapped to $F^{(i)}(a)$.

Algorithm 34: SPS Composition

```
Function SPS_Compose(F, G, n, m):
    F ← InverseBorel(F);
    a ← G[0];
    if a ≠ 0 then
        F ← SemiCorr(F, exp(ax));
        G[0] ← 0;
    Initialize C as an array of size 2^n filled with 0;
    for i ← n to 0 step −1 do
        for j ← n − i − 1 to 0 step −1 do
            X ← C[0 ... 2^j − 1];
            G_slice(j) ← G[2^j ... 2^{j+1} − 1];
            C[2^j ... 2^{j+1} − 1] ← X * G_slice(j);
        if i < m then C[0] ← F[i];
        else C[0] ← 0;
    return C;
```

Complexity The Inverse Borel Transform takes $O(m)$ time, and the Taylor shift takes $O(m \log m)$ time. The outer loop executes $O(n)$ times, and the inner loop performs a subset convolution on slices of size 2^j, requiring $O(j^2 2^j)$ operations. Summing this over i and j gives $O(n^2 2^n)$. Thus, the complexity is $O(n^2 2^n + m \log m)$.

7.6 Power Projection

Given a subset power series $G : 2^V \to \mathbb{F}$ over n variables and a weight function $W : 2^V \to \mathbb{F}$, the Power Projection problem asks us to compute the sequence of scalars c_k for $k = 0 \dots m - 1$, defined as the inner product of W with the convolution powers of G:

$$c_k = \langle W, G^k \rangle = \sum_{S \subseteq V} W(S)[x^S]G^k$$

This is the transpose of the subset composition algorithm. Let $L_G(F) = F(G)$ be the operator defined by subset composition. Its adjoint L_G^T maps the vector W to the polynomial $C(x) = \sum c_k x^k$ such that $\langle W, L_G(F) \rangle = \langle L_G^T(W), F \rangle$.

Assume $G(\emptyset) = 0$. In the forward composition DP, the state vector $C \in \mathbb{F}^{2^n}$ is updated iteratively by nesting loops: i downwards from n to 0, and j downwards from $n - i - 1$ to 0, we reverse this in the transposed algorithm.

Let X denote the lower half of the state $(0 \dots 2^j - 1)$ and Y denote the upper half $(2^j \dots 2^{j+1} - 1)$. The forward transition at step j is the map:

$$Y \leftarrow G_{\text{slice}(j)} * X$$

Let X^* and Y^* be the corresponding halves in the transposed algorithm. The adjoint of reading from X to overwrite Y is to project the dual values from Y^* through the transposed operator and accumulate them into X^*, then to clear Y^*.

$$X^* \leftarrow X^* + G_{\text{slice}(j)} *^T Y^*$$
$$Y^* \leftarrow 0$$

Finally, the forward DP puts the input coefficient f_i into $C[\emptyset]$ at the conclusion of the j iterations. The transposed operation extracts this value into the output $c_i = W[\emptyset]$ and clears $W[\emptyset] = 0$ at the start of the transposed iterations.

If the original series had a non-zero constant term $a = G(\emptyset)$, the forward composition shifted $F^{(i)}(0)$ to $F^{(i)}(a)$. The algorithm computes the semicorrelation with $\exp(ax)$, the transpose of which is multiplication with $\exp(ax)$.

In the forward algorithm, we applied the Inverse Borel transform, since this is diagonal, it's self-adjoint.

Algorithm 35: SPS Power Projection

```
Function SPS_PowerProj(G, W, n, m):
    a ← G[0];
    G[0] ← 0;
    Initialize C as an array of size n + 1 filled with 0;
    Initialize S as an intermediate array of size 2^n;
    for i ← 0 to n do
        C[i] ← W[0];
        W[0] ← 0;
        for j ← 0 to n − i − 1 do
            X ← W[0 ... 2^j − 1];
            Y ← W[2^j ... 2^{j+1} − 1];
            G_slice(j) ← G[2^j ... 2^{j+1} − 1];
            S ← Y *^T G_slice(j);
            X ← X + S;
            Y ← 0;
    if a ≠ 0 then
        C ← C · exp(ax) (mod x^m);
    C ← InverseBorel(C);
    return C;
```

7.7 Power

Given a subset power series $f : 2^V \to \mathbb{F}$ over n variables, computing the k-th power $h = f^k$ can be done in $O(n^2 2^n)$ time.

Case 1: $f(\emptyset) \neq 0$ If the constant term $a_0 = f(\emptyset)$ is non-zero, we can compute the power using the exponential and logarithm.

$$f^k = a_0^k \exp\left(k \log\left(\frac{f}{a_0}\right)\right)$$

Case 2: $f(\emptyset) = 0$ If the constant term is zero, every non-zero term in the series corresponds to a subset of size ≥ 1. Consequently, convolving the series with itself k times means every term in the result must have a subset size of at least k. Thus, if $k > n$, f^k is identically zero.

If $k \leq n$, we compute the power as a composition: $h(S) = P(f(S))$, where $P(x) = x^k$.

We can specialize our SPS Composition algorithm. We have:

$$P^{(i)}(0) = \begin{cases} k! & \text{if } i = k \\ 0 & \text{if } i < k \end{cases}$$

We C with $C[\emptyset] = k!$, which represents $P^{(k)}(f)$. We then run the DP downwards from $i = k - 1$ to 0. At the end of each derivative step, we set the base case $C[\emptyset] = P^{(i)}(0) = 0$.

Algorithm 36: SPS Power

```
Function SPS_Power(f, k, n):
    a_0 ← f[0];
    if a_0 ≠ 0 then
        a_0^inv ← (a_0)^{-1};
        f_norm ← f · a_0^inv;
        L ← SPS_Log(f_norm, n);
        L ← L · k;
        E ← SPS_Exp(L, n);
        return E · (a_0^k);
    else
        if k > n then return Array of 0s;
        Initialize C as an array of size 2^n filled with 0;
        C[0] ← k!;
        for i ← k − 1 to 0 step −1 do
            for j ← n − i − 1 to 0 step −1 do
                X ← C[0 ... 2^j − 1];
                f_slice(j) ← f[2^j ... 2^{j+1} − 1];
                C[2^j ... 2^{j+1} − 1] ← SubsetConvolution(X, f_slice(j));
            C[0] ← 0;
        return C;
```

Complexity If $f(\emptyset) \neq 0$, we perform one subset logarithm and one subset exponential, taking $O(n^2 2^n)$ time. If $f(\emptyset) = 0$, the DP matches the complexity of composition, and thus is also $O(n^2 2^n)$ time.

8 Matrix Analysis

8.1 Schur Complements

Lemma 8.1 (Schur Complement Positivity)**.** *For $B \succ 0$, $\begin{pmatrix} A & X \\ X^* & B \end{pmatrix} \succeq 0 \iff A \succeq XB^{-1}X^*$.*

Proof. Conjugating a block matrix by an invertible matrix preserves positive semidefiniteness. We calculate:

$$\begin{pmatrix} I & -XB^{-1} \\ 0 & I \end{pmatrix} \begin{pmatrix} A & X \\ X^* & B \end{pmatrix} \begin{pmatrix} I & 0 \\ -B^{-1}X^* & I \end{pmatrix} = \begin{pmatrix} A - XB^{-1}X^* & 0 \\ 0 & B \end{pmatrix}$$

Because $B \succ 0$, the block diagonal matrix on the right is positive semidefinite if and only if $A - XB^{-1}X^* \succeq 0$. □

Lemma 8.2 (Block Matrix Inversion)**.** *Let $M = \begin{pmatrix} A & X \\ Y & B \end{pmatrix}$ be a block matrix where B is invertible. Let $M/B = A - XB^{-1}Y$ denote the Schur complement of B in M. If M/B is also invertible, then M is invertible and its inverse is given by:*

$$M^{-1} = \begin{pmatrix} (M/B)^{-1} & -(M/B)^{-1}XB^{-1} \\ -B^{-1}Y(M/B)^{-1} & B^{-1} + B^{-1}Y(M/B)^{-1}XB^{-1} \end{pmatrix}$$

Proof. We can factor M into a block lower-triangular, block-diagonal, and block upper-triangular matrix (the LDU decomposition) as follows:

$$M = \begin{pmatrix} A & X \\ Y & B \end{pmatrix} = \begin{pmatrix} I & XB^{-1} \\ 0 & I \end{pmatrix} \begin{pmatrix} M/B & 0 \\ 0 & B \end{pmatrix} \begin{pmatrix} I & 0 \\ B^{-1}Y & I \end{pmatrix}$$

Because the outer block triangular matrices have identity blocks on their diagonals, they are always invertible. Thus, M is invertible if and only if both B and M/B are invertible.

