CS 170 | Efficient Algorithms and Intractable Problems
Fall 2017 | Prasad Raghavendra and Sanjam Garg

Homework 2

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.

- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.

- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Wednesday, September 13, at 4:59pm

1. (★★ level) Minimal Positive Valued Function

Consider a strictly decreasing function $f : \mathbb{N} \to \mathbb{Z}$, such that $f(i) > f(i+1)$ for all $i \in \mathbb{N}$. Assuming we can evaluate f at any number in constant time, we want to find $n = \min\{i \in \mathbb{N} : f(i) < 0\}$. Design a $O(\log n)$ algorithm to compute n.

(Please turn in a four part solution to this problem.)

Solution:
Main Idea: We can compute an upper-bound on n (denote this as n^*) in $O(\log n)$ time. From then on out the algorithm is a slight modification on binary search on the values $f(1), f(2), \ldots, f(n^*)$. The overall runtime is $O(\log n)$.

Pseudocode:

```
procedure SEARCHMIN(function $f$)
    $n^* = 1$
    while $f(n^*) > 0$ do
        $n^* := n^* \cdot 2$
    Output BINARYSEARCH on $[f(\frac{n^*}{2}), f(\frac{n^*}{2}+1), \ldots, f(n^*)]$
```

Note: It is OK to call binary search on $[f(1), f(2), \ldots, f(n^*)]$. This solution would still run in $O(\log n)$ time, but there would be one unnecessary comparison.

Proof of Correctness: We first claim that n^* is indeed an upper-bound on n. Suppose (for the sake of contradiction) that $n > n^*$. Based on how the stopping condition of the while statement $f(n^*) < 0$. The contradicts the fact that $n = \min\{i \in \mathbb{N} : f(i) < 0\}$.

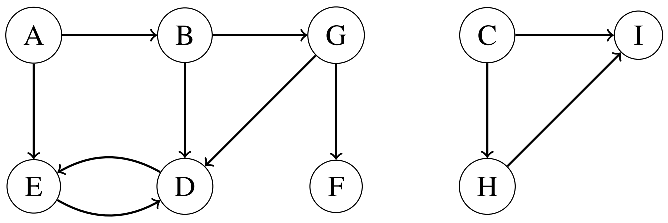
Since we can evaluate f in constant time, this scenario is analogous to binary search (for a negative value) on a length n array where $A[i] = f(i), \frac{n^*}{2} \leq i \leq n$. The proof of correctness of this part follows from the proof of correctness of binary search.

Runtime Analysis: We compute the upper bound n^* in $\lceil \log n \rceil \in O(\log n)$ time. Running binary search on $2^{\lceil \log n \rceil}$ elements also takes $O(\log n)$ time.

Note: It is **imperative** that we do not evaluate all of $f(\frac{n^*}{2}), f(\frac{n^*}{2}+1), \ldots, f(n^*)$, otherwise the runtime would be $O(n)$ instead of $O(\log n)$.

2. (★★ level) Graph Basics

For parts (a) and (b), refer to the figure below. For parts (c) through (f), please prove only for simple graphs; that is, graphs that do not have any parallel edges or self-loops.



(a) Run DFS at node A, trying to visit nodes alphabetically (e.g. given a choice between nodes D and F, visit D first).

- List the nodes in the order you visit them (so each node should appear in the ordering exactly once).
- List each node with its pre- and post-number. The numbering starts from 1 and ends at 18.
- Label each edge as **T**ree, **B**ack, **F**orward or **C**ross.

Solution: Ordering: ABDEGFCHI

nodes	pre-visit	post-visit
A	1	12
B	2	11
D	3	6
E	4	5
G	7	10
F	8	9
C	13	18
H	14	17
I	15	16

Tree: AB, BD, DE, BG, GF, CH, HI

Back: ED

Forward: AE, CI

Cross: GD

(b) Let $|E|$ be the number of edges in a simple graph and $|V|$ be the number of vertices. Show that $|E|$ is in $O(|V|^2)$.

Solution:

The maximum number of edges a graph can have is when every vertex is connected to every other vertex. This describes the 'complete' graph and has $\binom{V}{2} \in O(V^2)$ edges.

(c) For each vertex v_i, let d_i be the *degree*- the number of edges incident to it. Show that $\sum d_i$ must be even.

Solution: Each edge in the graph belongs to precisely 2 vertices. Thus, the total number of edges is $\frac{\sum d_i}{2}$. Since there are an integer number of edges, $\sum d_i$ must be even.

https://www.sharelatex.com/project/59aa37ab1fe674022acb7551

3. (★★★ level) Peak element

Prof. Garg just moved to Berkeley and would like to buy a house with a view on Euclid Ave. Needless to say, there are n houses on Euclid Ave, they are arranged in a single row, and have distinct heights. A house "has a view" if it is taller than its neighbors. For example, if the houses heights were $[3,7,5,9]$, then 7 and 9 would "have a view".

