CS 170 Homework 2

Due **9/14/2020, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

2 Werewolves

You are playing a party game with n other friends, who play either as werewolves or citizens. You do not know who is a citizen and who is a werewolf, but all your friends do. There are always more citizens than there are werewolves.

Your goal is to identify one citizen.

Your allowed 'query' operation is as follows: you pick two people. You ask each person if their partner is a citizen or werewolf. When you do this, a citizen must tell the truth about the identity of their partner, but a werewolf doesn't have to (they may lie or tell the truth about their partner).

Your algorithm should work regardless of the behavior of the werewolves.

(a) Give a way to test if a single player is a citizen using $O(n)$ queries. Just an informal description of your test and a brief explanation of why it works is needed.

(b) Show how to find a citizen in $O(n \log n)$ queries (where one query is asking a pair of people to identify the other person).

You cannot use a linear-time algorithm here, as we would like you to get practice with divide and conquer.

Hint: Split the group into two groups, and use part (a). What invariant must hold for at least one of the two groups?

Give a 3-part solution.

(c) (**Extra Credit**) Can you give a linear-time algorithm?

Hint: Don't be afraid to sometimes 'throw away' a pair of people once you've asked them to identify their partners.

Solution:

(a) To test if a player x is a citizen, we ask the other $n-1$ players what $x's$ identity is. Claim: x is a citizen if and only if at least half of the other players say x is a citizen. To see this, notice that if x is a citizen, at least half of the remaining players are also citizens, and so regardless of what the werewolves do at least half of the players will say

x is a citizen. On the other hand, if x is a werewolf, then strictly more than half of the remaining players are citizens, and so strictly less than half the players can falsely claim that $x$is a citizen.

(b) **Main idea** The divide and conquer algorithm to find a citizen proceeds by splitting the group of friends into two (roughly) equal sets A and B, and recursively calling the algorithm on A and B: $x = \textit{citizen}(A)$ and $y = \textit{citizen}(B)$, and checking x or y using the procedure in part (a) and returning one who is a citizen. If there is only one player, return that player.

Proof of correctness We will prove that the algorithm returns a citizen if given a group of n people of which a majority are citizens. By strong induction on n:

Base Case If $n = 1$, there is only one person in the group who is a citizen and the algorithm is trivially correct.

Induction hypothesis The claim holds for $k < n$.

Induction step: After partitioning the group into two groups A and B, at least one of the two groups has more citizens than werewolves. By the induction hypothesis the algorithm correctly returns a citizen from that group, and so when the procedure from part (a) is invoked on x and y at least one of the two is identified as a citizen.

Running time analysis Two calls to problems of size $n/2$, and then linear time to compare the two people returned to each of the friends in the input group: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$ by Master Theorem.

(c) **Main idea** Split up the friends into pairs and if either says the other is a werewolf, discard both friends. Otherwise discard any one and keep the other friend. If n was odd, use part (a) to test whether the odd man out is a citizen. If yes, you are done, else recurse on the remaining at most $n/2$ friends.

Proof of correctness After each pass through the algorithm, citizens remain in the majority if they were in the majority before the pass. To see this, let n_1, n_2 and n_3 be the pairs of friends with both citizens, both werewolves and one of each respectively. Then the fact that citizens are in the majority means that $n_1 > n_2$. Note that all the n_3 pairs of the third kind get discarded, and one friend is retained from each of the n_1 pairs of the first kind. So we are left with at most $n_1 + n_2$ friends of whom a majority n_1 are citizens. It is straightforward to now turn this into a formal proof of correctness by strong induction on n.

Running time analysis In a single run of the algorithm on an input set of size n, we do $O(n)$ work to check whether f_1 is a citizen in the case that n is odd and to pair up the remaining friends and pruning the candidate set to at most $n/2$ people. Therefore, the runtime is given by the following recursion:

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

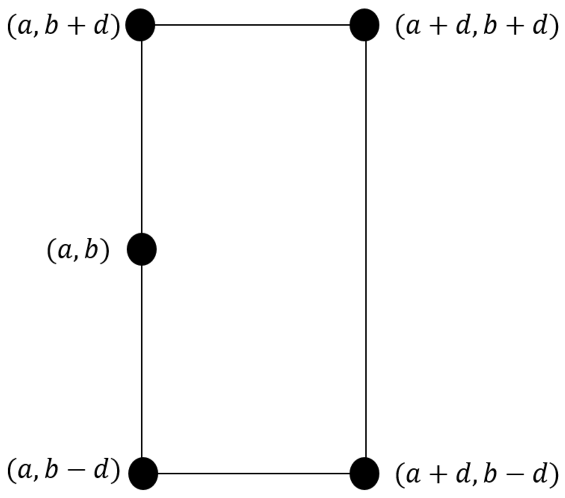
3 Agent Meetup

Manhattan has an "amazing" road system where streets form a checkerboard pattern, and all roads are either straight North-South or East-West. We simplify Manhattan's roadmap by assuming that each pair x, y, where x and y are integers, corresponds to an intersection. As a result, the distance between any two intersections can be measured as the *Manhattan distance* between them, i.e. $|x_i - x_j| + |y_i - y_j|$. You, working as a mission coordinator at the CS 170 Secret Service Agency, have to arrange a meeting between two of n secret agents located at intersections across Manhattan. Hence, your goal is to find the two agents that are the closest to each other, as measured by their Manhattan distance.