To find M^{-1}, we invert the three factors in reverse order:

$$M^{-1} = \begin{pmatrix} I & 0 \\ -B^{-1}Y & I \end{pmatrix} \begin{pmatrix} (M/B)^{-1} & 0 \\ 0 & B^{-1} \end{pmatrix} \begin{pmatrix} I & -XB^{-1} \\ 0 & I \end{pmatrix}$$

Performing the matrix multiplication yields:

$$M^{-1} = \begin{pmatrix} (M/B)^{-1} & 0 \\ -B^{-1}Y(M/B)^{-1} & B^{-1} \end{pmatrix} \begin{pmatrix} I & -XB^{-1} \\ 0 & I \end{pmatrix}$$

$$M^{-1} = \begin{pmatrix} (M/B)^{-1} & -(M/B)^{-1}XB^{-1} \\ -B^{-1}Y(M/B)^{-1} & B^{-1} + B^{-1}Y(M/B)^{-1}XB^{-1} \end{pmatrix}$$

□

8.2 Continuity of Eigenvalues

Theorem 8.3. *The roots of a monic complex polynomial of degree n depend continuously on its coefficients.*

Proof. Let $f : \mathbb{C}^n \to \mathbb{C}^n$ be the map taking an n-tuple of roots $(a_1, \ldots, a_n) \in \mathbb{C}^n$ to the coefficients $(c_0, \ldots, c_{n-1})$ of the monic polynomial

$$P(x) = \prod_{i=1}^{n}(x - a_i) = x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0.$$

Because the coefficients c_k are elementary symmetric polynomials in the roots a_i, the map f is clearly continuous. Furthermore, f is invariant under any permutation of the a_i. Therefore, it descends to a continuous quotient map $g : \mathbb{C}^n/S_n \to \mathbb{C}^n$, where the symmetric group S_n acts on $\mathbb{C}^n$ by permuting the coordinates.

By the Fundamental Theorem of Algebra, every monic polynomial of degree n over $\mathbb{C}$ factors uniquely into n linear factors (up to the ordering of the factors). This implies that g is a bijection. To establish that g is a homeomorphism, it remains to prove that its inverse g^{-1} is continuous.

We can prove this by compactifying the spaces. We consider $\mathbb{C}$ as a subspace of the complex projective line $\mathbb{CP}^1$, and $\mathbb{C}^n$ as a subspace of $\mathbb{CP}^n$. We can identify $\mathbb{CP}^1$ with the projectivization of the space of homogeneous linear polynomials in two variables (x, y), where $[u : v]$ corresponds to the polynomial $ux - vy$. Similarly, we identify $\mathbb{CP}^n$ with the projectivization of the space of homogeneous degree n polynomials in x and y.

We extend f to a map $F : (\mathbb{CP}^1)^n \to \mathbb{CP}^n$ by mapping n homogeneous linear polynomials to their product:

$$F([u_1 : v_1], \ldots, [u_n : v_n]) = \left[\prod_{i=1}^{n}(u_i x - v_i y)\right].$$

Notice that F extends f: if we restrict to $u_i = 1$, the roots are $v_i = a_i$, and setting $y = 1$ recovers our original polynomial $\prod_{i=1}^{n}(x - a_i)$.

Because polynomial multiplication is continuous, F is continuous. Like f, F is invariant under the action of S_n permuting the inputs. Hence, F descends to a continuous map $G : (\mathbb{CP}^1)^n/S_n \to \mathbb{CP}^n$ extending g. By the Fundamental Theorem of Algebra, G is also a bijection.

Now we use that any continuous bijection from a compact space to a Hausdorff space is automatically a homeomorphism. The space $(\mathbb{CP}^1)^n$ is compact (by Tychonoff's Theorem), so its quotient $(\mathbb{CP}^1)^n/S_n$ is also compact. The projective space $\mathbb{CP}^n$ is Hausdorff. Therefore, G is a homeomorphism.

Finally, we must verify that restricting the homeomorphism G to our original spaces yields a homeomorphism g. Specifically, we need to ensure that the subspace topology on $\mathbb{C}^n/S_n$ inherited from $(\mathbb{CP}^1)^n/S_n$ is identical to the quotient topology from the projection $\mathbb{C}^n \to \mathbb{C}^n/S_n$.

Because $\mathbb{C}^n$ is an open subset of $(\mathbb{CP}^1)^n$ and is strictly invariant under the action of S_n, an S_n-invariant open subset of $\mathbb{C}^n$ is exactly the same thing as an S_n-invariant open subset of $(\mathbb{CP}^1)^n$ that happens to be entirely contained within $\mathbb{C}^n$. Therefore, the two topologies coincide.

Since G is a homeomorphism, its restriction g to the open subspace $\mathbb{C}^n/S_n$ is a homeomorphism onto its image $\mathbb{C}^n$. Thus, g^{-1} is continuous, meaning the roots of a polynomial depend continuously on its coefficients. □

Corollary 8.4. *Let $A \in M_n(\mathbb{C})$. The eigenvalues of A depend continuously on the entries of A.*

Proof. Let $A \mapsto P_A(x) = \det(xI - A)$ map a matrix to its characteristic polynomial. The coefficients of $P_A(x)$ are multivariate polynomials in the entries of A, so this mapping is continuous. By the previous theorem, the map from the coefficients of $P_A(x)$ to its unordered n-tuple of roots in $\mathbb{C}^n/S_n$ is also continuous. □

8.3 Sampling Matrices without Replacement

We can show that sampling without replacement is more concentrated than sampling with replacement, this works for both scalars and matrices.

Lemma 8.5 (Hoeffding's Coupling)**.** *Let $\mathcal{C}$ be a finite set of Hermitian matrices. Let $X = (X_1, \ldots, X_m)$ be a sequence of m matrices sampled uniformly with replacement from $\mathcal{C}$. Let $Y = (Y_1, \ldots, Y_m)$ be a sequence sampled uniformly without replacement from $\mathcal{C}$.*

There exists a randomized surrogate function $Z(Y) = (Z_1, \ldots, Z_m)$ such that:

1. *The unconditional distribution of $Z(Y)$ is identical to X.*
2. *The conditional expectation satisfies $\mathbb{E}_{Z|Y}\left[\sum_{i=1}^{m} Z_i(Y)\right] = \sum_{i=1}^{m} Y_i$.*

Proof. We construct the sequence $Z_1, \ldots, Z_m$ conditionally from the drawn set Y via the following sequential recipe.

For the k-th step, let D_k be the subset of matrices from Y that have already been chosen in the sequence $Z_1, \ldots, Z_{k-1}$. We draw Z_k as follows:

- With probability $\frac{|D_k|}{|\mathcal{C}|}$, we sample Z_k uniformly from the already-drawn elements D_k.
- With probability $1 - \frac{|D_k|}{|\mathcal{C}|}$, we sample Z_k uniformly from the remaining un-drawn elements $Y \setminus D_k$.

