*Note*: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

# 1 Counting inversions

This problem arises in the analysis of *rankings*. Consider comparing two rankings. One way is to label the elements (books, movies, etc.) from 1 to $k$ according to one of the rankings, then order these labels according to the other ranking, and see how many pairs are "out of order".

We are given a sequence of $k$ distinct numbers $n_1, \cdots, n_k$. We say that two indices $i < j$ form an inversion if $n_i > n_j$, that is if the two elements $n_i$ and $n_j$ are "out of order". Provide a divide and conquer algorithm to determine the number of inversions in the sequence $n_1, \cdots, n_k$ in time $\mathcal{O}(k \log k)$. Only an algorithm description and runtime analysis is needed.

*Hint: Modify merge sort to count during merging.*

**Solution:**

(i) **Main idea**
There can be a quadratic number of inversions. So, our algorithm must determine the total count without looking at each inversion individually.
The idea is to modify merge sort. We split the sequence into two halves $n_1, \cdots, n_l$ and $n_{l+1}, \cdots, n_k$ and count number of inversions in each half while sorting the two halves separately. Then we count the number of inversions $(a_i, a_j)$, where two numbers belong to different halves, while combining the two halves. The total count is the sum of these three counts.

(ii) **Psuedocode**

```
procedure COUNT(A)
    if length[A] = 1 then
        return A, 0
    B, x ← COUNT(first half of A)
    C, y ← COUNT(rest of A)
    D ← empty list
    z ← 0
    while B is not empty or C is not empty do
        if B is empty then
            Append C to D and remove elements from C
        else if C is empty then
            Append B to D and remove elements from B
        else if B[1] < C[1] then
            Append B[1] to D and remove B[1] from B
        else
            Append C[1] to D and remove C[1] from C
            z ← z+ length[B]
    return D, x + y + z
```

(iii) **Proof of correctness** Consider now a step in merging. Suppose the pointers are pointing at elements $b_i$ and $c_j$. Because $B$ and $C$ are sorted , if $b_i$ is appended to $D$, no new inversions are encountered, since $b_i$ is smaller than everything left in list $C$, and it comes before all of them. On the other hand, if $c_j$ is appended to $D$, then it is smaller than all the remaining elements in $B$, and it comes after all of them, so we increase the count of inversions by the number of elements remaining in $B$.

(iv) **Running time analysis** In each recursive call, we merge the two sorted lists and count the inversions in $O(n)$. The running time is given by $T(n) = 2T(n/2) + O(n)$ which is $O(n \log n)$ by the master theorem.

# 2 Pareto Optimality

Given a set of points $P = \{(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)\}$, a point $(x_i, y_i) \in P$ is Pareto-optimal if there does not exist any $j \neq i$ such that such that $x_j > x_i$ and $y_j > y_i$. In other words, there is no point in $P$ above and to the right of $(x_i, y_i)$. Design a $O(n \log n)$-time divide-and-conquer algorithm that given $P$, outputs all Pareto-optimal points in $P$. Only an algorithm description and runtime analysis is needed.

*Hint: Split the array by $x$-coordinate. Show that all points returned by one of the two recursive calls is Pareto-optimal, and that you can get rid of all non-Pareto-optimal points in the other recursive call in linear time.*

**Solution:**

**Algorithm:** Let $L$ be the left half of the points when sorted by $x$-coordinate, and $R$ be the right half. Recurse on $L$ and $R$, let $L', R'$ be the sets of Pareto-optimal points returned. We can compute $y_{max}$, the largest $y$-coordinate in $R$, in a linear scan, and then remove all points in $L'$ with a smaller $y$-coordinate. We then return the union of $L', R'$.

