CS 170 HW 2

Due on 2018-09-09, at 11:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★★) 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 n 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 n distinct numbers $a_1, \cdots, a_n$. We say that two indices $i < j$ form an inversion if $a_i > a_j$, that is if the two elements a_i and a_j are "out of order". Provide a divide and conquer algorithm to determine the number of inversions in the sequence $a_1, \cdots, a_n$ in time $O(n \log n)$ (Four part solution required. *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 $a_1, \cdots, a_m$ and $a_{m+1}, \cdots, a_n$ 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 and 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
$z \leftarrow z+$ 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.

3 (★★★) Two sorted arrays

You are given two sorted arrays, each of size n. Give as efficient an algorithm as possible to find the k-th smallest element in the union of the two arrays. What is the running time of your algorithm as a function of k and n? (Four part solution required.)

Solution:

Main idea

We will assume $k \leq n$. If this is not the case, then for $k' = 2n + 1 - k$, we are trying to find the k'-th largest element, and $k' \leq n$. But finding the k-th largest element and finding the k-th smallest element are symmetric problems, so with some minor tweaks we can use the same algorithm. So for brevity, we will only give the answer for when $k \leq n$.

Since $k \leq n$, the kth smallest element can't exist past index k of either array. So we cut off all but the first k elements of both arrays. Now, we just want to find the median of the union of the two arrays. To do this, we check the middle elements of both arrays, and compare them. If, the median of the first list is smaller than the median of the second list, then the median can't be in the left half of the first list or the right half of the second list. If the median of the second list is smaller, the opposite is true. So we can cut these elements off, and then recurse on the resulting arrays.

Pseudocode

procedure TwoArraySelection($a[1..n]$, $b[1..n]$, element rank k)
Set $a := a[1, \cdots, k]$, $b := b[1, \cdots, k]$,
Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$
while $a[\ell_1] \neq b[\ell_2]$ and $k > 1$ **do**
if $a[\ell_1] > b[\ell_2]$ **then**
Set $a := a[1, \cdots, \ell_1]$; $b := b[\ell_2 + 1, \cdots, k]$; k := ℓ_1
Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$
else
Set $a := a[\ell_1 + 1, \cdots, k]$; $b := b[1, \cdots, \ell_2]$; $k := \ell_2$
Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$

if $k = 1$ **then**
    return $\min(a[1], b[1])$
else
    return $a[\ell_1]$

Proof of correctness

Our algorithm starts off by comparing elements $a[\ell_1]$ and $b[\ell_2]$. Suppose $a[\ell_1] > b[\ell_2]$. Then, in the union of a and b there can be at most $k-2$ elements smaller than $b[\ell_2]$, i.e. $a[1, \cdots, \ell_1 - 1]$ and $b[1, \cdots, \ell_2 - 1]$, and we must necessarily have $s_k > b[\ell_2]$. Similarly, all elements $a[1, \cdots, \ell_1]$ and $b[1, \cdots, \ell_2]$ will be smaller than $a[\ell_1 + 1]$; but these are k elements, so we must have $s_k < a[\ell_1 + 1]$.
This shows that s_k must be contained in the union of the subarrays $a[1, \cdots, \ell_1]$ and $b[\ell_2 + 1, \cdots, k]$. In particular, because we discarded ℓ_2 elements smaller than s_k, s_k will be the ℓ_1th smallest element in this union.
We can then find s_k by recursing on this smaller problem. The case for $a[\ell_1] < b[\ell_2]$ is symmetric.
If we reach $k = 1$ before $a[\ell_1] = b[\ell_2]$, we can cut the recursion a little short and just return the minimum element between a and b. You can make the algorithm work without this check, but it might get clunkier to think about the base cases.

Alternatively, if we reach a point where $a[\ell_1] = b[\ell_2]$, then there are exactly k greater elements, so we have $s_k = a[\ell_1] = b[\ell_2]$.

Running time analysis

At every step we halve the number of elements we consider, so the algorithm will terminate in $\log(k)$ recursive calls. Assuming the comparison takes constant time, the algorithm runs in time $\Theta(\log k)$. Note that this does not depend on n.