Devise an efficient algorithm to help Prof. Garg find a house with a view on Euclid Ave. (If there are multiple such houses, you may return any of them.)

Solution:

Main idea Divide and conquer. At each step, test if the middle house has a view. If not, at least one of its neighbors must be taller. Recurse on a side with a taller neighbor.

Psuedocode

```
procedure PEAK($a[1..n]$)
    if $n \leq 3$ then                                    ▷ base case
        Return index of max element.

    $m \leftarrow \lceil n/2 \rceil$                       ▷ divide and conquer
    if $a[m] > a[m+1]$ AND $a[m] > a[m-1]$ then
        Return $m$.
    else if $a[m] < a[m-1]$ then
        Return PEAK($a[1..m-1]$).
    else
        Return $m +$ PEAK($a[m+1..n]$).
```

Proof of correctness When $a[m]$ has a view, the algorithm outputs it. If $a[m]$ does not, suppose we are in the case that the algorithm chooses to search $A[1..m-1]$ (symmetric argument for the other case). Now, imagine if we start at $a[m-1]$, and continue searching to the left until we either reach $a[1]$ or a house that has a smaller neighbor to its left. Either way, this house must have a view. Thus, there must be a house with a view in this half of the street. Furthermore, exactly one end house of this half of the street is not an end house of the original street, that is $a[m-1]$, but we know $a[m-1] > a[m]$, so if $A[m-1]$ has a view with respect to this half, it has a view in the original street.

Running time analysis The running time is $\Theta(\log n)$, since at each level we solve **one** problem of size $\frac{n}{2}$, plus constant work (comparing the middle elements to neighbors). $T(n) = T(\frac{n}{2}) + \Theta(1)$, and so $T(n) = \Theta(\log n)$ by the Master Theorem.

4. (★★★ level) Exact Change

Your friend from Mars visits you and wants to buy a CS 170 textbook, which is priced at \$$t$ dollars. Unfortunately, Martians have their own currency. You are given an array $A[0..n-1]$ with n elements representing the value of each Martian coin in dollars. The value of each coin is a distinct integer in the range $0 \leq A[i] \leq 170n$. For some reason, your friend brought only 4 coins of each type.

Design an efficient algorithm that determines, given A and t, whether your friend can give you exact change **using exactly 4 coins** (no more or no less). Note: the algorithm should run asymptotically faster than $O(n^2)$.

(Please turn in a four part solution to this problem.)
Hint: Think about properties of exponents.

Solution:

Main idea: Exponentiation converts addition to multiplication. So, define

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

Notice that $p(x)^4$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} \cdot x^{A[l]} = x^{A[i]+A[j]+A[k]+A[l]}$. We then observe that the coefficient of x^t is the number of combinations of 4 elements of A that sum up to t. So we just need to check whether the coefficient of x^t is greater than 0.

Pseudocode:

```
procedure ExactChange($A[0 \ldots n-1], t$)
    $p(x) := \sum_{i=0}^{n-1} x^{A[i]}$
    $q(x) := p(x) \cdot p(x) \cdot p(x) \cdot p(x)$, computed using the FFT.
    Set $c :=$ the coefficient of $x^t$ in $q$
    Return TRUE if $c \neq 0$, FALSE otherwise
```

Correctness: Observe that

$$q(x) = p(x)^4 = \left(\sum_{0 \leq i < n} x^{A[i]} \right)^4 = \sum_{0 \leq i,j,k,l < n} x^{A[i]} x^{A[j]} x^{A[k]} x^{A[l]} = \sum_{0 \leq i,j,k,l < n} x^{A[i]+A[j]+A[k]+A[l]}.$$

Therefore, the coefficient of x^t in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] + A[l] = t$. So the algorithm is correct. In fact, it does more: the coefficient of x^t tells us *how many* such triples (i, j, k, l) there are, since each combination of $x^{A[i]}, x^{A[j]}, x^{A[k]}, x^{A[l]}$ contributes one x^t term at the end (although this is not relevant for this problem).

Runtime Analysis: Constructing $p(x)$ clearly takes $O(n)$ time. Since $0 \leq A[i] \leq 170n$, $p(x)$ is a polynomial of degree at most $170n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. 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 $0 \ldots 170n$. 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.

Comment: The technique used to solve this problem (namely encoding information in the coefficients of a polynomial and then manipulating the polynomial in some way) is closely related to the technique of generating functions, which are used in combinatorics to do many cool things such as giving a standard way to find a closed form solution to certain types of recurrence relations (for instance the recurrence relation defining the Fibonacci numbers).