It is straightforward to verify that the joint probability of drawing any sequence $x \in \mathcal{C}^m$ is $|\mathcal{C}|^{-m}$.

For the second property, observe that the conditional expectation $\mathbb{E}_{Z|Y}[\sum_{i=1}^{m} Z_i]$ is a linear combination of the variables $Y_1, \ldots, Y_m$ and invariant under a permutation of the indices of Y.

It thus must take the form $K \sum_{i=1}^{m} Y_i$ for some scalar constant K. To determine K, we take the expectation over Y:

$$\mathbb{E}_Y\left[\mathbb{E}_{Z|Y}\left[\sum_{i=1}^{m} Z_i\right]\right] = m\mathbb{E}[Y_1] \quad \text{and} \quad \mathbb{E}_Y\left[K\sum_{i=1}^{m} Y_i\right] = Km\mathbb{E}[Y_1]$$

So $K = 1$. □

8.4 Schur and Kronecker Product

Lemma 8.6 (Kronecker Product Preserves Positivity)**.** *Let A and B be Hermitian matrices. If $A, B \succeq 0$, then $A \otimes B \succeq 0$. Furthermore, if $A, B \succ 0$, then $A \otimes B \succ 0$.*

Proof. Assume $A \succeq 0$ and $B \succeq 0$. Every PSD matrix has a unique PSD square root. We can factor the Kronecker product:

$$A \otimes B = (A^{1/2}A^{1/2}) \otimes (B^{1/2}B^{1/2}) = (A^{1/2} \otimes B^{1/2})(A^{1/2} \otimes B^{1/2})$$

Because the Kronecker product of two Hermitian matrices is Hermitian, $A^{1/2} \otimes B^{1/2}$ is Hermitian. Therefore, we have expressed $A \otimes B$ as the square of a Hermitian matrix, which implies $A \otimes B \succeq 0$.

Now assume $A \succ 0$ and $B \succ 0$. In this case, both A and B are invertible. The Kronecker product of invertible matrices is invertible, with its inverse given by $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. □

Lemma 8.7 (Schur Product Theorem)**.** *Let $A, B \in M_n(\mathbb{C})$ be Hermitian matrices. Let $A \circ B$ denote their Schur (entrywise) product, defined by $(A \circ B)_{ij} = A_{ij}B_{ij}$. If $A, B \succeq 0$, then $A \circ B \succeq 0$. Furthermore, if $A, B \succ 0$, then $A \circ B \succ 0$.*

Proof. The Schur product $A \circ B$ can be realized as a principal submatrix of the Kronecker product $A \otimes B$. Let e_i denote the i-th standard basis vector of $\mathbb{C}^n$. Define the $n^2 \times n$ matrix P such that its i-th column is $e_i \otimes e_i$.

The (i, j)-th entry of the matrix $P^*(A \otimes B)P$ evaluates to:

$$[P^*(A \otimes B)P]_{ij} = (e_i \otimes e_i)^*(A \otimes B)(e_j \otimes e_j)$$

This becomes:

$$[P^*(A \otimes B)P]_{ij} = (e_i^* A e_j)(e_i^* B e_j) = A_{ij}B_{ij} = [A \circ B]_{ij}$$

Thus, we have the identity $A \circ B = P^*(A \otimes B)P$.

Assume $A, B \succeq 0$. By Lemma 8.6, $A \otimes B \succeq 0$. For any vector $x \in \mathbb{C}^n$, the quadratic form is:

$$x^*(A \circ B)x = x^*P^*(A \otimes B)Px = (Px)^*(A \otimes B)(Px)$$

Because $A \otimes B \succeq 0$, this quadratic form is always non-negative, proving $A \circ B \succeq 0$.

Now assume $A, B \succ 0$. By Lemma 8.6, $A \otimes B \succ 0$. Notice that the columns of P are $e_i \otimes e_i$, so P has full rank. For any non-zero vector $x \in \mathbb{C}^n$, the vector $y = Px$ must be non-zero, so the quadratic form $y^*(A \otimes B)y > 0$. □

8.5 Integral Representations

For positive-definite matrices $A > 0$, we have the following integral representations, where C_p denotes a positive constant depending only on p:

- For $-1 < p < 0$:
$$A^p = C_p \int_0^\infty t^p (tI + A)^{-1} dt$$
- For $0 < p < 1$:
$$A^p = C_p \int_0^\infty t^p \left(\frac{1}{t} I - (tI + A)^{-1} \right) dt$$
- For $1 < p < 2$:
$$A^p = C_{p-1} \int_0^\infty t^{p-1} \left(\frac{A}{t} + t(tI + A)^{-1} - I \right) dt$$
$$\log A = \int_0^\infty \left[(1 + u)^{-1} I - (A + uI)^{-1} \right] du$$
- For $\alpha \in (0, 2)$:
$$|x|^\alpha = c_\alpha \int_0^\infty \frac{1 - \cos(tx)}{t^{\alpha+1}} dt$$

Furthermore, we have the following limits as in the scalar case:

$$\log A = \lim_{p \to 0} \frac{A^p - I}{p}$$
$$A \log A = \lim_{p \to 1} \frac{A^p - A}{p - 1}$$

8.6 Golden-Thompson Inequality

Lemma 8.8. *Consider $A \in \mathbb{M}_d(\mathbb{C})$, and suppose that $A_i \in \{A, A^*\}$ for each $i = 1, 2, \ldots, 2n$. Then,*

$$|\operatorname{Tr}(A_1 A_2 \ldots A_{2n})| \leq \operatorname{Tr}((A^*A)^n).$$

Proof. We may assume that $A \neq A^*$. Let $\mathcal{P}_n$ denote the set of all such length-$2n$ products $P = A_1 A_2 \ldots A_{2n}$. We define the number of transitions in P as the number of times in the cyclic order that AA^* or A^*A occurs (i.e., the number of indices where $A_{i \pmod{2n}} \neq A_{(i+1) \pmod{2n}}$).

Let $P = A_1 A_2 \ldots A_{2n}$ be a product that maximizes $|\operatorname{Tr}(P)|$ among all $P \in \mathcal{P}_n$. If the number of transitions in P is exactly $2n$, then P simply alternates perfectly, meaning $P = (A^*A)^n$ or $P = (AA^*)^n$, and the bound holds.

Otherwise, P must contain an adjacent pair of identical symbols. Because the trace is invariant under cyclic permutations, we may cycle the product so that this identical pair appears at the split: $A_n = A_{n+1}$. Let us write

$P = QR$, where $Q = A_1 \dots A_n$ and $R = A_{n+1} \dots A_{2n}$. Define $P' = Q^*Q$ and $P'' = R^*R$, both of which are valid products in $\mathcal{P}_n$. By the Cauchy-Schwarz inequality for the Frobenius inner product, we have:

$$|\operatorname{Tr}(P)|^2 = |\operatorname{Tr}(QR)|^2 \le |\operatorname{Tr}(Q^*Q)||\operatorname{Tr}(R^*R)| = |\operatorname{Tr}(P')||\operatorname{Tr}(P'')|.$$

Since P is a maximizer, we must have $|\operatorname{Tr}(P)| = |\operatorname{Tr}(P')| = |\operatorname{Tr}(P'')|$.

Now we count the transitions. Let n_Q and n_R be the number of transitions internal to Q and R, respectively. Because $A_n = A_{n+1}$, there is no transition in the middle, meaning P has at most $n_Q + n_R + 1$ transitions (accounting for the cyclic wrap-around from A_{2n} to A_1).

