CS 170
Fall 2017

Efficient Algorithms and Intractable Problems
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

Keep doubling an index i starting at one until $f(i) < 0$. Once a nonzero element is reached, then you know that $n = \min\{i \in \mathbb{N} : f(i) < 0\} \in (\frac{i}{2}, i]$. Then, run a slightly modified binary search on that interval.

Pseudocode

```
minIndex(f)
  i = 1
  while f(i) < 0
    i* = 2
  bSearch(i)

bSearch(i)
  middle = (i/2 + i)/2
  if f(middle) < 0 AND f(middle-1) ≥ 0
    return middle
  else if f(middle) < 0
    return bSearch(middle)
  else
    return bSearch(middle + (i - middle)/2)
```

Proof of Correctness

If we stop the doubling of i due to $f(i) < 0$, then we know for sure that $f(\frac{i}{2}) \geq 0$. Then, it is clear that $\min\{i \in \mathbb{N} : f(i) < 0\} \in (\frac{i}{2}, i]$. We now reduce the problem to finding the first nonzero element in a "decreasing array" that has the elements $[f(\frac{i}{2}) + 1..., f(i-3), f(i-2), f(i-1), f(i)]$. Note that $i = 2^k, k \in \mathbb{N}$. Then, a simple modification of binary search where the index limits are shifted, and the test for going to the left or right comes from whether or not $f(\text{middle}) < 0$ completes the algorithm.

Running Time

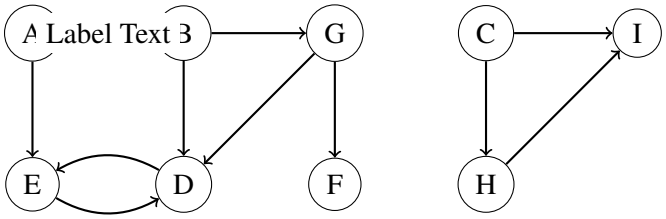
$O(\log(n))$

Justification

It will take at most $\lceil \log_2(n) \rceil$ doublings in order for i to surpass n, the assumed first index where $f(n) < 0$. Then, running binary search on an "array"(function evaluations are constant, so we can treat it like an array in terms of runtime) takes $O(\log(2^{\lceil \log_2(n) \rceil} - 2^{\lceil \log_2(n) \rceil - 1})) \in O(\log(n))$. The latter is the same as the former, so the overall runtime is simply $O(\log(n))$.

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.

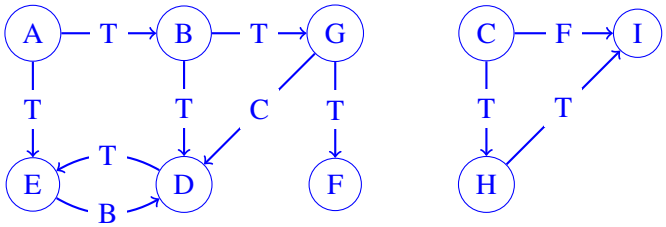


(a) Run DFS at node A, trying to visit nodes alphabetically (e.g. given a choice between nodes D and F, visit D first).

- List the nodes in the order you visit them (so each node should appear in the ordering exactly once).
  Solution: A,B,D,G,F,E,C,H,I
- List each node with its pre- and post-number. The numbering starts from 1 and ends at 18.
  Solution:

	A	B	C	D	E	F	G	H	I
Pre	1	2	13	3	4	8	7	14	15
Post	12	11	18	6	5	9	10	17	16

- Label each edge as **T**ree, **B**ack, **F**orward or **C**ross.
  Solution:



(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:** Saw we have $|V|$ vertices. Then to maximize $|E|$ we should make our graph complete. That is, $|V|$ is connected to every other vertex possible. Then, we have that the first vertex can connect to $|V|-1$ other vertices, and the second vertex can connect to $|V|-2$ vertices, and so forth. So the total number of edges is

$$\sum_{i=0}^{|V|} i \in O(|V|^2)$$

□

(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:

Proof by induction on $|V|$.
Base case: $|V| = 1 \Rightarrow \sum d_i = 0$.
Inductive Hypothesis: If a graph with $|V|$ vertices has $\sum d_i$ even, then any graph with $|V|+1$ must also have $\sum d_i$ even.
Inductive Step: To avoid build-up error, instead of adding a vertex to achieve a graph of $|V|+1$ vertices, we assume that we have any arbitrary graph with $|V|+1$ vertices. Now, remove any vertex, and the

inductive hypothesis now applies to the new graph. This is valid because we do not require our graph to be connected. Now, after replacing the node that we removed, we can see that any edge that was nullified due to the removal that comes back adds 2 to $\sum d_i$. Thus, any amount of edges nullified that are restored can only change $\sum d_i$ by a multiple of 2, maintaining the evenness. □

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

Looking at the middle element of the array, we can determine whether or not there is a "peak" in the left, right, or both of the left and right subarrays, or, if the middle element is a peak itself. This splits the problem in half.