5. **(★★★★ level)** **Local Maxima** Consider an $n \times n$ matrix M with distinct integer entries. Call an entry M_{ij} a *local maximum* if it is greater than all of its neighbors. More precisely, M_{ij} is a local maximum if for all a,b with $|i-a| \leq 1$ and $|j-b| \leq 1$, $M_{ij} \geq M_{ab}$. In an array A, say that A_i is a *local maximum* of A is $A_i \geq A_{i+1}$ and $A_i \geq A_{i-1}$. Note that if i is the last element in A, then $A_i \geq A_{i-1}$ is sufficient for A_i to be local maximum, and similarly if i is the first element, then $A_i \geq A_{i+1}$ is sufficient.

Suppose that M is guaranteed to have exactly one local maximum and that every column of M, when viewed as an array, contains exactly one local maximum. Show that the local maximum of M can be found in $O(\log^2 n)$ time.

(Please give a four part solution for this problem.)

Solution:

Main idea:

Do binary search on both rows and columns. Pick the middle column and use binary search to find the maximum value in this column. If this element is a local maximum, we are done. Otherwise, there is some adjacent element that is larger. If there are larger elements in both adjacent columns, then this would imply the existence of two local maxima. Therefore, only one adjacent column contains a larger element. Recur on that submatrix.

Pseudocode:

```
procedure MatrixMax(Matrix $M$)
    $m$ = number of rows of $M$
    $n$ = number of columns of $M$
    if $n = 1$ then Output ArrayMax($M$ as an array)
    $A$ = column $\lfloor n/2 \rfloor - 1$ as an array
    $e$ = ArrayMax(A)
    if $e$ is a local maximum then Output $e$
    else if $e$ is adjacent to some larger element $f$ in column $\lfloor n/2 \rfloor$ then Output MatrixMax(the submatrix
of $M$ of higher index columns than $A$)
    elseOutput MatrixMax(the submatrix of $M$ of lower index columns than $A$)
procedure ArrayMax(Array $A$)
    if $|A| = 1$ then Output $A[0]$
    $a = A[\lfloor |A|/2 \rfloor - 1]$
    if $a$ is to the left of a larger element $b$ then Output ArrayMax(the subarray of $A$ containing $b$ and all
elements to the right of it)
    else if $a$ is to the right of a larger element $b$ then Output ArrayMax(the subarray of $A$ containing $b$
and all elements to the left of it)
    elseOutput $a$
```

Runtime analysis:

This algorithm takes $O(\log^2 n)$ time. This is because ArrayMax(A) is just doing binary search and takes $O(\log |A|) \leq O(\log n)$ time. Let $T(m,n)$ be the runtime of MatrixMax. Notice that $T(m,n) \leq T(m,n/2) + O(\log n)$. By the Master Theorem, the total runtime is $O(\log^2 n)$ since $m = n$ for the input matrix.

Proof of correctness:

First, we show inductively that ArrayMax(A) outputs the maximum entry of A. Since A is a column of M, A has only one local maximum. If $|A| = 1$ then A has only one element. If $|A| > 1$, let c be its maximum. Since c is the only local maximum, the middle element a must be less than it. If a is to the left of c, then the first ArrayMax call will be executed; otherwise the second one will be executed. If $c = a$, then the algorithm will output c. By induction, both ArrayMax calls will return c, so ArrayMax(A) will return c.

Now, we show that MatrixMax(M) outputs the maximum value of M. Let M_L and M_R be the submatrices to the left and right of A respectively. We will show that at least one of $\{M_L, M_R\}$ only contains elements that are less than e. Suppose not, i.e. that both M_L and M_R contain elements e_L and e_R that are greater than e. Define f_L and f_R as follows. Let $f_L = e_L$ and $f_R = e_R$. Repeatedly replace f_L and f_R with any larger neighbors as long as they are not local maxima. Since f_L and f_R are local maxima, $f_L = f_R$. But this means that some intermediate value of f_L or f_R must lie in A, since A separates M_L from M_R. This contradicts the fact that all entries of A are less than e. This shows that at least one of M_L and M_R only contain elements less than e.

If e is not a local maximum, then exactly one of M_L or M_R contains the local maximum. By the previous paragraph, the correct M_x will contain entries adjacent to e that are greater than e. Therefore, the algorithm will pick the right M_x to recur on.

6. (★★★★★ level) DNA Sequence Alignment

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

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

(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 $t[\ell \dots \ell+m-1]$ 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., $s_0, s_1, \dots, s_{m-1}$ agrees with $t_i, t_{i+1}, t_{i+2}, \dots, t_{i+m-1}$ 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 $t'[4\ell \dots 4\ell+4m-1]$ with at most $2k$ errors. This match in turn corresponds to a match between s and $t[\ell \dots \ell+m-1]$ 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.

7. (??? level) (Optional) Redemption for Homework 1

Submit your *redemption file* for Homework 1 on Gradescope. If you looked at the solutions and took notes on what you learned or what you got wrong, include them in the redemption file.

Solution: N/A