In this problem, you will devise an efficient algorithm for this special purpose. As mentioned before, you can assume that the coordinates of the agents are integer values, i.e. the ith agent is at location (x_i, y_i) where x_i, y_i are integers.

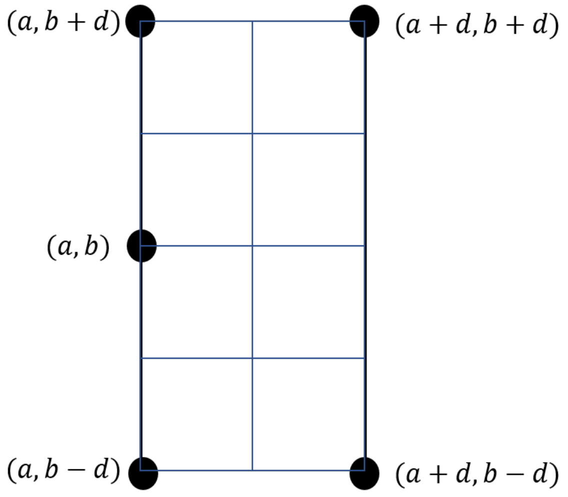
Note: This problem is very geometric, we suggest you draw examples when working on it!

(a) Let (a, b) be an arbitary intersection. Suppose all agents i for which $x_i > a$ are Manhattan distance strictly greater than d apart from each other, where $d > 0$. Give an upper bound on the number of agents i that satisfy $a \le x_i \le a + d$ and $b - d \le y_i \le b + d$. In other words, how many agents can fit in this rectangle (visualized below) without two of these agents being Manhattan distance d or less apart?



Briefly justify your answer. A reasonable bound suffices, you are not expected to provide the tightest possible bound.

Solution: An easy upper bound is 8. Split this region into eight squares of side-length $d/2$.



At most one agent can be in each square, since the distance between every pair of points within each square is at most d.

Since we assume all locations are integer, a better bound is possible but likely not as straightforward.

Since any constant will result in the same asymptotic result in the next part, any reasonable constant with a correct justification should get full credit for this problem.

(b) Design a divide and conquer algorithm to find the minimum Manhattan distance between any two agents. **Give a three-part solution.** A solution that has runtime within logarithmic factors of the optimal runtime will still get full credit.

Hint: Think about Maximum Subarray Sum from HW1 and the cases we used in that problem. Can similar cases be used here? How can we use part (a) to handle one of those cases efficiently?

Solution: We give an algorithm whose runtime is $O(n \log^2 n)$. It is possible to improve it to $O(n \log n)$ by optimizing parts of the algorithm, to simplify the solution we won't bother to do so here.

Main idea.

The high level idea is similar to maximum subarray sum. Define L and R to be the left and right half of the agents when sorted by x-coordinate; why this is the right way to split the agents will become clear later. The closest pair of agents are either (1) both in L, (2) both in R, or (3) one is in L and one is in R. We can handle cases (1) and (2) recursively. It might seem like to handle case (3) we have to find the closest distance between an agent in L and an agent in R. The key idea is that letting the smallest distance we found in cases (1) and (2) be $d+1$, we only need to consider pairs of agents in case (3) that are distance at most d apart (recall all distances are integer). This idea will allow us ignore many pairs of agents in L and R.

If m_x is the median x-coordinate of all agents, note that any agent in R with $x_i > m_x + d$ can't be distance d or less from any agent in L. So let R_{close} be all agents in R for which

$x_i \le m_x + d$; we only need to consider agents in R_{close} in case (3). Now consider any agent i in L at position (x_i, y_i). We can skip computing the distance between agent i and any agent in R_{close} whose y-coordinate does not lie in the range $[y_i - d, y_i + d]$. If we sort R_{close} by y-coordinate beforehand, we can quickly identify agents in R_{close} in this range by binary searching.

So our non-recursive work is: for each agent i in L, identify the agents in R_{close} in the y-coordinate range $[y_i - d, y_i + d]$ and compute the distance between i and these agents. We then output the smallest distance we found either between the recursive calls and this non-recursive step.

As a base case, if there are only two agents, we can just report the distance between them.

Correctness. If there are only two agents, our algorithm is of course correct. Otherwise, we proceed by induction.