4 (★★★★) Majority Elements

An array $A[1 \ldots n]$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be **no** comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) The elements are also not hashable, i.e., you are *not* allowed to use any form of sets or maps with constant time insertion and lookups. However you *can* answer questions of the form: "is $A[i] = A[j]$?" in constant time. Four part solutions are required for each part below.

(a) Show how to solve this problem in $O(n \log n)$ time. (Hint: Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm? (Your algorithm for this part would technically meet the runtime requirements for part a, but you should *not* reuse this algorithm to answer part a)

Solution:

(a) **Main idea** Divide and conquer. At each step, partition into two arrays of equal size. Notice that if A has a majority element v, v must also be a majority element of A_1 or A_2 or both.

Psuedocode Note: The recursive calls in this algorithm to smaller arrays do not require duplicating the subarray. Instead, appropriate use of indices will suffice. We express the solution here without indices for legibility.

Let array A containing n values be the input to the algorithm.

`Majority`(A):

(a) If $n = 1$, then return $A[1]$.

(b) Otherwise, let H_1, H_2 be the two halves of the array.[1]

(c) Let $v_i =$ `Majority`(H_i) for $i = 1, 2$.

(d) In a linear sweep over A, count the number of elements of A equalling v_i. Call this c_i.

(e) Return v_i for any i such that $c_i > n/2$. Otherwise, return $\perp$.

Proof of correctness By strong induction:

Induction hypothesis The algorithm is correct for $k \leq n - 1$.

Base Case If $n = 1$, the array contains exactly one element, and we always return it.

Induction step: Suppose some element v appears more than $n/2$ times in the array. Then after we partition into sub-arrays of sizes n_1, n_2, with $n_1 + n_2 = n$, either the first array contains more than $n_1/2$ copies of v, or the second array contains more than $n_2/2$ copies of v. Either way, v is a majority element of at least one of the sub-arrays, and by our induction hypothesis will be returned by the recursive call to the algorithm.

Running time analysis Two calls to problems of size $n/2$, and then linear time to test $v1, v2$: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

(b) **Main idea** Removing two different items does not change the majority element (but may create spurious solutions!). We recursively pair up the elements and remove all pairs with different elements. For every pair with identical elements, it suffices to keep just one (since we do so for all pairs, we again don't change the majority). If n is odd, brute-force test in linear time whether the unpaired element is a majority element.

Psuedocode

Let array A containing n values be the input to the algorithm.

$\widetilde{\texttt{Majority}}(A)$: If n is odd, check if the first element is the majority element in a linear sweep over the array. If it is, return it. If not, throw it away. Instead if n is even, in a linear sweep over the array A, compare elements in pairs. If the pair of elements are same, keep one of them. If they are different, throw them both away.

[1] For $n = 2k + 1$, break the array into pieces of size k and $k + 1$.

Recurse on the reduced array until either one element remains or the array is empty (in which case, return $\perp$, i.e. no majority element exists). If the recursive call returns $\perp$, return $\perp$. Otherwise, check if the element returned by the recursive call is the majority element in a linear sweep over the array. If it is, return it, if not, return $\perp$.

Proof of correctness By strong induction:

Induction hypothesis The algorithm is correct for $k \leq n-1$.

Base Case If $n = 1$, the array contains exactly one element, and we always return it.

Induction step If n is odd, we test whether the first element is the majority element. If it is not, discarding it will certainly preserve the real majority element (if it exists).

Consider now when n is even. We can seperate the process into two steps for the analysis. First, consider all pairs consisting of two distinct elements. Discarding all these pairs preserves the majority element v. This is because if we discarded m pairs, we could have discarded at most m copies of v. So we are left with at least $\frac{n}{2} - m$ copies of v among $n-2m$ elements, and therefore v is still a majority element.

Now, the only pairs that remain have matching elements. So we can choose one from each pair, we leave the proportion of every element u unchanged. Therefore, this transformation preserves the majority element but leaves us with an array of size $\frac{n}{2} - m < n$. The correctness of the recursive call to the algorithm now follows from the induction hypothesis.

(Note that deleting elements might mean there is a majority element in the array passed to the recursive call when none existed originally, for example the array [1, 2, 1, 2, 3, 3] will be trimmed down to [3] before recursing. However, the final check in the algorithm ensures that we never return an element that is not the majority element).

