1. (20 pts.) Pattern matching, with tolerance for noise

We are given binary strings s,t; s is m bits long, and t is n bits long, and $m < n$. We are also given an integer k. We want to find whether s occurs as a substring of t, but with $\leq k$ errors, and if so, find all such matches. In other words, we want to determine whether there exists an index i such that $s_0, s_1, \ldots, s_{m-1}$ agrees with $t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+m-1}$ in all but k bits; and if yes, find all such indices i.

(a) Describe an $O(mn)$ time algorithm for this string matching problem. Just show the pseudocode; you don't need to give a proof of correctness or show the running time.

Solution: We try matching s against t at all possible shifts, counting how many errors there are at each one:

Algorithm Match($s[0 \ldots m-1], t[0 \ldots n-1], k$):
1. Set $M := \{\}$.
2. For $i := 0, \ldots, n-m$:
3. Set $e := 0$.
4. For $j := 0, \ldots, m-1$:
5. If $s[j] \neq t[i+j]$, set $e := e+1$.
6. If $e \leq k$, add i to M.
7. Return M.

The runtime of this algorithm is clearly $O((n-m) \cdot m) = O(nm)$.

(b) Let's work towards a faster algorithm. Suggest a way to choose polynomials $p(x), q(x)$ of degree $m-1, n-1$, respectively, with the following property: the coefficient of x^{m-1+i} in $p(x)q(x)$ is $m - 2d(i)$, where $d(i)$ is the number of bits that differ between $s_0, s_1, \ldots, s_{m-1}$ and $t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+m-1}$.

Hint: use coefficients $+1$ and -1.

Solution: Define

$$p(x) = \sum_{i=0}^{m-1} (-1)^{s[m-1-i]} x^i$$

and

$$q(x) = \sum_{i=0}^{n-1} (-1)^{t[i]} x^i.$$

Multiplying, we get

$$p(x) \cdot q(x) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (-1)^{s[m-1-i]+t[j]} x^{i+j}.$$

Therefore the coefficient of $x^{m-1+\ell}$ in $p(x) \cdot q(x)$ is

$$\sum_{\substack{i+j=m-1+\ell \\ 0 \leq i < m \\ 0 \leq j < n}} (-1)^{s[m-1-i]+t[j]} = \sum_{i=0}^{m-1} (-1)^{s[m-1-i]+t[m-1-i+\ell]} = \sum_{i=0}^{m-1} (-1)^{s[i]+t[\ell+i]} = m - 2d(\ell),$$

using the fact that the sum of $m-d(\ell)$ $+1$'s and $d(\ell)$ -1's is $m-2d(\ell)$. Also, we have used the fact that $(-1)^{y+z}$ is $+1$ if $y=z$ and -1 if $y\neq z$ (when y,z are bits).

Alternatively, you could have written your answer as follows. Define

$$p(x) = (-1)^{s[m-1]} + (-1)^{s[m-2]}x + (-1)^{s[m-3]}x^2 + \cdots + (-1)^{s[0]}x^{m-1}$$
$$q(x) = (-1)^{t[0]} + (-1)^{t[1]}x + (-1)^{t[2]}x^2 + \cdots + (-1)^{t[n-1]}x^{n-1}.$$

Multiplying, we find

$$\begin{aligned} p(x)\cdot q(x) =& (-1)^{s[m-1]+t[0]} + [(-1)^{s[m-2]+t[0]} + (-1)^{s[m-1]+t[1]}]x \\ &+ [(-1)^{s[m-3]+t[0]} + (-1)^{s[m-2]+t[1]} + (-1)^{s[m-1]+t[2]}]x^2 + \cdots \end{aligned}$$

(For instance, we can see that the coefficient of x^2 is the number of bits that differ between $s[m-3], s[m-2], s[m-1]$ and $t[0], t[1], t[2]$.) Now the coefficient of $x^{m-1+\ell}$ in $p(x)\cdot q(x)$ is

$$(-1)^{s[0]+t[\ell]} + (-1)^{s[1]+t[\ell+1]} + \cdots + (-1)^{s[m-1]+t[m+\ell-1]},$$

and this is $m-2d(\ell)$, as explained above.

(c) Describe an $O(n\lg n)$ time algorithm for this string matching problem, taking advantage of the polynomials $p(x), q(x)$ from part (b).