On the other hand, the product $P' = Q^*Q$ contains all n_Q transitions of Q, all n_Q transitions of Q^*, plus two guaranteed transitions at the boundary between Q and Q^*. Thus, $N_{P'} \ge 2n_Q + 2$. By identical logic, $N_{P''} \ge 2n_R + 2$.

Taking the average gives $(N_{P'} + N_{P''})/2 \ge n_Q + n_R + 2 > N_P$. This implies that one of our new maximizers, P' or P'', has more transitions than P. By iterating this process, we reach a maximizer with exactly $2n$ transitions. □

Lemma 8.9 (Disentangling Lemma)**.** *For every integer $k \ge 1$ and Hermitian matrices $U, V \in \mathbb{M}_d(\mathbb{C})$, it holds that*

$$\operatorname{Tr}\left((U^2V^2)^{2^{k-1}}\right) \le \operatorname{Tr}\left((U^4V^4)^{2^{k-2}}\right) \le \cdots \le \operatorname{Tr}(U^{2^k}V^{2^k}).$$

Proof. Define $A = UV$. Because U and V are Hermitian, we can apply the preceding lemma to bound the trace of 2^k copies of A:

$$\operatorname{Tr}((UV)^{2^k}) \le \operatorname{Tr}\left((A^*A)^{2^{k-1}}\right) = \operatorname{Tr}\left((V^*U^*UV)^{2^{k-1}}\right) = \operatorname{Tr}\left((VU^2V)^{2^{k-1}}\right).$$

Using the cyclic property of the trace, we can move a V from the front of the inner grouping to the back, yielding:

$$\operatorname{Tr}((UV)^{2^k}) \le \operatorname{Tr}\left((U^2V^2)^{2^{k-1}}\right).$$

Continuing this procedure inductively gives:

$$\operatorname{Tr}\left((U^2V^2)^{2^{k-1}}\right) \le \operatorname{Tr}\left((U^4V^4)^{2^{k-2}}\right) \le \cdots \le \operatorname{Tr}(U^{2^k}V^{2^k}).$$

□

Theorem 8.10 (Golden-Thompson)**.** *For all Hermitian matrices $A, B \in \mathbb{M}_d$, we have $\operatorname{Tr}(e^{A+B}) \le \operatorname{Tr}(e^Ae^B)$.*

Proof. The Lie-Trotter product formula states that for any matrices A, B,

$$e^{A+B} = \lim_{N\to\infty} \left(e^{A/N}e^{B/N}\right)^N.$$

The proof is by Taylor expansion. By setting $N = 2^k$, we can define the Hermitian matrices $U = e^{A/2^k}$ and $V = e^{B/2^k}$. Applying the Disentangling Lemma yields:

$$\operatorname{Tr}(e^{A+B}) = \lim_{k\to\infty} \operatorname{Tr}\left(\left(e^{A/2^k}e^{B/2^k}\right)^{2^k}\right) \le \lim_{k\to\infty} \operatorname{Tr}\left(e^Ae^B\right) = \operatorname{Tr}(e^Ae^B).$$

□

8.7 Hölder Inequality for Trace

Theorem 8.11 (Hölder Inequality for Trace)**.** *For any integer $k \ge 1$ and matrices $A_1, \dots, A_{2^k} \in \mathbb{M}_d(\mathbb{C})$,*

$$|\operatorname{Tr}(A_1A_2 \dots A_{2^k})| \le \prod_{i=1}^{2^k} \|A_i\|_{2^k}$$

where $\|A\|_p = (\operatorname{Tr}((A^*A)^{p/2}))^{1/p}$ is the Schatten p-norm. This extends to arbitrary $p_i > 0$ with $\sum p_i^{-1} = 1$.

Proof. We proceed by induction on k. The base case $k = 1$ is exactly the Cauchy-Schwarz inequality for the Frobenius inner product: $|\operatorname{Tr}(A_1A_2)| \le \|A_1\|_2\|A_2\|_2$.

Assume the claim holds for $k-1$. Grouping adjacent matrices gives:

$$|\operatorname{Tr}(A_1A_2 \dots A_{2^k})| \le \prod_{j=1}^{2^{k-1}} \|A_{2j-1}A_{2j}\|_{2^{k-1}}.$$

We bound each of these pairwise factors by expanding the definition and utilizing the cyclic property of the trace:

$$\|A_1A_2\|_{2^{k-1}}^{2^{k-1}} = \operatorname{Tr}\left(((A_1A_2)^*(A_1A_2))^{2^{k-2}}\right) = \operatorname{Tr}\left((A_2^*A_1^*A_1A_2)^{2^{k-2}}\right) = \operatorname{Tr}\left((A_1^*A_1A_2A_2^*)^{2^{k-2}}\right).$$

We can now apply the inductive hypothesis to this product:

$$\|A_1A_2\|_{2^{k-1}}^{2^{k-1}} \le \|A_1^*A_1\|_{2^{k-1}}^{2^{k-2}}\|A_2A_2^*\|_{2^{k-1}}^{2^{k-2}} = \|A_1\|_{2^k}^{2^{k-1}}\|A_2\|_{2^k}^{2^{k-1}}.$$

Thus $\|A_1A_2\|_{2^{k-1}} \le \|A_1\|_{2^k}\|A_2\|_{2^k}$. Substituting this back into our initial sequence completes the proof.

To generalize to arbitrary exponents $p_1, \dots, p_n > 0$ that sum to 1, we construct a sequence of blocks by setting $N = 2^m$ and $k_i = \lfloor N/p_i \rfloor$. By expressing the product using fractional powers A_i^{1/k_i} repeated k_i times, we can apply the 2^m bound and recover the generic Hölder inequality by taking the limit as $m \to \infty$. □

8.8 Generalized Klein Inequality

Lemma 8.12 (Generalized Klein Inequality)**.** *Let $f_i, g_i : I \to \mathbb{R}$ be functions on an interval I such that $\sum_i f_i(x)g_i(y) \geq 0$ for all $x, y \in I$. If A and H are Hermitian matrices with eigenvalues in I, then $\sum_i \operatorname{Tr}[f_i(A)g_i(H)] \geq 0$.*

Proof. Consider the eigenvalue decompositions $A = \sum_j \lambda_j u_j u_j^*$ and $H = \sum_k \mu_k v_k v_k^*$. We evaluate the sum of traces:

$$\sum_i \operatorname{Tr}[f_i(A)g_i(H)] = \sum_i \operatorname{Tr}\left[\left(\sum_j f_i(\lambda_j)u_j u_j^*\right)\left(\sum_k g_i(\mu_k)v_k v_k^*\right)\right]$$

Using the linearity and cyclic property of the trace, this simplifies to:

$$\sum_{j,k}\left[\sum_i f_i(\lambda_j)g_i(\mu_k)\right]\operatorname{Tr}(u_j u_j^* v_k v_k^*) = \sum_{j,k}\left[\sum_i f_i(\lambda_j)g_i(\mu_k)\right]|\langle u_j, v_k\rangle|^2$$

By assumption, the bracketed sum is non-negative for all eigenvalue pairs (λ_j, μ_k), and the squared inner product $|\langle u_j, v_k\rangle|^2 \geq 0$. Thus, the entire sum is non-negative. □

8.9 Von Neumann's Trace Inequality

Lemma 8.13 (Von Neumann's Trace Inequality)**.** *If $A, B \in \mathbb{M}_n(\mathbb{C})$ have singular values $\sigma_1(A) \geq \cdots \geq \sigma_n(A)$ and $\sigma_1(B) \geq \cdots \geq \sigma_n(B)$, then*

$$|\operatorname{Tr}(A^*B)| \leq \sum_{i=1}^n \sigma_i(A)\sigma_i(B).$$

Proof. By Cauchy-Schwarz, we can bound $|\operatorname{Tr}(A^*B)| \leq (\operatorname{Tr}(|A||B|)\operatorname{Tr}(|A^*||B^*|))^{1/2}$, which allows us to assume without loss of generality that A and B are PSD matrices.