Running time analysis For any n, in two recursions of the algorithm, the problem size reduces to $\leq n/2$. The overhead is $O(n)$. This follows the recursion formula,

$$T(n) = T(\frac{n}{2}) + O(n) = O(n).$$

5 (★★★★★) Merged Median

Given k sorted arrays of length l, design a *deterministic* algorithm (i.e. an algorithm that uses *no* randomness) to find the median element of all the $n = kl$ elements. Your algorithm should run asymptotically faster than $O(n)$.

(You need to give a four-part solution for this problem. You may assume that k, l are both large and roughly the same size, as when e.g. $l = 1$ you can't do better than $O(n)$ time)

Solution:

We provide two solutions. The first reduces the problem size by a larger amount in one iteration and thus requires fewer iterations, but does more work per iteration. The second reduces the problem size by a small amount in one iteration, but does so very efficiently.

Solution 1

Main Idea. We will design a divide-and-conquer algorithm which finds the jth smallest element in all the arrays. This algorithm can be used to find the merged median by calling it with $j = \lfloor n/2 \rfloor$. The algorithm will behave similarly to the median-finding algorithm for an unsorted list. That is, at each step the algorithm chooses a pivot x, and then splits the elements into those smaller than, equal to, and larger than x. There are three cases:

- If j or more elements are less than x, we recurse on these elements, using the same value of j.
- If less than j elements are less than x, but at least j elements are less than or equal to x, it must be the case that x is the jth smallest element. In this case, we can return it immediately.
- Otherwise, we recurse on the elements greater than x. We decrease j by the number of elements that are less than or equal to x accordingly.

We note that because the arrays are sorted, we can do all of this very quickly using binary search - this will be what lets us do better than $O(n)$ time.

Now, all that is left is to decide our pivot. If we were allowed to write a randomized algorithm, we could choose a random pivot, and at least a quarter of the elements would be removed in each level of the recursion with probability $1/2$. Since our algorithm must be determistic, we cannot do this, but we will still remove a quarter of the elements in each recursive call.

To do this, we will find a "weighted" median of medians. Let $A_1^* \dots A_k^*$ refer to the arrays, in increasing median order. There is some i such that the union of $A_1^* \dots A_i^*$ contains at least half the elements and the union of $A_i^* \dots A_k^*$ also contains at least half the elements. After sorting the arrays by their median, we can find this i in $O(k)$ time. If we choose our pivot to be the median of A_i^*, then the pivot is greater than or equal to the medians of $A_1^*, \dots A_i^*$, and thus greater than or equal to half the elements in these arrays. Since these arrays contain at least half the elements, we know the pivot is greater than or equal to at least a quarter of the elements. A similar argument shows the pivot is less than or equal to at least a quarter of the elements. This completes the description of the algorithm.

In the following implementation, we design an auxiliary function MergedSelect$(A_1, \dots, A_k, j)$ which returns the j-th smallest element of the merged array $A_1, \dots, A_k$. The merged median can be found by calling MergedSelect$(A_1, \dots, A_k, \lfloor n/2 \rfloor)$.

Pseudocode.

procedure MergedSelect$(A_1, \dots, A_k, j)$
  if length$[A_1]$ + ... + length$[A_k] = 1$ **then** ▷ (End condition: one element left which is the median!)
    return the only element in those arrays
  $m \leftarrow []$
  for $i = 1, \dots, k$ **do**
    $m[i] = (A_i[l/2], A_i)$ ▷ (Find the median of each array and store (median, array) as a tuple)
  Sort list m according to the first element (medians) in an ascending order.

halfSizeCounter $\leftarrow 0$, indexCounter $\leftarrow 1$
while halfSizeCounter $<$ (length$[A_1] + \cdots +$ length$[A_k])/2$ **do**
halfSizeCounter $\leftarrow$ halfSizeCounter + length$[m$[indexCounter][2]] ▷ (Consistent with 1-index)
indexCounter $\leftarrow$ indexCounter + 1
medianOfMedians $\leftarrow m$[indexCounter][1]
for $i = 1, \ldots, k$ **do**
$B_i \leftarrow A_i[A_i <$ medianOfMedians] ▷ (B gets all the elements in A which are less than the pivot)
$C_i \leftarrow A_i[A_i =$ medianOfMedians] ▷ (C gets all the elements in A which are equal to the pivot)
$D_i \leftarrow A_i[A_i >$ medianOfMedians] ▷ (D gets all the elements in A which are greater than the pivot)
if $j <=$ length$[B_1] + \cdots +$ length$[B_k]$ **then** ▷ (i-th element must be in B)
return MergedSelect$(B_1, \cdots, B_k, j)$
else if $j <=$ length$[B_1] + \cdots +$ length$[B_k]$ +length$[C_1] + \cdots +$ length$[C_k]$ **then** ▷ (In C)
return medianOfMedians
else ▷ (In D)
return MergedSelect$(D_1, \cdots, D_k, j-$length$[B_1]- \cdots -$ length$[B_k]$ $-$length$[C_1] - \cdots -$ length$[C_k])$