Solution: Construction of the polynomials $p(x)$ and $q(x)$ above clearly can be done in $O(n)$ time, and each has degree $O(n)$. So we can multiply them in $O(n\log n)$ time with the FFT, and in linear time find all indices ℓ such that the coefficient of $x^{m-1+\ell}$ in the product is $m-2k$ or larger. By part (b), every such index corresponds to a match between s and $t[\ell \ldots \ell+m-1]$ with at most k errors. Therefore this algorithm is correct, and its total runtime is $O(n\log n)$.

(d) Now imagine that s,t are not binary strings, but DNA sequences: each position is either A, C, G, or T (rather than 0 or 1). As before, we want to check whether s matches any substring of t with $\leq k$ errors (i.e., $s_0, s_1, \ldots, s_{m-1}$ agrees with $t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+m-1}$ in all but k letters), and if so, output the location of all such matches. Describe an $O(n\lg n)$ time algorithm for this problem.

Hint: encode each letter into 4 bits.

Solution: Replace the letters by 1000, 0100, 0010, and 0001, so that s and t are converted to binary strings s' and t'. Then we can compute $p(x)$ and $q(x)$ and before, and return all indices ℓ such that the coefficient of $x^{4m-1+4\ell}$ in the product is $4m-4k$ or larger. By part (b), every such index corresponds to a match between s' and $t'[4\ell \ldots 4\ell+4m-1]$ with at most $2k$ errors. This match in turn corresponds to a match between s and $t[\ell \ldots \ell+m-1]$ with at most k errors, since the codes for two different letters differ on exactly two bits (and we're only looking at shifts which are multiples of 4, so that the codes in s' and t' line up). Therefore this algorithm is correct, and since we only increase the sizes of s and t by a constant factor (namely 4), it runs in $O(n\log n)$ time.

2. (10 pts.) 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 strictly faster than $O(n^2)$. There may be multiple possible answers, but your algorithm should return the polynomial where the coefficient of the highest degree term is 1.

Note: A root of a polynomial p is a number r such that $p(r)=0$.

Solution: 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 $(x-r)$. The recurrence for this algorithm is $T(n) = 2T(n/2) + O(n\log n)$, which gives $O(n\log^2 n)$.

3. (10 pts.) Triple sum

Design an efficient algorithm for the following problem. We are given an array $A[0..n-1]$ with n elements, where each element of A is an integer in the range $-10n \le A[i] \le 10n$. The algorithm must answer the following yes-or-no question: does there exist indices i, j, k such that $A[i]+A[j]+A[k] = 0$?

Design an $O(n \lg n)$ time algorithm for this problem. Note that you do not need to actually return the indices; just yes or no is enough.

Hint: define a polynomial of degree $O(n)$ based upon A, then use FFT for fast polynomial multiplication.

Reminder: don't forget to include explanation, pseudocode, running time analysis, and proof of correctness.

Solution:

Main idea: First, let's get rid of the negative numbers in our array. Let's add $10n$ to every element in our array, and now we want to know if there exists three indices i, j, k such that $A[i]+A[j]+A[k] = 30n$. From here on out, let $t = 30n$.

Exponentiation converts addition to multiplication. So, define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether $p(x)^3$ contains x^t as a term.

Pseudocode:

Algorithm TripleSum($A[0 \ldots n-1]$, t):
1. Set $p(x) := \sum_{i=0}^{n-1} x^{A[i]}$.
2. Set $q(x) := p(x) \cdot p(x) \cdot p(x)$, computed using the FFT.
3. Return whether the coefficient of x^t in q is nonzero.

Correctness: Observe that

$$q(x) = p(x)^3 = \left(\sum_{0 \le i < n} x^{A[i]} \right)^3 = \sum_{0 \le i,j,k < n} x^{A[i]} x^{A[j]} x^{A[k]} = \sum_{0 \le i,j,k < n} x^{A[i]+A[j]+A[k]}.$$

Therefore, the coefficient of x^t in q is nonzero if and only if there exist indices i, j, k such that $A[i]+A[j]+A[k] = t$. So the algorithm is correct. (In fact, it does more: the coefficient of x^t tells us *how many* such triples (i, j, k) there are.)

Constructing $p(x)$ clearly takes $O(n)$ time. Since $-10n \le A[i] \le 10n$, $p(x)$ is a polynomial of degree at most $20n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^t takes constant time, so overall the algorithm takes $O(n \log n)$ time.