Write their spectral decompositions as $A = \sum_{i=1}^n \sigma_i(A)u_i u_i^*$ and $B = \sum_{j=1}^n \sigma_j(B)v_j v_j^*$, where $\{u_i\}$ and $\{v_j\}$ are orthonormal bases. Evaluating the trace of their product gives:

$$\operatorname{Tr}(AB) = \sum_{i=1}^n\sum_{j=1}^n \sigma_i(A)\sigma_j(B)|\langle u_i, v_j\rangle|^2.$$

Define the matrix $M_{ij} = |\langle u_i, v_j\rangle|^2$. Because both $\{u_i\}$ and $\{v_j\}$ are orthonormal bases, every row and column of M sums to 1, meaning M is a doubly stochastic matrix.

By the Birkhoff-von Neumann theorem, M can be expressed as a convex combination of permutation matrices, meaning $M_{ij} = \sum_\pi w_\pi I_{i,\pi(j)}$ for some weights $w_\pi \geq 0$ that sum to 1. Substituting this back into the trace yields:

$$\operatorname{Tr}(AB) = \sum_\pi w_\pi \sum_{i=1}^n \sigma_i(A)\sigma_{\pi(i)}(B) \leq \max_\pi \sum_{i=1}^n \sigma_i(A)\sigma_{\pi(i)}(B).$$

Because the sequences of singular values $\sigma_i(A)$ and $\sigma_i(B)$ are both sorted in decreasing order, the rearrangement inequality gives that the maximum is achieved by the identity. □

8.10 Wielandt Minimax Formula

Theorem 8.14 (Wielandt Minimax Formula)**.** *Let A be an $n \times n$ Hermitian matrix with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. For any sequence of indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$,*

$$\sum_{j=1}^k \lambda_{i_j}(A) = \max_{\substack{V_1 \subset \cdots \subset V_k \\ \dim(V_j) = i_j}} \min_{W \in X(V_1,\ldots,V_k)} \operatorname{tr}_W(A)$$

where $X(V_1, \ldots, V_k)$ denotes the set of k-dimensional subspaces W that admit a basis $v_1, \ldots, v_k$ with $v_j \in V_j$.

Proof. **The $\leq$ case:** Let $V_j = \operatorname{span}(e_1, \ldots, e_{i_j})$, where e_i are the eigenvectors of A. For any $W \in X(V_1, \ldots, V_k)$, it remains to show that $\sum_{j=1}^k \lambda_{i_j}(A) \leq \operatorname{tr}_W(A)$.

To show this, we construct an orthonormal set of vectors $v_1, \ldots, v_k$ such that $v_j \in V_j \cap W$. Since $\dim(V_1 \cap W) \geq 1$, we pick any unit vector $v_1 \in V_1 \cap W$. Next, since $\dim(V_2 \cap W) \geq 2$, we pick any unit vector $v_2 \in V_2 \cap W$ that is perpendicular to v_1, and so on. Then we have

$$\operatorname{tr}_W(A) \geq \sum_{j=1}^k \langle v_j, Av_j\rangle \geq \sum_{j=1}^k \lambda_{i_j}(A),$$

where the last inequality follows from the Rayleigh quotient minimum property on the spaces V_j.

The $\geq$ case: For any sequence of subspaces $V_1 \subset \cdots \subset V_k$ with $\dim(V_j) = i_j$, we must find some $W \in X(V_1, \ldots, V_k)$ such that $\sum_{j=1}^k \lambda_{i_j}(A) \geq \mathrm{tr}_W(A)$.

We prove this by induction on n. The $n = 1$ case is trivially the Courant-Fischer theorem. Assume now $n \geq 2$.

If $i_1 \geq 2$, then we can apply induction. Let $E = \mathrm{span}(e_{i_1}, \ldots, e_n)$. We construct a partial flag within E from the intersection of E with $V_1, \ldots, V_k$. We begin by picking an $(i_k - (i_1 - 1))$-dimensional subspace $W'_k \subset E \cap V_k$, which exists by counting dimensions (it has codimension $i_1 - 1$ within V_k). Then we go down by one space, picking an $(i_{k-1} - (i_1 - 1))$-dimensional subspace $W'_{k-1} \subset W'_k \cap V_{k-1}$. This still exists. We repeat this to form a flag $W'_1 \subset \cdots \subset W'_k$. Now, since $\dim(E) \leq n - 1$, we can apply the induction hypothesis: there exists some $W \in X(W'_1, \ldots, W'_k)$ such that

$$\sum_{j=1}^k \lambda_{i_j - (i_1 - 1)}(A|_E) \geq \mathrm{tr}_W(A).$$

Note that $\lambda_{i_j-(i_1-1)}(A|_E)$ is the $(i_j - (i_1 - 1))$-th eigenvalue of A orthogonally projected down to E. By the Cauchy interlacing theorem, $\lambda_{i_j-(i_1-1)}(A|_E) \leq \lambda_{i_j}(A)$. Since $X(W'_1, \ldots, W'_k) \subset X(V_1, \ldots, V_k)$, we are done.

If $i_1 = 1$, we perform a similar construction. Let $E = \mathrm{span}(e_2, \ldots, e_n)$. If $V_k \subset E$, then we can induct immediately. Otherwise, we construct a partial flag sequence $W'_2 \subset \cdots \subset W'_k$ inside E. By induction, there exists some $W' \in X(W'_2, \ldots, W'_k) \subset X(V_2, \ldots, V_k)$ such that

$$\sum_{j=2}^k \lambda_{i_j - 1}(A|_E) \geq \mathrm{tr}_{W'}(A).$$

Thus, by Cauchy interlacing, $\sum_{j=2}^k \lambda_{i_j}(A) \geq \mathrm{tr}_{W'}(A)$. It remains to find some vector v such that $W' \oplus \mathrm{span}(v) \in X(V_1, \ldots, V_k)$. If $V_1 \not\subset W'$, then any $v \in V_1 \setminus W'$ would work. Otherwise, if $V_2 \not\subset W'$, then any $v \in V_2 \setminus W'$ would work, and so on. If none of these work, it implies $V_k \subset E$, which is a contradiction. This completes the induction. □

Corollary 8.15 (Weyl's Inequality)**.** *Let A and B be $n \times n$ Hermitian matrices with eigenvalues sorted in descending order: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. For all $1 \leq i, j \leq n$ such that $i + j - 1 \leq n$, we have:*

$$\lambda_{i+j-1}(A + B) \leq \lambda_i(A) + \lambda_j(B)$$

Theorem 8.16 (Lidskii's Inequality)**.** *Let A and B be $n \times n$ Hermitian matrices with their eigenvalues sorted in descending order, $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. For any sequence of indices $1 \leq i_1 < \cdots < i_k \leq n$, it holds that*

$$\sum_{j=1}^k \lambda_{i_j}(A + B) \leq \sum_{j=1}^k \lambda_{i_j}(A) + \sum_{j=1}^k \lambda_j(B)$$
$$\sum_{j=1}^k \lambda_{i_j}(A + B) \geq \sum_{j=1}^k \lambda_{i_j}(A) + \sum_{j=1}^k \lambda_{n-j+1}(B)$$

Corollary 8.17 (Extremal Partial Trace)**.** *Let A be a Hermitian matrix. We have:*

$$\sum_{j=1}^k \lambda_j(A) = \sup_{\dim(V)=k} \mathrm{tr}_V(A)$$
$$\sum_{j=1}^k \lambda_{n-j+1}(A) = \inf_{\dim(V)=k} \mathrm{tr}_V(A)$$