Pseudocode

findPeak(houses)
middle $= \frac{\text{size(houses)}}{2}$
  if size(houses) $==$ 1
    return houses[0]
  if size(houses) $==$ 2
    return larger of two
  if houses(middle) $<$ $houses(middle+1)$
    return findPeak(houses[middle:])
  else if houses(middle) $<$ $houses(middle-1)$
    return findPeak(houses[:middle])
  else return houses[middle]

Proof of Correctness

If houses(middle) $<$ $houses(middle+1)$, then there are two cases:

1. The second half of the houses array is strictly increasing. Then the last house must be a peak.
2. The second half of the houses array is not strictly increasing, meaning that somewhere, there must be at least one "turning point" where the houses start to decrease. The "turning point" must then be a peak.

If houses(middle) $>$ $houses(middle+1)$, then there are two cases:

1. The first half of the houses array is strictly decreasing starting from the left. Then the first house must be a peak.
2. The second half of the houses array is not strictly decreasing, meaning that somewhere, there must be at least one "turning point" where the houses start to increase. The "turning point" must then be a peak.

If neither of the previous two cases are true, then the middle must be a peak. This proves that cutting the problem in half in this way is valid.

Running Time

$O(log(n))$

Justification

At each stage, we are eliminating one half of the array, halving the problem size.

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. 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

Represent array A as a degree at most $170n$ polynomial.

$$p(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_{170n}x^{170n}, a_0 = 1$$

Have the coefficient a_i is either 1 or 0 depending on if that element exists in A. Then, we can simply take this polynomial to the fourth power, and check to see if the result has a non-zero coefficient for the n^t term, using FFT to do the heavy lifting of multiplying the polynomials.

Pseudocode

exactChange(DENOMS, COST)
  denomTuple = Coefficient representation of degree at most 170n polynomial of the form $\sum_{i=0}^{170n} a_i x^{A[i]}$
  valueDenomTuple = FFT evaluation performed on denomTuple
  Take the fourth power of each element in valueDenomTuple
  resultTuple = FFT interpolation on valueDenomTuple
  if (resultTuple has nonzero coefficient for the x^t term)
    return true
  else
    return false

Proof of Correctness

After the first squaring of $p(x)$, we have all the possible values that are possible with two coins of any two denominations. This is because of the nature of polynomial multiplication. Because the x^c term represents a coin value of c, when multiplied with say x^d, we have $x^c x^d = x^{c+d}$, which gives us a term that corresponds to a value $c+d$ coin, as desired. Each term in the polynomial is multiplied with each of the other terms in the polynomial, including itself. The result must then be all the possible sums of two coins from the denomination list. Because we require four coins, we take the fourth power of $p(x)$.

Running Time

$O(n\log(n))$

Justification

Converting A to the coefficient representation of $p(x)$ takes n time. Using FFT to evaluate a degree $170n$ polynomial takes time $170n\log(170n)$. However, we want to evaluate $p(x)$ at $4(170n)+1$ points, because we know that our result is of at most degree $680n$. This takes $680n\log(680n)$. Then, determining whether our desired term is in the result takes n time. $O(2n+680n\log(680n)) \in O(n\log(n))$.

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

Notice that if there is a guarantee that there is only one local maximum for every column of M, then that local maximum is also the global maximum for that column. After finding the local maximum, we need to check whether this is a peak. If that is not the case we can split the problem by eliminating the half of the array.

Pseudocode

```
findPeak(M)
  if M only has one column
    return local max of that column
  i = width(M)/2
  ithCol = ith column of M
  localMaxIndex = index of local maximum in ithCol
  if (M[localMaxIndex][i] < M[localMaxIndex][i-1])
    return findPeak(left half of M)
  if (M[localMaxIndex][i] < M[localMaxIndex][i+1])
    return findPeak(right half of M)
  else
    return M[localMaxIndex][i]
```

Proof of Correctness

We can cut the problem in half once we know the relationship between a columns local max and its left and right neighbors. Suppose we have a local max (i, j). Then suppose that $(i, j+1) > (i, j)$. Then, we know that the peak must be to the right of column j. Then, if $(i, j+1)$ is not the local maximum of column $j+1$, we go to the local maximum, say $(i', j+1)$. Either this is the peak, or we look at column $j+2$. Eventually, we will either have to see a peak, or reach the end. Because the point before the end $(i'', n-1)$ must have been less than (i'', n), either (i'', n) is the peak, or the local max of column n must be the peak. The latter follows because the peak of column n must be larger than all the elements in column $n-1$, because the local max of column $n-1$, $(i'', n-1)$ is less than (i'', n). Because our peak is unique, and we showed that it must be in the right half, we can disregard the left half. A symmetric argument for the left side where $(i, j+1) < (i, j)$ must follow.

Running Time

$O((\log(n))^2)$

Justification

It takes $\log(n)$ time to find the local max of a column. We must do this every time we cut the matrix in half. The matrix can be cut in half in the worst case $\log(n)$ times. Thus the algorithm is $\in O((\log(n))^2)$.

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:

Pseudocode

```
stringMatch(s,t,k)
  result = []
  tCounter = 0
  while(tCounter < len(t) − len(s))
    if length s array starting at tCounter has ≤ k errors.
      append tCounter to result
    tCounter+ = 1
```

(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.

(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).

(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.

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.