**Proof:** We now prove the correctness of the algorithm on a sorted array of points. Note that in the simplest case, there is only one point, which is by default Pareto-optimal. Now, say that there are more than one points. Then, we can assume that $L'$ contains the points that are Pareto-optimal in the set $L$ and $R'$ contains the points that are Pareto-optimal in the set $R$. Every point in $R'$ is Pareto-optimal in $L \cup R$, since all points in $L$ have smaller $x$-coordinates and can't violate Pareto-optimality of points in $R'$. For each point in $L'$, it's Pareto-optimal in $L \cup R$ iff its $y$-coordinate is larger than $y_{max}$, the largest $y$-coordinate in $R$. Hence, our algorithm returns a set that is Pareto-optimal in the set $L \cup R$.

**Runtime:** Using the master theorem, we can see this runs in $T(n) = 2T(n/2) + O(n) = O(n \log n)$ time.

# 3 Monotone matrices

A $m$-by-$n$ matrix $A$ is *monotone* if $n \geq m$, each row of $A$ has no duplicate entries, and it has the following property: if the minimum of row $i$ is located at column $j_i$, then $j_1 < j_2 < j_3 \ldots j_m$. For example, the following matrix is monotone (the minimum of each row is bolded):

$$\begin{bmatrix} \mathbf{1} & 3 & 4 & 6 & 5 & 2 \\ 7 & 3 & \mathbf{2} & 5 & 6 & 4 \\ 7 & 9 & 6 & 3 & 10 & \mathbf{0} \end{bmatrix}$$

Give an efficient (i.e., better than $O(mn)$-time) algorithm that finds the minimum in each row of an $m$-by-$n$ monotone matrix $A$.

**Give a 3-part solution.** You do not need to write a formal recurrence relation in your runtime analysis; an informal summary of the runtime analysis such as "proof by picture" is fine.

**Solution:**

(i) **Main idea:** If $A$ has one row, we just scan that row and output its minimum.

Otherwise, we find the smallest entry of the $m/2$-th row of $A$ by just scanning the row. If this entry is located at column $j$, then since $A$ is a monotone matrix, the minimum for all rows above the $m/2$-th row must be located to the left of the $j$-th column. i.e. in the submatrix formed by rows 1 to $m/2-1$ and columns 1 to $j-1$ of $A$, which we will denote by $A[1 : m/2-1, 1 : j-1]$. Similarly, the minimum for all rows below the $m/2$-th row must be located to the right of the $j$-th column. So we can recursively call the algorithm on the submatrices $A[1 : m/2-1, 1 : j-1]$ and $A[m/2+1 : m, j+1 : n]$ to find and output the minima for rows 1 to $m/2-1$ and rows $m/2+1$ to $m$.

(ii) **Proof of correctness:** We will prove correctness by (total) induction on $m$.

As a base case, $m = 1$, and the algorithm explicitly finds and outputs the minimum of the single row.

If $A$ has more than one row, we of course find and output the correct row minimum for row $m/2$. As argued above, the minima of rows 1 to $m/2-1$ of $A$ are the same as the minima of the submatrix $A[1 : m/2-1, 1 : j-1]$, and the minima of rows $m/2+1$ to $m$ are the same as the minima of the submatrix $A[m/2+1 : m, j+1 : n]$. By the induction hypothesis, the algorithm correctly outputs the minima of these matrices, and together with the $m/2$ row above, they find and output the minima of all the rows of $A$.

(iii) **Running time analysis:** There are two ways to analyze the run time. One involves doing an explicit accounting of the total number of steps at each level of the recursion, as we did before we relied on the master theorem, or as we did in the proof of the master theorem. Since $m$ is halved at each step of recursion, there are $\log m$ levels of recursion. At each level of recursion, the number of columns of the matrix get split into two – those associated with the left matrix and those associated with the right matrix. Moreover, the number of steps required to perform the split is just $n$, since it involves scanning all the entries of a single row. This means that at any level of the recursion, all the submatrices have disjoint columns, meaning if the different submatrices have $n_k$ columns, then $\sum_k n_k \leq n$. The total number of steps required to split these matrices to go to the next level of recursion is then just $\sum_k n_k \leq n$. So there are $\log m$ levels of recursion, each taking total time $n$, for a grand total of $n \log m$.