If the closest pair of agents are distance $d + 1$ apart and both in L or both in R, one of the recursive calls returns the right answer by our inductive hypothesis. Otherwise, the closest pair of agents is distance at most d apart and split between L and R. By part (a) and the definition of R_{close}, every pair of agents in $L \times R$ at distance at most d has their distance computed by the algorithm, in which case the final output will be correct.

Runtime analysis

The non-recursive work we do is as follows.

- Sorting the list of all agents by x-coordinate, which takes $O(n \log n)$ time.
- Sorting R_{close} by y-coordinate, which takes $O(n \log n)$ time.
- Locating the agents in R_{close} to compare agents in L to. Since this takes $O(\log n)$ time per agent, this overall takes $O(n \log n)$ time.
- By part (a), for each agent $i \in L$, there are $O(1)$ agents j for which $m_x \le x_j \le m_x + d, y_i - d \le y_j \le y_i + d$. So we do $O(n)$ non-recursive distance computations in $O(n)$ time.

So the recurrence relation is $T(n) = 2T(n/2) + O(n \log n)$. We can't use the master theorem, but drawing the tree of subproblems, we can see the i th level of recursion has 2^i subproblems doing $O(n/2^i \log(n/2^i))$ work. So the work per level is $O(n \log n)$, i.e. the total work is $O(n \log^2 n)$. (We can also show a lower bound of $\Omega(n \log^2 n)$ since levels 0 to $\frac{1}{2} \log n$ do $\Omega(n \log n)$ work.)

4 Practice with Polynomial Multiplication with FFT

(a) Suppose that you want to multiply the two polynomials $x + 1$ and $2x + 1$ using the FFT. Choose an appropriate power of two n and corresponding root of unity ω_n to use in FFT, find the FFT of the two sequences, multiply the results componentwise, and compute the inverse FFT to get the final result.

(b) Repeat for the pair of polynomials $1 + x + 2x^2$ and $2 + 3x$.

Solution:

(a) The resulting polynomial will be degree 2, i.e. have 3 coefficients and be uniquely identified by its value at 3 points. FFT requires us to evaluate the polynomial at a number of points that is a power of 2. So we round up 3 to the nearest power of 2 and get that the appropriate power of two to use is 4. Using $\omega_4 = i$, the FFT of $x + 1$ is:

$$FFT(1,1,0,0) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1+i \\ 0 \\ 1-i \end{bmatrix}$$

and the FFT of $2x + 1$ is

$$FFT(1,1,0,0) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 1+2i \\ -1 \\ 1-2i \end{bmatrix}$$

.

Point-wise multiplying their FFTs, we get that the FFT of their product is $(6, -1 + 3i, 0, -1 - 3i)$. Multiplying this by the inverse FFT matrix we get:

$$IFFT(6, -1+3i, 0, -1-3i) = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \cdot \begin{bmatrix} 6 \\ -1+3i \\ 0 \\ -1-3i \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 0 \end{bmatrix}$$

So their product is

$$2x^2 + 3x + 1$$

.

(b) The resulting polynomial will be degree 3, i.e. have 4 coefficients. For the same reasons as before, we use $\omega_4 = i$ as the root of unity in FFT. Using $\omega_4 = i$, the FFT of $2x^2 + x + 1$ is

$$FFT(1,1,2,0) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ -1+i \\ 2 \\ -1-i \end{bmatrix}$$

and the FFT of $3x+2$ is

$$FFT(2,3,0,0) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 2+3i \\ -1 \\ 2-3i \end{bmatrix}$$

The FFT of their product is then $(20, -5-i, -2, -5+i)$. The inverse FFT of this is:

$$IFFT(20, -5-i, -2, -5+i) = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \cdot \begin{bmatrix} 20 \\ -5-i \\ -2 \\ -5+i \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 7 \\ 6 \end{bmatrix}$$

So the product of the polynomials is $6x^3 + 7x^2 + 5x + 2$.

5 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 ω are 4^{th} roots of unity (modulo 5), i.e., solutions to $z^4 = 1$. When doing the FT in modulo 5, this ω will serve a similar role to the primitive root of unity in our standard FT. Show that $\{1,2,3,4\}$ are the 4^{th} roots of unity (modulo 5). Also show that $1 + \omega + \omega^2 + \omega^3 = 0 \pmod 5$ for $\omega = 2$.

(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 $\omega = 2$. 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) We can check that $1^4 = 1 \pmod 5$,
$2^4 = 16 = 1 \pmod 5$,
$3^4 = 81 = 1 \pmod 5$,
$4^4 = 256 = 1 \pmod 5$.

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

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

(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:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ 2 \\ 2 \end{bmatrix}$$

(c) Recall that when 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 $4 \pmod 5$ (which is 4), and ω^{-1} with the multiplicative inverse of $2 \mod 5$ (which is 3).