Corollary 8.18 (Schur-Horn Inequality)**.** *The sum of the largest k eigenvalues is a convex function, and the sum of the smallest k eigenvalues is a concave function. Furthermore, for any subset of indices $i_1 < \cdots < i_k$,*

$$\sum_{j=1}^k \lambda_{n-j+1}(A) \leq \sum_{j=1}^k A_{i_j, i_j} \leq \sum_{j=1}^k \lambda_j(A).$$

Corollary 8.19 (Hadamard's Determinant Inequality)**.** *For any positive definite matrix $A \succ 0$, $\det(A) \leq \prod_{i=1}^n A_{ii}$.*

Proof. The function $f(x) = -\log x$ is strictly convex on $(0, \infty)$. By Corollary 8.18, the diagonal of A is majorized by its eigenvalues $\lambda_1, \ldots, \lambda_n$. Applying Karamata's inequality for the convex function f, we have

$$\sum_{i=1}^n -\log(A_{ii}) \leq \sum_{i=1}^n -\log(\lambda_i).$$

Exponentiating both sides gives $\prod_{i=1}^n A_{ii} \geq \prod_{i=1}^n \lambda_i = \det(A)$. □

Theorem 8.20 (Schatten-norm Hölder Inequality)**.** *Given Hermitian matrices A, B and Hölder conjugate pairs $1/p + 1/q = 1$ with $p, q \geq 1$,*

$$|\operatorname{tr}(AB)| \leq \|A\|_{S_p} \|B\|_{S_q}$$

Proof. Without loss of generality, we can apply a unitary transformation so that B is diagonalized. We then need to show

$$\left| \sum_{i=1}^{n} B_{ii} A_{ii} \right| \leq \|A\|_{S_p} \|(B_{ii})\|_{\ell_q}.$$

By Hölder's inequality, it follows that

$$\left| \sum_{i=1}^{n} B_{ii} A_{ii} \right| \leq \|(A_{ii})\|_{\ell_p} \|(B_{ii})\|_{\ell_q} = \|(A_{ii})\|_{\ell_p} \|B\|_{S_q}.$$

Therefore, it suffices to show that $\|(A_{ii})\|_{\ell_p} \leq \|A\|_{S_p}$. By Lemma 8.18, the diagonal entries of A are majorized by the eigenspectrum of A. Because the norm function $f(x_1, \ldots, x_n) = \|x\|_p$ is symmetric and convex, it is Schur-convex by Lemma 2.14. Thus:

$$\|(A_{ii})\|_{\ell_p} \leq \|\lambda(A)\|_{\ell_p} = \|A\|_{S_p}.$$

□

8.11 The Operator Jensen Inequality

Definition 8.21 (Matrix Convex Combination)**.** *Let A_1 and A_2 be Hermitian matrices. Consider a decomposition of the identity of the form $K_1^* K_1 + K_2^* K_2 = I$. Then the Hermitian matrix $K_1^* A_1 K_1 + K_2^* A_2 K_2$ is called a matrix convex combination of A_1 and A_2.*

Theorem 8.22 (Operator Jensen Inequality)**.** *Let f be an operator convex function on an interval $I \subseteq \mathbb{R}$, and let A_1, A_2 be Hermitian matrices with eigenvalues in I. Consider a decomposition of the identity $K_1^* K_1 + K_2^* K_2 = I$. Then*

$$f(K_1^* A_1 K_1 + K_2^* A_2 K_2) \preceq K_1^* f(A_1) K_1 + K_2^* f(A_2) K_2.$$

Proof. Introduce the block-diagonal matrix $A = \begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix}$, such that $f(A) = \begin{pmatrix} f(A_1) & 0 \\ 0 & f(A_2) \end{pmatrix}$. We can construct a unitary matrix $Q = \begin{pmatrix} K_1 & L_1 \\ K_2 & L_2 \end{pmatrix}$ because its first block of columns is orthonormal: $K_1^* K_1 + K_2^* K_2 = I$. Direct computation shows that the matrix convex combination appears in the (1,1) block of the unitary conjugation:

$$Q^* A Q = \begin{pmatrix} K_1^* A_1 K_1 + K_2^* A_2 K_2 & * \\ * & * \end{pmatrix}$$

Let $[\cdot]_{11}$ denote the operation that extracts the (1,1) block of a matrix. We define a unitary $U = \begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix}$ such that for any block matrix, the diagonalizing operation $\frac{1}{2}(M + U^* M U)$ zeros out the off-diagonal blocks while preserving the (1,1) block. Utilizing the operator convexity of f and the fact that matrix functions commute with unitary conjugation, we calculate:

$$\begin{aligned}
f(K_1^* A_1 K_1 + K_2^* A_2 K_2) &= f([Q^* A Q]_{11}) \\
&= f([\tfrac{1}{2} Q^* A Q + \tfrac{1}{2} U^* (Q^* A Q) U]_{11}) \\
&= [f(\tfrac{1}{2} Q^* A Q + \tfrac{1}{2} (QU)^* A (QU))]_{11} \\
&\preceq [\tfrac{1}{2} f(Q^* A Q) + \tfrac{1}{2} f((QU)^* A (QU))]_{11} \\
&= [\tfrac{1}{2} Q^* f(A) Q + \tfrac{1}{2} U^* (Q^* f(A) Q) U]_{11} \\
&= [Q^* f(A) Q]_{11} \\
&= K_1^* f(A_1) K_1 + K_2^* f(A_2) K_2
\end{aligned}$$

□

8.12 Lieb's Concavity Theorem

Lemma 8.23 (Joint Convexity of Matrix Fractional Function)**.** *The map $(X, B) \mapsto X B^{-1} X^*$ is jointly convex for $B \succ 0$.*

Proof. Let $X = \frac{1}{2}(X_1+X_2)$ and $B = \frac{1}{2}(B_1+B_2)$. For $i \in \{1,2\}$, Lemma 8.1 implies $\begin{pmatrix} X_i B_i^{-1} X_i^* & X_i \\ X_i^* & B_i \end{pmatrix} \succeq 0$. Averaging these two positive block matrices yields:

$$\begin{pmatrix} \frac{1}{2}(X_1 B_1^{-1} X_1^* + X_2 B_2^{-1} X_2^*) & X \\ X^* & B \end{pmatrix} \succeq 0$$

Applying Lemma 8.1 to this average matrix gives $\frac{1}{2}(X_1 B_1^{-1} X_1^* + X_2 B_2^{-1} X_2^*) \succeq X B^{-1} X^*$. This establishes midpoint convexity, which implies full joint convexity by continuity. □

Lemma 8.24 (Joint Operator Concavity of Parallel Sum)**.** *The map $(A,B) \mapsto A : B := (A^{-1}+B^{-1})^{-1}$ is jointly operator concave for $A, B \succ 0$.*

Proof. The identity $A : B = A - A(A+B)^{-1}A$ follows from Lemma 8.2. By Lemma 8.23, the map $(A,B) \mapsto A(A+B)^{-1}A$ is jointly operator convex (proved by substituting $X = A$ and $B \leftarrow A+B$). Therefore, its negation is concave, making the parallel sum $A : B$ jointly operator concave. □

Lemma 8.25 (Joint Operator Concavity of Tensor Power)**.** $(A,B) \mapsto A^q \otimes B^{1-q}$ *is jointly operator concave for* $q \in (0,1)$.