Actually to be more accurate, when m=1, $\log m = 0$, so the expression should be $n(\log m + 1)$ to get the base case right.

For a “proof by picture”, consider the following picture, where the grey boxes represent the submatrices we’re solving the problem for at each level of recursion, and the blue lines represent the rows we’re scanning at each level of the recursion. We can see the total length of the blue lines in each level of the recursion is $O(n)$.

Another way to analyze the running time is by writing and solving a recurrence relation (though again, this isn’t necessary for full credit). The recurrence is easy to write out: let $T(m, n)$ be the number of steps to solve the problem for an $m \times n$ array. It takes $n$ time to find the minimum of row $m/2$. If this row has minimum at column $j$, we recurse on submatrices of size at most $m/2$-by-$j$ and $m/2$-by-$(n - j)$. So we can write the following recurrence relation: $T(m, n) \leq T(m/2, j) + T(m/2, n - j) + n$.

This does not directly look the recurrences in the master theorem — it is “2-D” since it depends upon two variables. You need some inspiration to guess the solution. We will guess that $T(m, n) \leq n(\log m + 1)$. We can prove this by strong induction on $m$.

Base case: $T(1, n) = n = n(\log 1 + 1)$.
Induction step:

$$T(m, n) \leq T(m/2, j) + T(m/2, n - j) + n \leq j \log(m/2) + 1 + (n - j) \log(m/2) + 1 + n$$

(by the induction hypothesis)

$$= n(\log(m/2) + 1 + 1) = n(\log m + 1).$$

# 4 FFT Intro

We will use $\omega_n$ to denote the first $n$-th root of unity $\omega_n = e^{2\pi i/n}$. The most important fact about roots of unity for our purposes is that the squares of the $2n$-th roots of unity *are* the $n$-th roots of unity.

**Fast Fourier Transform!** The *Fast Fourier Transform* $\text{FFT}(p, n)$ takes arguments $n$ (an integer power of 2), and $p$ (some vector $[p_0, p_1, \ldots, p_{n-1}]$).

Here, we describe how we can view FFT as a way to perform a specific matrix multiplication involving the DFT matrix. Note, however, that the FFT algorithm will not explicitly compute this matrix. We have written out the matrix below for convenience.

Treating $p$ as a polynomial $P(x) = p_0 + p_1 x + \ldots + p_{n-1}x^{n-1}$, the FFT computes the value of $P(x)$ for all $x$ that are $n$-th roots of unity by computing the result of the following matrix multiplication in $\mathcal{O}(n \log n)$ time:

$$\begin{bmatrix} P(1) \\ P(\omega_n) \\ P(\omega_n^2) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \cdots & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

If we let $E(x) = p_0 + p_2 x + \ldots p_{n-2}x^{n/2-1}$ and $O(x) = p_1 + p_3 x + \ldots p_{n-1}x^{n/2-1}$, then $P(x) = E(x^2) + xO(x^2)$, and then $FFT(p, n)$ can be expressed as a divide-and-conquer algorithm:

1. Compute $E' = \text{FFT}(E, n/2)$ and $O' = \text{FFT}(O, n/2)$.
2. For $i = 0 \ldots n-1$, assign $P(\omega_n^i) \leftarrow E((\omega_n^i)^2) + \omega_n^i O((\omega_n^i)^2)$

Also observe that:

$$\frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \cdots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)(n-1)} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \cdots & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix}^{-1}$$

(You should verify this on your own!) And so given the values $P(1), P(\omega_n), P(\omega_n^2)\ldots$, we can compute $P$ by finding the result of the following matrix multiplication in $O(n \log n)$ time:

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \cdots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} P(1) \\ P(\omega_n) \\ P(\omega_n^2) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix}$$

This can be done in $O(n \log n)$ time using a similar divide and conquer algorithm.

(a) Let $p = [p_0]$. What is $\text{FFT}(p, 1)$?