So for $\omega = 2$, the inverse matrix of $M_4(2)$ is the matrix

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

We can verify that multiplying these two matrices mod 5 equals the identity. Also:

$$4 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3 & 4 & 2 \\ 1 & 4 & 1 & 4 \\ 1 & 2 & 4 & 3 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 3 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 2 \end{bmatrix}$$

(d) Just like in FFT, we will multiply $M_4(\omega)$ by the coefficient representations of the polynomials, point-wise multiply the resulting vectors, and then multiply the resulting vector by $M_4(2)^{-1}$ to get a coefficient representation for their product.

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.

Multiplying the matrix $M_4(2)$ with both produces $(0, 1, 0, 1)$ and $(2, 1, 4, 0)$ respectively.

Then we just multiply the vectors coordinate-wise to get $(0, 1, 0, 0)$.

Now, we multiply inverse FT matrix $M_4(2)^{-1}$ that we wrote down in the previous part to get the final polynomial in the coefficient representation. The product is as follows: $\boxed{(4, 2, 1, 3)}$. This corresponds to the polynomial $\boxed{3x^3 + x^2 + 2x + 4}$.

We can verify this is correct by multiplying the two polynomials using FOIL to get $-2x^3 + 6x^2 - 3x + 9$. The coefficients of these two polynomials are equivalent $\pmod 5$.

6 Sparse FFT

Given an input vector $(p_0, p_1, \ldots, p_{n-1})$, the FFT algorithm computes the following matrix-vector product:

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

in time $O(n \log n)$. Sometimes we want to apply FFT to a *sparse* vector, i.e. a vector where most p_i values are 0. In this case, it is not necessary to do $O(n \log n)$ work.

Modify the FFT algorithm so that it runs in $O(n \log k)$ time rather than $O(n \log n)$ time, where k is the number of p_i that are non-zero. Give a description of your algorithm and a runtime analysis; no proof of correctness is necessary, as long as your main idea is clear.

Solution:

Solution 1:

Main idea. Note that naively, we can do the matrix-vector computation in $O(nk)$ time without recursing, since we can ignore all but k columns in the matrix and the corresponding entries in the vector.

We run FFT for the first $\log k$ levels of recursion. At the $\log k$-th level of recursion, rather than recurse further, we do the matrix-vector computations using the naive method instead.

Runtime analysis.

The work done at levels 0 to $\log k$ of the recursion remains $O(n)$. For the subproblems at the $\log k$-th level which we solve naively, let k_i denote the number of non-zero coefficients in the ith subproblem. The naive method then takes $O(\frac{n}{k} \cdot k_i)$ time for the ith subproblem. Since $\sum_i k_i = k$, the total time spent at this level is $O(n)$. Putting it together, this algorithm takes $O(n \log k)$ time.

Solution 2:

Main idea.

Modify the FFT procedure to check whether the input is all-zeroes. In this case output all-zeroes. Note that this requires an extra $O(n)$ steps, but the advantage is that in this case the algorithm does not recurse any further.

Runtime analysis. The main point is that since there are only k non-zero coefficients, in this modified FFT there can't be any level of recursion where more than k subproblems have input that is not the all-zeroes vector. So:

- The work done at levels 0 to $\log k$ remains $O(n)$.
- For level $i > \log k$, there are at most k subproblems instead of 2^i subproblems, but the work per subproblem is still $O(n/2^i)$. So the total work in level i is $n/2^{i-\log k}$. In other words, the work per level starts decreasing geometrically starting at level $\log k$, so the total work done at levels $\log k + 1$ to $\log n$ is $O(n)$.

Putting it together, the runtime is $O(n \log k)$.

7 Polynomial from roots

A root of a polynomial p is a number r such that $p(r) = 0$. Given a polynomial with exactly n distinct roots at $r_1, \ldots, r_n$, compute the coefficient representation of this polynomial. Your runtime should be $\mathcal{O}(n \log^c n)$ for some constant $c > 0$ (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 algorithm description and runtime analysis, a three-part solution is not required.

Hint: One polynomial with roots $r_1, ..., r_k$ is

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

. Recall that you can use FFT as a black-box algorithm for multiplying polynomials in time $O(n \log n)$.

Solution:

Main idea

Recursively multiply $(x - r_1), (x - r_2), \cdots (x - r_{n/2})$ to obtain polynomial $P(x)$, and $(x - r_{n/2+1}), (x - r_{n/2+2}), \cdots (x - r_n)$ to obtain polynomial $Q(x)$, and then use FFT to multiply $P(x)Q(x)$.

The base case is when we only want to multiply one term $(x - r_1)$, in which case we can just output that term.

Runtime Analysis

The recurrence for this algorithm is $T(n) = 2T(n/2) + O(n \log n)$, which we showed has solution $O(n \log^2 n)$ in Problem 3.