Proof. We utilize the integral representation of the power $x^q = c_q \int_0^\infty t^q x(t+x)^{-1} dt$. Rewriting the integrand yields:

$$x(t+x)^{-1} = \left(\frac{t+x}{x}\right)^{-1} = \left(\left(\frac{x}{t}\right)^{-1} + 1\right)^{-1} = (x/t : 1)$$

For positive definite matrices, this gives the representation $A^q = c_q \int_0^\infty t^q (A/t : I) dt$. Applying this to the tensor product:

$$A^q \otimes B^{1-q} = (I \otimes B)(A \otimes B^{-1})^q = c_q \int_0^\infty t^q (I \otimes B) \left(\frac{A \otimes B^{-1}}{t} : I \otimes I\right) dt$$

For commuting matrices, the parallel sum satisfies the distributive property $Z(X : Y) = ZX : ZY$. Distributing $Z = I \otimes B$ into the parallel sum evaluates to:

$$A^q \otimes B^{1-q} = c_q \int_0^\infty t^q \left(\frac{A \otimes I}{t} : I \otimes B\right) dt$$

Since the parallel sum is jointly operator concave by Lemma 8.24, and integration with positive weights preserves concavity, the mapping $(A,B) \mapsto A^q \otimes B^{1-q}$ is jointly operator concave. □

Proposition 8.26 (Lieb's concavity theorem)**.** *For all $m \times n$ matrices K, and all q, r such that $0 \le q \le 1$ and $0 \le r \le 1$, with $q + r \le 1$, the function $F(A,B) \overset{def}{=} \operatorname{Tr}(K^T A^q K B^r)$ is jointly concave over (A,B), where A (resp. B) is over the set of all $m \times m$ (resp. $n \times n$) positive definite matrices.*

Proof. We can rewrite the trace objective using the vectorization operator $\operatorname{vec}(\cdot)$, which stacks the columns of a matrix into a single column vector. Utilizing the identity $\operatorname{vec}(XYZ) = (Z^T \otimes X)\operatorname{vec}(Y)$, we have:

$$\operatorname{vec}(A^q K B^r) = ((B^r)^T \otimes A^q)\operatorname{vec}(K) = ((B^T)^r \otimes A^q)\operatorname{vec}(K)$$

The trace of a product $K^T M$ corresponds to $\operatorname{vec}(K)^T \operatorname{vec}(M)$. Therefore, we can express the function $F(A,B)$ as a quadratic form over the Kronecker product:

$$F(A,B) = \operatorname{Tr}(K^T A^q K B^r) = \operatorname{vec}(K)^T \left((B^T)^r \otimes A^q\right) \operatorname{vec}(K)$$

First, consider the case where $q + r = 1$. By Lemma 8.25, the mapping $(A, B^T) \mapsto A^q \otimes (B^T)^{1-q}$ is jointly operator concave. Since the transpose map $B \mapsto B^T$ is linear and preserves positive definiteness, $(A,B) \mapsto (B^T)^{1-q} \otimes A^q$ is jointly operator concave. Applying $M \mapsto \operatorname{vec}(K)^T M \operatorname{vec}(K)$ preserves this concavity, proving $F(A,B)$ is jointly concave over (A,B).

Now consider the general case where $q + r < 1$. Let $s = q + r \in (0,1)$. We can factor the tensor product as:

$$(B^T)^r \otimes A^q = \left((B^T)^{r/s} \otimes A^{q/s}\right)^s$$

Because $\frac{q}{s} + \frac{r}{s} = 1$, the inner mapping $(A,B) \mapsto (B^T)^{r/s} \otimes A^{q/s}$ is jointly operator concave by the case established above. Furthermore, setting $B = I$ in Lemma 8.25 shows that the fractional power map $X \mapsto X^s$ is operator concave, and the integral representation shows it is operator monotone. Because the composition of an operator monotone concave function with a jointly operator concave function remains jointly operator concave, the composite map $(A,B) \mapsto \left((B^T)^{r/s} \otimes A^{q/s}\right)^s$ is jointly operator concave. Thus, $F(A,B)$ is jointly concave for all $q + r \le 1$. □

Lemma 8.27 (Joint Convexity of Matrix Relative Entropy)**.** $S(A||B) = \operatorname{Tr}(A \log A - A \log B - A + B)$ *is jointly convex for $A, B \succ 0$.*

Proof. Define the function $I_t(A,B) = \mathrm{Tr}(B^t A^{1-t} - A)$ for $t \in (0,1)$. By Proposition 8.26 (with $K = I$, $q = 1-t, r = t$), the map $(A,B) \mapsto \mathrm{Tr}(B^t A^{1-t})$ is jointly concave, making $I_t(A,B)$ jointly concave. We take the directional derivative as $t \to 0^+$:

$$\lim_{t\to 0^+} \frac{I_t(A,B)}{t} = \frac{d}{dt}\,\mathrm{Tr}(B^t A^{1-t} - A)\Big|_{t=0} = \mathrm{Tr}(A\log B - A\log A)$$

A pointwise limit of concave functions remains concave, so $\mathrm{Tr}(A\log B - A\log A)$ is jointly concave. Its negation, $S(A||B) - \mathrm{Tr}(B-A)$, is therefore convex. Adding the linear term $\mathrm{Tr}(B-A)$ proves the joint convexity of $S(A||B)$. □

Lemma 8.28 (Variational Formula for Trace)**.** *For $M \succ 0$, $\mathrm{Tr}(M) = \max_{T\succ 0} \mathrm{Tr}(T\log M - T\log T + T)$.*

Proof. The tangent line to the strictly convex function $f(x) = x\log x - x$ at $x = h$ yields the scalar inequality $a\log a - a\log h - a + h \geq 0$ for all $a, h > 0$. By Lemma 8.12, we obtain $S(T||M) = \mathrm{Tr}(T\log T - T\log M - T + M) \geq 0$ for positive definite matrices T, M. Rearranging this inequality yields $\mathrm{Tr}(M) \geq \mathrm{Tr}(T\log M - T\log T + T)$. Equality is explicitly achieved when $T = M$. □

Theorem 8.29 (Lieb)**.** *Let H be a fixed Hermitian matrix with dimension d. The map*

$$A \longmapsto \mathrm{tr}\exp(H + \log A)$$

is concave on the convex cone of $d \times d$ positive-definite Hermitian matrices.

Proof. Substitute $M = \exp(H + \log A)$ directly into the variational formula from Lemma 8.28:

$$\mathrm{Tr}\exp(H+\log A) = \max_{T\succ 0} \mathrm{Tr}\left(T(H+\log A) - T\log T + T\right) = \max_{T\succ 0}\left\{\mathrm{Tr}(TH) - S(T||A) + \mathrm{Tr}(A)\right\}$$

By Lemma 8.27, $S(T||A)$ is jointly convex in the variables (T, A). Consequently, the target objective function inside the supremum, $\mathrm{Tr}(TH) - S(T||A) + \mathrm{Tr}(A)$, is jointly concave in (T, A). Because the partial supremum of a jointly concave function over a convex set inherently remains concave, the resultant map $A \mapsto \max_{T\succ 0}\{\dots\}$ is strictly concave on the convex cone of positive-definite matrices. □

8.13 Lieb-Thirring Inequality

Theorem 8.30 (Lieb-Thirring Inequality)**.** *For positive definite matrices $A, B \succ 0$ and $q \geq 1$,*

$$\mathrm{Tr}\left((BAB)^q\right) \leq \mathrm{Tr}(B^q A^q B^q).$$

Proof. **Case 1:** $q = 2^m$
Let $U = B$ and $V = A^{1/2}$. By Lemma 8.9:

$$\mathrm{Tr}\left((B^2A)^q\right) \leq \mathrm{Tr}(B^{2q}A^q)$$

Using the cyclic property of the trace:

$$\mathrm{Tr}\left((B^2A)^q\right) = \mathrm{Tr}\left((BAB)^q\right).$$

On the right side, $\mathrm{Tr}(B^{2q}A^q) = \mathrm{Tr}(B^qB^qA^q) = \mathrm{Tr}(B^qA^qB^q)$. Therefore:

$$\mathrm{Tr}\left((BAB)^q\right) \leq \mathrm{Tr}(B^qA^qB^q),$$

Case 2: All real $q \geq 1$.
For any $d \times d$ positive definite matrix X and integer $1 \leq k \leq d$, let $\bigwedge^k X$ denote its k-th exterior power. The exterior power preserves multiplicativity, meaning $\bigwedge^k(XY) = (\bigwedge^k X)(\bigwedge^k Y)$ and $\bigwedge^k(X^q) = (\bigwedge^k X)^q$. The largest eigenvalue (the spectral norm) of $\bigwedge^k X$ is the product of the k largest eigenvalues of X:

$$\left\|\bigwedge^k X\right\|_\infty = \lambda_1\left(\bigwedge^k X\right) = \prod_{i=1}^k \lambda_i(X)$$

where $\lambda_i(X)$ are sorted in descending order $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d > 0$.