Correctness. The proof of correctness is very similar to the one shown in the book median-finding section, we give it here for completeness.

We can prove the algorithm is correct by induction:

Base case: Our algorithm always behaves correctly when there is only one element in all the arrays.

Inductive step: Assume our algorithm works correctly when there are at most m elements in all the arrays. We will show our algorithm works correctly when there are $m+1$ elements in all the arrays. If there are j or more elements in $B_1 \ldots B_k$ (elements which are less than medianOfMedians), then the jth smallest element is the jth smallest element in $B_1 \ldots B_k$, and by the inductive hypothesis the recursive call finds that element correctly. If there are less than j elements in $B_1 \ldots B_k$, but at least j elements in $B_1 \ldots B_k, C_1 \ldots C_k$, then $C_1 \ldots C_k$ must contain the jth smallest element, i.e. medianOfMedians is exactly the jth smallest element. Otherwise, the jth smallest element is the j'th smallest element in $D_1 \ldots D_k$, where j' is j minus the total number of elements in $B_1 \ldots B_k, C_1 \ldots C_k$. In this case, again by the inductive hypothesis our recursive call finds the element correctly.

Running time. Finding the median of each array is essentially choosing the middle element which takes $O(1)$ per array. Then, finding medianOfMedians takes $O(k \log k)$ time since we need to sort them. Then, we use this element to partition the elements in each list. By binary search we can find which are bigger and which are smaller, so partitioning requires $O(k \log l)$ time.

For now, we'll assume that we eliminate at least a quarter of the elements from the arrays in each level (we'll show this formally at the end). Then we have the recurrence relation

$$T(n) \leq O(k \log l) + O(k \log k) + T(3n/4).$$

The first term is the time to find the split index for the arrays using binary search in each array. The second for finding the median of medians and the third for the recursive call. Initially there were $n = kl$ elements in all, and after $O(\log l)$ recursive calls we have $O(k)$ elements, in which case we can finish in time $O(k)$. Thus, the total cost is bounded by the time for $O(\log l)$ recursive calls. This is bounded by $O(k \log^2 l + k \log l \log k)$.

Showing we chose a good pivot: Suppose $A_1^*, \dots A_k^*$ are the arrays in increasing median order. The algorithm above adds up the size of each array until the sum is at least half the total size (which we'll call s). Suppose we stop adding at array A_i^*. Then "medianOfMedians" guarantees that

$$\text{length}(A_1^*) + \cdots + \text{length}(A_i^*) \geq s/2$$

and

$$\text{median}(A_1^*) \leq \cdots \text{median}(A_i^*) \leq \text{medianOfMedians}.$$

In arrays $A_1^*, \dots A_i^*$, each median is greater than or equal to at half of the elements in each array. There are $\geq s/2$ elements in those arrays. Since each median is less than or equal to our "median of medians", the "median of medians" must be greater than or equal to at least $s/4$ elements in all k arrays. Similarly, we know that the total size of $A_i^* \dots A_k$ is at least $s/2$ (if it was not, then the while loop would have stopped after looking at A_{i-1}^*, not at A_i^*), so the "median of medians" must be less than or equal to at least $s/4$ elements in all k arrays. This shows that we will remove a quarter of the elements in each recursive call.

Solution 2

Main Idea: The goal of this algorithm is to in one iteration: delete some number of elements that are smaller than or equal to the median (i.e. in the bottom half of all numbers), and then delete the same number of elements that are larger than or equal to the median (i.e. in the top half of all numbers). If it succeeds, then the median will stay the median after one iteration. After repeating this for enough iterations, the number of elements will be small and then one can just use brute force search or an algorithm from class to find the median.