**Solution:** Notice the FFT matrix is just [1], so $\text{FFT}(p, 1) = [p_0]$.

(b) Use the FFT algorithm to compute $\text{FFT}([1, 4], 2)$ and $\text{FFT}([3, 2], 2)$.

**Solution:** $\text{FFT}([1, 4], 2) = [5, -3]$ and $\text{FFT}([3, 2], 2) = [5, 1]$.

We show how to compute $\text{FFT}([1, 4], 2)$, and $\text{FFT}([3, 2], 2)$ is similar.

First we compute $\text{FFT}([1], 1) = [1] = E'$ and $\text{FFT}([4], 1) = [4] = O'$ by part (a). Notice that $E' = [E(1)] = 1$ and $O' = [O(1)] = [4]$, so when we need to use these values later they have already been computed in $E'$ and $O'$.

Let $P$ be our result. We wish to compute $P(\omega_2^0) = P(1)$ and $P(\omega_2^1) = P(-1)$.

$$P(1) = E(1) + 1 \cdot O(1) = 1 + 4 = 5$$
$$P(-1) = E(1) + -1 \cdot O(1) = 1 - 4 = -3$$

So our answer is $[5, -3]$.

(c) Use your answers to the previous parts to compute $\text{FFT}([1, 3, 4, 2], 4)$.

**Solution:** $\omega_4 = i$. The following table is good to keep handy:

| | | | | |
|---|---|---|---|---|
| $\boldsymbol{\omega_4^i}$ | $1$ | $i$ | $-1$ | $-i$ |
| $\boldsymbol{(\omega_4^i)^2}$ | $1$ | $-1$ | $1$ | $-1$ |

Let $E' = \text{FFT}([1, 4], 2) = [5, -3]$ and $O' = \text{FFT}([3, 2], 2) = [5, 1]$. Notice that $E' = [E(1), E(-1)] = [5, -3] =$ and $O' = [O(1), O(-1)] = [5, 1]$, so when we need to use these values later they have already been computed in the divide step. Let $R$ be our result, we wish to compute $R(1), R(i), R(-1), R(-i)$.

$$R(1) = E(1) + 1 \cdot O(1) = 5 + 5 = 10$$
$$R(i) = E(-1) + i \cdot O(-1) = -3 + i$$
$$R(-1) = E(1) - 1 \cdot O(1) = 5 - 5 = 0$$
$$R(-i) = E(-1) - i \cdot O(-1) = -3 - i$$

So our answer $[10, -3 + i, 0, -3 - i]$.

(d) Describe how to multiply two polynomials $P(x), Q(x)$ in coefficient form of degree at most $d$.

**Solution:** The idea is to take the FFT of both $P$ and $Q$, multiply the evaluations, and then take the inverse FFT. Note that $P \cdot Q$ has degree at most $2d$, which means we need to pick $n$ as the smallest power of 2 greater than $2d$, call this $2^k$. We can zero-pad both polynomials so they have degree $2^k - 1$.

Then $R = \mathrm{FFT}(P, 2^k) \cdot \mathrm{FFT}(Q, 2^k)$ (with multiplication elementwise) computes $PQ(\omega_{2^k}^i)$ for all $i = 0, \ldots, 2^k - 1$.

We take the inverse FFT of $R$ to get back to $P \cdot Q$ in coefficient form.

(e) Use the algorithm from the previous part to multiply the two polynomials $P(x) = 1 + 2x$ and $Q(x) = 3 - x$ in coefficient form.

**Solution:**

**Standard FFT:**

First we need to compute $FFT([1, 2, 0, 0], 4)$. To do this, we need to compute $FFT([1, 0], 2) = [E(1)+1\cdot O(1), E(1)-1\cdot O(1)] = [1, 1]$ and $FFT([2, 0], 2) = [E(1)+1\cdot O(1), E(1)-1\cdot O(1)] = [2, 2]$. So, now we can compute $FFT([1, 2, 0, 0], 4) = [FFT([1, 0], 2)+(1, i)\cdot FFT([2, 0], 2), FFT([1, 0], 2)-(1, i)\cdot FFT([2, 0], 2)] = [3, 1 + 2i, -1, 1 - 2i]$

Now we need to compute $FFT([3, -1, 0, 0], 4)$. With a similar approach we get $[2, 3-i, 4, 3+i]$. Now we can multiply these to get the pointwise representation of the product as $[2\cdot 3, (1 + 2i)(3 - i), -1\cdot 4, (1 - 2i)(3 + i)] = [6, 5 + 5i, -4, 5 - 5i]$.

