

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, \dots, 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, \dots, 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, \dots, n_l and n_{l+1}, \dots, 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  $A, 0$ 
   $B, x \leftarrow$  COUNT(first half of  $A$ )
   $C, y \leftarrow$  COUNT(rest of  $A$ )
   $D \leftarrow$  empty list
   $z \leftarrow 0$ 
  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  $C$ 
       $z \leftarrow z + \text{length}[B]$ 
  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 Maximum Subarray Sum

Given an array A of n integers, the maximum subarray sum is the largest sum of any contiguous subarray of A (including the empty subarray). In other words, the maximum subarray sum is:

$$\max_{i \leq j} \sum_{k=i}^j A[k]$$

For example, the maximum subarray sum of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ is 6, the sum of the contiguous subarray $[4, -1, 2, 1]$.

Design an $O(n \log n)$ -time algorithm that finds the maximum subarray sum.

Give a 3-part solution.

Hint 1: Split the array into two equally-sized pieces. What are the possibilities for the subarray, and how does this apply if we want to use divide and conquer?

Solution:

- (i) **Main idea:** At first glance it seems like we have to find two indices i and j such that the subarray $A[i : j]$ has sum as large as possible. There are n^2 possibilities. However, there is a divide and conquer solution: split A into two equal pieces, L and R , defined as $A[1 : n/2]$ and $A[n/2 + 1 : n]$. There are two options for the subarray with the largest sum:

- (a) It is contained entirely in L or R : Solve by recursion. Let s_1 be the result of recursively calling the algorithm on L , and s_2 be the result of recursively calling the algorithm on R .
- (b) It “crosses the boundary”, i.e. starts in L and ends in R : if the maximum subarray runs from $A[i : j]$, then $A[i, n/2]$ must be the maximum subarray ending at $A[n/2]$, and $A[n/2 + 1, j]$ must be the maximum subarray starting at $A[n/2 + 1]$. It might seem like we must now solve two largest subarray sum problems: find i such that the subarray sum $A[i, n/2]$ is as large as possible. And same for $A[n/2 + 1, j]$. Note that each of these require finding just one index i or j for which there are only $n/2$ possibilities each. So each can be computed in $O(n)$ steps. e.g. to find i , compute each successive subarray sum $A[k, n/2]$ for $k = 1$ to $n/2$ and take the maximum. Finally, let s_3 be the sum of the elements in $A[i, n/2]$ plus the sum of the elements in $A[n/2 + 1, j]$.

If $n = 1$, return $\max\{A[1], 0\}$. Else return $\max\{s_1, s_2, s_3\}$.

- (ii) **Proof of correctness:** We prove by induction. When the array is length 1, our algorithm is clearly correct.

When the array is length at least 2, there are two options for the subarray with the largest sum:

- (a) It is contained entirely in L or R .
- (b) It “crosses the boundary”, starts before element $n/2$ and ends after element $n/2$.

In the first case, either s_1 or s_2 is outputted by the algorithm, which we inductively assume is correct.

To handle the second case, the maximum subarray sum crossing the boundary must consist of the subarray with the largest sum ending in element $n/2$, and the subarray with the largest sum beginning with element $n/2 + 1$. So s_3 will be outputted by the algorithm in this case, which is correct.

- (iii) **Runtime analysis:** Our method for computing s_3 takes $O(n)$ time, since each of the sums of $A[i : n/2]$ and $A[n/2 + 1 : i]$ is computed in $O(1)$ time given a previous sum. So we get the recurrence:

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

Using the Master theorem, this results in a runtime of $\mathcal{O}(n \log n)$.

3 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 $x_j > x_i$ 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.

4 Complex numbers review

A *complex number* is a number that can be written in the rectangular form $a + bi$ (i is the imaginary unit, with $i^2 = -1$). The following famous equation (*Euler's formula*) relates the polar form of complex numbers to the rectangular form:

$$re^{i\theta} = r(\cos \theta + i \sin \theta)$$

In polar form, $r \geq 0$ represents the distance of the complex number from 0, and θ represents its angle. Note that since $\sin(\theta) = \sin(\theta + 2\pi)$, $\cos(\theta) = \cos(\theta + 2\pi)$, we have $re^{i\theta} = re^{i(\theta+2\pi)}$ for any r, θ .

The n -th *roots of unity* are the n complex numbers satisfying $\omega^n = 1$. They are given by

$$\omega_k = e^{2\pi i k/n}, \quad k = 0, 1, 2, \dots, n-1$$

- (a) Let $x = e^{2\pi i 3/10}, y = e^{2\pi i 5/10}$ which are two 10-th roots of unity. Compute the product $x \cdot y$. Is this an n -th root of unity for some n ? Is it a 10-th root of unity?

What happens if $x = e^{2\pi i 6/10}, y = e^{2\pi i 7/10}$?

For all your answers, simplify if possible.

Solution: $x \cdot y = e^{2\pi i 8/10}$. This is always an 10-th root of unity (it is in general). But because $8/10 = 4/5$, this is also a 5th root of unity.

If $x = e^{2\pi i 6/10}, y = e^{2\pi i 7/10}$, then we ‘wind around’ and the product becomes $e^{2\pi i 13/10} = e^{2\pi i 3/10}$.

- (b) Show that for any n -th root of unity $\omega \neq 1$, $\sum_{k=0}^{n-1} \omega^k = 0$, when $n > 1$.

Hint: Use the formula for the sum of a geometric series $\sum_{k=0}^n \alpha^k = \frac{\alpha^{n+1}-1}{\alpha-1}$. It works for complex numbers too!

Solution: Remember that $\omega^n = 1$. So

$$\sum_{k=0}^{n-1} \omega^k = \frac{\omega^n - 1}{\omega - 1} = \frac{1 - 1}{\omega - 1} = 0$$

- (c) (i) Find all ω such that $\omega^2 = -1$.

Solution: $\omega = i, -i$

There are many ways to arrive at the solution, here's one: Squaring both sides, we get $\omega^4 = 1$. So we only need to consider the 4th roots of unity, $e^{2\pi i \cdot 0/4}, e^{2\pi i \cdot 1/4}, e^{2\pi i \cdot 2/4}, e^{2\pi i \cdot 3/4}$, or equivalently $1, i, -1, -i$. Geometrically, we get these by going 0, 1, 2, 3 quarters of the way around the complex unit circle. Of these four values, the ones that square to -1 are $i, -i$.

- (ii) Find all ω such that $\omega^4 = -1$.

Solution: $\omega = e^{2\pi i \cdot 1/8}, e^{2\pi i \cdot 3/8}, e^{2\pi i \cdot 5/8}, e^{2\pi i \cdot 7/8}$

Similarly to the previous part, squaring both sides we get $\omega^8 = 1$, so we only need to consider the 8th roots of unity. However, $\omega^4 \neq 1$, so ω is not a 4th root of unity. The 8th roots of unity that are not 4th roots of unity are $e^{2\pi i \cdot 1/8}, e^{2\pi i \cdot 3/8}, e^{2\pi i \cdot 5/8}, e^{2\pi i \cdot 7/8}$, and we can check that these all are solutions to $\omega^4 = -1$.

5 (OPTIONAL) 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 $j_1 < j_2 < j_3 \dots j_m$. 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)

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