More formally: In the first iteration of our algorithm, it sorts all the lists in increasing median order. It then deletes the left half of the first list (the list with the smallest median), i.e. all elements to the left of the median. We also delete the right half of the last list. The elements we deleted in the first list are in the smaller half of all elements, and the elements we deleted in the last list are in the larger half of all elements, as desired.

However, after the first iteration, we have two problems. First, at the start of future iterations, the first/last list no longer necessarily have the smallest/largest median, so the lists are no longer sorted. To fix this, at the end of each iteration we binary search over the k lists, we can insert these lists into the correct position at the end of the iteration, maintaining that the list of lists is sorted in median order. Second, the lists with the smallest and largest median in iterations after the first iteration may no longer be the same size, so if we naively delete half of the elements from each list, we are not guaranteed to maintain that the median is the median after deletions. To fix this, let l_1, l_2 be the length of these two lists. We will delete $\lfloor \min(l_1, l_2)/2 \rfloor$ elements from each list instead of half of the elements in both lists. This guarantees that our deletions are "balanced" and thus the median remains the median.

(We do have to handle an edge case where if a list is size 1, we might not be able to delete the only element left in it as it might be the median (e.g. consider when the lists are $\{2\}, \{1, 4\}, \{1, 4\}$. Then we can't delete 2 because it is the median, and deleting it changes

the median), but keeping that list might stop our algorithm from making progress (since $\lfloor 1/2 \rfloor = 0$). As a lazy fix, when a list is size 1 we can insert it into another list arbitrarily via binary search. This is really not what we're trying to test with this problem, so we'll ignore this when analyzing the runtime.)

As a base case, when the number of remaining elements is less than k, we can just concatenate all the lists, and then sort the lists to find the median.

Pseudocode:

Note: For simplicity, in this pseudocode, we use A_i to refer to the i-th list using the algorithm's current ordering of the lists, not the order given in the input. Similarly, we use k to refer to the number of remaining lists, not the number of lists we started with.

Sort $A_1, \ldots A_k$ by increasing median
while Total size of $A_1 \ldots A_k > k$ **do**
  Let l_1 be the length of A_1, l_2 be the length of A_k
  Delete the first $\lfloor \min(l_1, l_2)/2 \rfloor$ elements of A_1,
  Delete the last $\lfloor \min(l_1, l_2)/2 \rfloor$ elements from the end of A_k
  if A_1 or A_k is size 1 **then**
    Insert its last element into another list arbitrarily via binary search, then delete it.
  Insert A_1, A_k into the list of lists such that they remain sorted via binary search
Merge all remaining lists together, sort the merged list, output its median

Proof of correctness:

It suffices to show that in one iteration, the elements we delete from A_1 are in the smaller half of all elements (i.e. appear before the median in the merged list), and the elements we delete in A_2 are in the larger half of all elements. This ensures the median remains the median after any series of deletions. In any iteration, consider the elements we delete from A_1. These elements are less than or equal to A_1's median. In particular, that means they are less than or equal to all the medians, and thus also less than the right half of every list. Since this comprises at least half of all remaining elements, we have shown that the elements we delete from A_1 are in the bottom half of all elements. A symmetric argument can be used to argue about the elements we delete from A_k.

Runtime analysis:

Since every list starts out size l and has half its elements removed when it is the smaller of the two lists whose elements are being removed, every list can only be the smaller of the two lists whose elements are being removed $O(\log l)$ times. This means the while loop runs for at most $O(k \log l)$ iterations. Trimming the lists can be done in $O(1)$ time, but binary searching to reinsert the lists takes $O(\log k)$ time. So the while loop runs in $O(k \log k \log l)$ time. The base case runs in $O(k \log k)$ time.

6 (★★) Fourier Transform Basics

Answer the following. Do any required matrix multiplications by hand, using e.g. a Fourier transform calculator will not get you full credit.

(a) What is the Fourier transform of $(3, i, 2, 4)$?

(b) Find x and y such that $\text{FT}(x) = (5, 2, 1, -i)$ and $\text{FT}(y) = (4, 4, i, i)$.