Taking the limit $m \to \infty$ of Case 1, we have:

$$\left\|(BAB)^q\right\|_\infty \leq \left\|B^qA^qB^q\right\|_\infty$$

By substituting the matrices A and B with their k-th exterior powers $\bigwedge^k A$ and $\bigwedge^k B$ into this norm inequality, and applying the multiplicativity property, we obtain:

$$\left\|\bigwedge^k (BAB)^q\right\|_\infty \leq \left\|\bigwedge^k B^qA^qB^q\right\|_\infty$$

This implies that for every $1 \le k \le d$:

$$\prod_{i=1}^{k} \lambda_i\big((BAB)^q\big) \le \prod_{i=1}^{k} \lambda_i\big(B^q A^q B^q\big)$$

Weak log-majorization implies weak majorization by Theorem 2.15, therefore:

$$\sum_{i=1}^{k} \lambda_i\big((BAB)^q\big) \le \sum_{i=1}^{k} \lambda_i\big(B^q A^q B^q\big) \quad \text{for all } 1 \le k \le d.$$

We evaluate this at $k = d$. Thus:

$$\operatorname{Tr}\big((BAB)^q\big) \le \operatorname{Tr}(B^q A^q B^q).$$

□

Theorem 8.31 (Extended Lieb-Thirring Inequality)**.** *For $A \succ 0$, $B \succeq 0$, and $\alpha \in [0,1]$,*

$$\operatorname{Tr}(BA^\alpha BA^{1-\alpha}) \le \operatorname{Tr}(B^2 A).$$

Proof. Define the function $f(\alpha) = \operatorname{Tr}(BA^\alpha BA^{1-\alpha})$. By the cyclicity of the trace, $f(\alpha) = \operatorname{Tr}(A^{1-\alpha} BA^\alpha B) = f(1-\alpha)$, so the function f is symmetric around $\alpha = 1/2$. Our goal is to show that for any $\alpha \in [0, 1/2]$, the function satisfies:

$$f(0) + f(2\alpha) \ge 2f(\alpha). \tag{31}$$

If inequality (31) holds, it implies that the maximum of f on the interval $[0,1]$ must be achieved at the endpoints $\alpha \in \{0,1\}$. To see this, let α be the maximizer, if it was an interior point then we would achieve a larger value at 2α.

It remains to prove (31). First, let

$$M_1 = \begin{pmatrix} B & -B^{1/2}A^\alpha B^{1/2} \\ -B^{1/2}A^\alpha B^{1/2} & B^{1/2}A^{2\alpha}B^{1/2} \end{pmatrix}.$$

To verify that $M_1 \succeq 0$, we take its quadratic form with respect to an arbitrary block vector $w = \begin{pmatrix} u \\ v \end{pmatrix}$. Let $\tilde{u} = B^{1/2}u$ and $\tilde{v} = B^{1/2}v$. The quadratic form expands to:

$$w^* M_1 w = \tilde{u}^*\tilde{u} - \tilde{u}^* A^\alpha \tilde{v} - \tilde{v}^* A^\alpha \tilde{u} + \tilde{v}^* A^{2\alpha} \tilde{v}.$$

Since $A \succ 0$, we can diagonalize it as $A = \sum_i \lambda_i e_i e_i^*$. Letting $\tilde{u}_i = e_i^*\tilde{u}$ and $\tilde{v}_i = e_i^*\tilde{v}$ be the components of the vectors in this eigenbasis, we can rewrite the form as a sum over the eigenspace:

$$w^* M_1 w = \sum_i \left(|\tilde{u}_i|^2 - \lambda_i^\alpha \tilde{u}_i^* \tilde{v}_i - \lambda_i^\alpha \tilde{v}_i^* \tilde{u}_i + \lambda_i^{2\alpha} |\tilde{v}_i|^2\right) = \sum_i |\tilde{u}_i - \lambda_i^\alpha \tilde{v}_i|^2 \ge 0.$$

Thus, $M_1 \succeq 0$. Similarly, we define a second block matrix:

$$M_2 = \begin{pmatrix} B^{1/2}AB^{1/2} & B^{1/2}A^{1-\alpha}B^{1/2} \\ B^{1/2}A^{1-\alpha}B^{1/2} & B^{1/2}A^{1-2\alpha}B^{1/2} \end{pmatrix}.$$

Using the same change of basis, the quadratic form for M_2 evaluates to:

$$\begin{aligned} w^* M_2 w &= \tilde{u}^* A \tilde{u} + \tilde{u}^* A^{1-\alpha} \tilde{v} + \tilde{v}^* A^{1-\alpha} \tilde{u} + \tilde{v}^* A^{1-2\alpha} \tilde{v} \\ &= \sum_i \left(\lambda_i |\tilde{u}_i|^2 + \lambda_i^{1-\alpha}(\tilde{u}_i^* \tilde{v}_i + \tilde{v}_i^* \tilde{u}_i) + \lambda_i^{1-2\alpha} |\tilde{v}_i|^2\right) \\ &= \sum_i \lambda_i^{1-2\alpha} \left(\lambda_i^{2\alpha} |\tilde{u}_i|^2 + \lambda_i^\alpha (\tilde{u}_i^* \tilde{v}_i + \tilde{v}_i^* \tilde{u}_i) + |\tilde{v}_i|^2\right) \\ &= \sum_i \lambda_i^{1-2\alpha} |\lambda_i^\alpha \tilde{u}_i + \tilde{v}_i|^2 \ge 0. \end{aligned}$$

Because $\lambda_i > 0$, the sum is non-negative, meaning $M_2 \succeq 0$.

Then we know $\operatorname{Tr}(M_1 M_2) \ge 0$. We calculate the trace:

$$\begin{aligned} \operatorname{Tr}([M_1 M_2]_{11}) &= \operatorname{Tr}(BB^{1/2}AB^{1/2}) - \operatorname{Tr}(B^{1/2}A^\alpha B^{1/2}B^{1/2}A^{1-\alpha}B^{1/2}) \\ \operatorname{Tr}([M_1 M_2]_{22}) &= -\operatorname{Tr}(B^{1/2}A^\alpha B^{1/2}B^{1/2}A^{1-\alpha}B^{1/2}) + \operatorname{Tr}(B^{1/2}A^{2\alpha}B^{1/2}B^{1/2}A^{1-2\alpha}B^{1/2}) \end{aligned}$$

Simplifying we get:

$$\operatorname{Tr}(M_1 M_2) = \operatorname{Tr}(B^2 A) - 2\operatorname{Tr}(BA^\alpha BA^{1-\alpha}) + \operatorname{Tr}(BA^{2\alpha}BA^{1-2\alpha}) \ge 0.$$

□