Now we can use the inverse FFT algorithm. We need to compute $iFFT([6, 5+5i, -4, 5-5i], 4)$. To do this, we need to compute $iFFT([6, -4], 2) = [E(1) + 1\cdot O(1), E(1) - 1\cdot O(1)] = [2, 10]$ and $iFFT([5 + 5i, 5 - 5i], 2) = [E(1) + 1\cdot O(1), E(1) - 1\cdot O(1)] = [10, 10i]$. So, now we can compute $iFFT([6, 5 + 5i, -4, 5 - 5i], 4) = [FFT([6, -4], 2) + (1, -i)\cdot FFT([5 + 5i, 5 - 5i], 2), FFT([6, -4], 2) - (1, -i)\cdot FFT([5 + 5i, 5 - 5i], 2)] = [12, 20, -8, 0]$. And finally we need to multiply by $\frac{1}{4}$ to get the final product polynomial representation as $[3, 5, -2, 0]$ or $3+5x-2x^2$

**FFT via Matrix Multiplication**

We first convert $p$ and $q$ to their vector representations, making sure to pad with 0s so that they are $2^2 - 1 = 3$ degree polynomials:

$$p = [1, 2, 0, 0]^\top$$
$$q = [3, -1, 0, 0]^\top$$

Then, using the 4th root of unity ($w_4 = e^{i\frac{2\pi}{4}} = e^{i\frac{\pi}{2}} = i$), we can compute the FFT matrix:

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w_4 & w_4^2 & w_4^3 \\ 1 & w_4^2 & w_4^4 & w_4^6 \\ 1 & w_4^3 & w_4^6 & w_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Then, we can compute the values of $p$ and $q$ when evaluated at the 4th roots of unity as follows:

$$P = \begin{bmatrix} P(1) \\ P(w_4) \\ P(w_4^2) \\ P(w_4^3) \end{bmatrix} = Mp = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 1+2i \\ -1 \\ 1-2i \end{bmatrix}$$

$$Q = \begin{bmatrix} Q(1) \\ Q(w_4) \\ Q(w_4^2) \\ Q(w_4^3) \end{bmatrix} = Mq = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 3 \\ -1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 3-i \\ 4 \\ 3+i \end{bmatrix}$$

Then, we can compute the product $R(x) = P(x)Q(x)$ evaluated at each of the 4th roots of unity by obtaining the element-wise product of evaluations $P$ and $Q$:

$$R = \begin{bmatrix} R(1) \\ R(w_4) \\ R(w_4^2) \\ R(w_4^3) \end{bmatrix} = \begin{bmatrix} P(1) \cdot Q(1) \\ P(w_4) \cdot Q(w_4) \\ P(w_4^2) \cdot Q(w_4^2) \\ P(w_4^3) \cdot Q(w_3^2) \end{bmatrix} = \begin{bmatrix} 3 \cdot 2 \\ (1+2i) \cdot (3-i) \\ 4 \cdot (-4) \\ (1-2i) \cdot (3+i) \end{bmatrix} = \begin{bmatrix} 6 \\ 5+5i \\ -4 \\ 5-5i \end{bmatrix}$$

Finally, to convert $R$ back to coefficient form, we multiply what we just computed by the inverse FFT matrix:

$$r = M^{-1}R = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} 6 \\ 5+5i \\ -4 \\ 5-5i \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ -2 \\ 0 \end{bmatrix}$$

Which we is the coefficient representation for $R(x) = 3 + 5x - 2x^2$.