(c) *Using part b*, find z such that $\mathrm{FT}(z) = (1, -2, 1-i, -2i)$. (*Hint:* Observe that $\mathrm{FT}(v)$ is a linear transform of v, and recall the properties of linear transforms).

(d) Compute $(2x^2 + 1)(x + 4)$ using a Fourier transform. (*Hint:* Recall that to multiply two polynomials, first, they must both be converted by the Fourier transformation, then multiplied pointwise, and finally converted back to coefficient form)

Solution:

(a) The primitive root of unity we consider in this case is $\omega = e^{(2*i*\pi)/4} = i$. The Fourier transform is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 3 \\ i \\ 2 \\ 4 \end{pmatrix} = \boxed{\begin{pmatrix} 9+i \\ -4i \\ 1-i \\ 2+4i \end{pmatrix}}.$$

(b) To find the inverse Fourier transform, we just need to take the Fourier transform with $\omega = -i$ and then normalize it. In particular

$$x = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 5 \\ 2 \\ 1 \\ -i \end{pmatrix} = \boxed{\frac{1}{4}\begin{pmatrix} 8-i \\ 5-2i \\ 4+i \\ 3+2i \end{pmatrix}}$$

$$y = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 4 \\ 4 \\ i \\ i \end{pmatrix} = \boxed{\frac{1}{4}\begin{pmatrix} 8+2i \\ 3-5i \\ 0 \\ 5+3i \end{pmatrix}}.$$

(c) Notice first that the FT is a linear transform (since it can be expressed as just a matrix multiplication). Notice then that $\mathrm{FT}(z) = \mathrm{FT}(x) - \mathrm{FT}(y)$, so $\mathrm{FT}(z) = \mathrm{FT}(x - y)$, which tells us that $z = x - y$. Hence,

$$\boxed{z = \frac{1}{4}\begin{pmatrix} -3i \\ 2+3i \\ 4+i \\ -2-i \end{pmatrix}}.$$

For this part, only partial credit is awarded for not using part (b). Full credits for getting $z = x - y$, even if some computations are wrong in part (b) or (c).

(d) Observe that Fourier transform of $(1, 0, 2, 0)$ is (we take Fourier transform with $\omega = i$)

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \\ 3 \\ -1 \end{pmatrix}$$

Again, the Fourier transform of $(4, 1, 0, 0)$ is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 4 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 4+i \\ 3 \\ 4-i \end{pmatrix}$$

Multiplying the above pointwise and taking inverse Fourier transform, we get

$$\frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 15 \\ -4-i \\ 9 \\ -4+i \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 8 \\ 2 \end{pmatrix}$$

This gives us the polynomial $4 + x + 8x^2 + 2x^3$

7 (★★) Modular Fourier Transform

Fourier transforms (FT) have to deal with computations involving irrational numbers which can be tricky to implement in practice. Motivated by this, in this problem you will demonstrate how to do a Fourier transform in modular arithmetic, using modulo 5 as an example.

(a) There exists $\omega \in \{0, 1, 2, 3, 4\}$ such that all the powers $\omega, \omega^2, \ldots, \omega^4$ are distinct (modulo 5). When doing the FT in modulo 5, this ω will serve a similar role to the primitive root of unity in our standard FT. Find such an ω (there are multiple, you only need to find one), and show that $\omega + \omega^2 + \omega^3 + \omega^4 = 0 \pmod 5$. (Interestingly, for any prime modulus there is such a number.)

(b) Using the matrix form of the FT, produce the transform of the sequence $(0, 1, 0, 2)$ modulo 5; that is, multiply this vector by the matrix $M_4(\omega)$, for the value ω you found earlier. Be sure to explicitly write out the FT matrix you will be using (with specific values, not just powers of ω). In the matrix multiplication, all calculations should be performed modulo 5.

(c) Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. (Again all arithmetic should be performed modulo 5.)

(d) Now show how to multiply the polynomials $2x^2 + 3$ and $-x + 3$ using the FT modulo 5.