Comment: This problem promised you that each element of the array is in the range $-10n \ldots 10n$. What if we didn't have any such promise? Then the FFT-based method above becomes inefficient (because the degree of the polynomial is as large as the largest element of A). It is easy to find a $O(n^2)$ time algorithm, but no faster algorithm is known. In particular, it is a famous open problem (called the 3SUM problem) whether this problem can be solved more efficiently than $O(n^2)$ time. This problem has been studied extensively, because it is closely connected to a number of problems in computational geometry.

4. (10 pts.) Cycle Detection

Design a linear-time algorithm which, given an undirected graph G and a particular edge e in it, determines whether G has a cycle containing e.

Solution:

The graph has a cycle containing $e = (u, v)$ if and only if u and v are in the same connected component in the graph obtained by deleting e. Thus, we remove the edge $e = (u, v)$ from G and then run DFS starting at node v. If node u is reached at any point in the DFS, return true. Otherwise, return false.

Removing the edge e from an adjacency matrix takes constant time and DFS takes linear time. Thus, this algorithm takes linear time.

5. (20 pts.) Bipartite Graphs

A *bipartite graph* is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V_1$, then there is no edge between u and v).

(a) Give a linear-time algorithm to determine whether an undirected graph is bipartite.

Solution:

Let us identify the sets V_1 and V_2 with the colors red and blue. We perform a DFS on the graph and color alternate levels of the DFS tree as red and blue (clearly they must have different colors). Then the graph is bipartite iff there is no monochromatic edge. This can be checked during DFS itself as such an edge must be a back-edge, since tree edges are never monochromatic by construction and DFS on undirected graphs produces only tree and back edges.

This algorithm consists of a slightly modified DFS, which takes linear time. Thus, our algorithm also takes linear time ($O(|V| + |E|)$).

(b) There are many other ways to formulate this property. For instance, an undirected graph is bipartite if and only if it can be colored with just two colors. We say that a graph can be colored with k colors if we can assign one of k different colors to each vertex such that no adjacent vertices share the same color.

Prove the following formulation: an undirected graph is bipartite if and only if it contains no cycles of odd length.

Solution:

The "only if" part is trivial since an odd cycle cannot be colored by two colors. To prove the "if" direction, consider the run of the above algorithm on a graph which is not bipartite. Let u and v be two vertices such that (u, v) is a monochromatic back-edge and u is an ancestor of v. The path length from u to v in the tree must be even, since they have the same color. This path, along with the back-edge, gives an odd cycle.

(c) At most how many colors are needed to color in an undirected graph with exactly *one* odd-length cycle?

Solution:

If a graph has exactly one odd cycle, it can be colored by 3 colors. To obtain a 3-coloring, delete one edge from the odd cycle. The resulting graph has no odd cycles and can be 2-colored. We now add back the deleted edge and assign a new (third) color to one of its end points.

Rubric:

Please make sure you are using the updated self-grading template (more details on Piazza)

For any part of a problem with 10 points or more, use the following guideline for partial credit:

- All correct: 100%
- A few minor errors: 80%
- On right track, but many/major errors: 50%
- Had some ideas, but wasnt on right track: 20%
- No answer: 0%

For problems with a stated runtime requirement, any solutions not meeting the requirement are not eligible for points.

For rubric items for giving a correct algorithm with proof and runtime justification (i.e. the three parts are not split into separate items), award yourself credit according to the guideline above, while keeping in mind that the algorithm is worth roughly the same as the proof and runtime justification together.

Redemption:

You can redeem points on problems 1, 2, and 5.

Problem 1

- 5 points for correct pseudocode.
- 5 points for observing how to choose polynomials as in the solution (or an alternate correct solution fulfilling the required properties).
- 10 points for giving a correct algorithm in the correct format, with a valid proof and runtime justification.
- 10 points for giving a correct algorithm in the correct format, with a valid proof and runtime justification.

Problem 2

- 10 points for giving a correct algorithm with pseudocode, with runtime fulfilling the requirements stated.
- 5 points for a proof of correctness of the algorithm.
- 5 points for a correct runtime with justification.

Problem 3

- 10 points for giving a correct algorithm with pseudocode, with runtime fulfilling the requirements stated.

- 5 points for a proof of correctness of the algorithm.
- 5 points for a correct runtime with justification.

Problem 4 10 points for giving a correct algorithm in the correct format, with a valid proof and runtime justification.

Problem 5

- 10 points for giving a correct algorithm in the correct format, with a valid proof and runtime justification.
- 5 points for giving a correct proof.
- 5 points for the correct answer with justification.