Solution:

(a) Observe that taking $\boxed{\omega = 2}$ produces the following powers: $(\omega, \omega^2, \omega^3, \omega^4) = (2, 4, 3, 1)$. Verify that

$$\omega + \omega^2 + \omega^3 + \omega^4 = 1 + 2 + 3 + 4 = 10 = 0 \pmod 5.$$

Alternatively, $\boxed{\omega = 3}$*, producing* $(3, 4, 2, 1)$*, also works.*

(b) For $\omega = 2$:

$$M_4(2) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}.$$

Multiplying with the sequence $(0, 1, 0, 2)$ we get the vector $\boxed{(3, 3, 2, 2)}$.

For $\omega = 3$, we instead get:

$$M_4(3) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3 & 4 & 2 \\ 1 & 4 & 1 & 4 \\ 1 & 2 & 4 & 3 \end{bmatrix}.$$

and the vector $\boxed{(3, 2, 2, 3)}$.

(c) For $\omega = 2$, the inverse matrix of $M_4(2)$ is the matrix

$$4^{-1} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3 & 4 & 2 \\ 1 & 4 & 1 & 4 \\ 1 & 2 & 4 & 3 \end{bmatrix}$$

Verify that multiplying these two matrices mod 5 equals the identity. Also multiply this matrix with vector $(3, 3, 2, 2)$ to get the original sequence.

For $\omega = 3$:, the inverse matrix of $M_4(3)$ is the matrix

$$4^{-1} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}$$

(d) For $\omega = 2$: We first express the polynomials as vectors of dimension 4 over the integers mod 5: $a = (3, 0, 2, 0)$, and $b = (3, -1, 0, 0) = (3, 4, 0, 0)$ respectively. We then apply the matrix $M_4(2)$ to both to get the transform of the two sequences. That produces $(0, 1, 0, 1)$ and $(2, 1, 4, 0)$ respectively. Then we just multiply the vectors coordinate-wise to get $(0, 1, 0, 0)$. This is the transform of the product of the two polynomials. Now, all we have to do is multiply by the inverse FT matrix $M_4(2)^{-1}$ to get the final polynomial in the coefficient representation. Recall that whe working with the FT outside of modspace, our inverse matrix of $M_4(\omega)$ would be given by $\frac{1}{4}M_4(\omega^{-1})$. In modspace, we can replace $\frac{1}{4}$ with the multiplicative inverse of 4, and ω^{-1} with the multiplicative inverse of 2. The properties of 2 that we found in the first part allow for this identity to hold. Thus the product is as follows: $\boxed{(4, 2, 1, 3)}$ or $\boxed{3x^3 + x^2 + 2x + 4}$.

For $\omega = 3$, the transforms are $(0, 1, 0, 1)$ and $(2, 0, 4, 1)$ and the coordinate-wise product is $(0, 0, 0, 1)$. The final product remains $(4, 2, 1, 3)$.

8 (★★★) Polynomial from roots

Given a polynomial with exactly n distinct roots at $r_1, \ldots, r_n$, compute the coefficient representation of this polynomial in time. Your runtime should be $O(n \log^c n)$ for some constant c (you should specify what c is in your runtime analysis). There may be multiple possible answers, but your algorithm should return the polynomial where the coefficient of the highest degree term is 1. You can give only the main idea and runtime analysis, a four part solution is not required.

Note: A root of a polynomial p is a number r such that $p(r) = 0$. The polynomial with roots $r_1, ..., r_k$ can be expressed as $\prod_i (x - r_i)$.

Solution:

Main idea

We are trying to find the coefficients of the polynomial $(x - r_1)(x - r_2) \cdots (x - r_n)$. Split the roots into two (approximately) equal halves: $r_1, r_2, \ldots, r_{\lfloor n/2 \rfloor}$ and $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Recursively find the polynomial whose roots are $r_1, \ldots, r_{\lfloor n/2 \rfloor}$, and the polynomial whose roots are $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Multiply these two polynomials together using FFT, which takes $O(n \log n)$. When the base case is reached with only 1 root r, return $(x - r)$.

Runtime Analysis

The recurrence for this algorithm is $T(n) = 2T(n/2) + O(n \log n)$. Note that the master theorem doesn't apply here, so we'll have to solve this recurrence relation out directly. If we write out the recurrence tree, we can see that on the ith level, we do $2^i * (\frac{n}{2^i}) * log(\frac{n}{2^i})$ work for $\log n$ levels. We can upper bound the work on each level with $n \log n$, to get a total runtime of $\mathrm{O}(n \log^2 n)$.