Solutions to Homework 11

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't do last week's homework, choose a problem from it that looks challenging to you, and in a few sentences, explain the key ideas behind its solution in your own words.

Solution: My solutions for 5.b and 5.c were as follows:

> **Solution:** Given S and an element $a \in S$, compute
>
> $$r = |\{x \in S : x < a\}| + 1.$$
>
> Then a is a near median if and only if
>
> $$\frac{2n}{5} + 1 \leq r \leq \frac{3n}{5}.$$
>
> This process takes $O(n)$ time.

> **Solution:** Repeat the following process 21 times:
> ⟨omitted for brevity⟩
> If no near median is found after 21 iterations, compute the median of S using a deterministic linear-time selection algorithm and output it.
>
> The probability of not finding a near median in one iteration is at most $4/5$. Thus, the probability of not finding a near median in 21 iterations is at most
>
> $$\left(\frac{4}{5}\right)^{21} \leq \frac{1}{100}.$$
>
> The expected running time of the algorithm is $O(n)$ because each iteration takes $O(n)$ time, and the expected number of iterations is constant.

Both of these solutions are correct, however, I did not include a correctness proof for either of them. 5.b. requires a correctness proof for it's algorithm, and 5.c. requires a correctness proof for the 99% success probability requested by the question.

Solutions to Homework 11

2. **Short-answer potpourri.**

(4 pts) (a) Suppose that Alice's bit string is $x = 101111000$ and Bob's bit string is $y = 111100110$. State, with justification, **all** the "bad" primes p for which the fingerprinting protocol from class will (wrongly) *accept*, even though $x \neq y$.

Hint: Recall that x and y are interpreted as integers written in base two, with digits written from most to least significant.

Solution:

$$x = 101111000_2 = 376, \quad y = 111100110_2 = 486.$$

$$x - y = -110 = -(2 \cdot 5 \cdot 11).$$

Bad primes divide $x - y$,

$$p \in \{2, 5, 11\}.$$

For each p,

$$x \equiv y \pmod{p}.$$

(4 pts) (b) Recall the fast "spot checking" algorithm for verifying matrix multiplication from lecture. Consider the matrices

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 2 & 1 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 3 & 2 & 4 \\ 1 & 3 & 3 \\ 2 & 5 & 1 \end{bmatrix}, \quad A \cdot B = \begin{bmatrix} 11 & 23 & 13 \\ 13 & 17 & 19 \\ 13 & 22 & 14 \end{bmatrix}, \quad C = \begin{bmatrix} 13 & 21 & 18 \\ 13 & 17 & 19 \\ 14 & 21 & 10 \end{bmatrix},$$

and notice that $AB \neq C$. Give, with justification, **all** the binary vectors for which the algorithm (wrongly) accepts.

Solution: $A(Br) \stackrel{?}{=} Cr$:

$$\begin{aligned} 11r_1 + 23r_2 + 13r_3 &= 13r_1 + 21r_2 + 18r_3, \\ 13r_1 + 17r_2 + 19r_3 &= 13r_1 + 17r_2 + 19r_3, \\ 13r_1 + 22r_2 + 14r_3 &= 14r_1 + 21r_2 + 10r_3. \end{aligned}$$

So,

$$\begin{aligned} -2r_1 + 2r_2 - 5r_3 &= 0, \\ 0 &= 0, \\ -r_1 + r_2 + 4r_3 &= 0. \end{aligned}$$

Thus,

$$r_1 = r_2 + 4r_3.$$

In order for this to remain true, because we are limited to $\{0, 1\}$, we must have $r_3 = 0$ and thus $r_1 = r_2 + 0 = r_2$. So, binary vectors for which the algorithm wrongly accepts are:

$$r = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad r = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}.$$

(5 pts) (c) Three weeks ago, Anonymous Beaker posted a long, unexplained string of decimal digits on Piazza. You didn't think much of it at the time, but something about it stuck with you.

Now that you've learned about cryptography, it clicks: this must be a message, hidden in plain sight. A follow-up reply hints that the message format as follows:

- The digits represent ASCII codes written in pairs of decimal digits (e.g., `65` → A, `38` → &, `87` → W, etc.).
- The message is encrypted using a Caesar cipher, but it's applied to each *individual digit*, not the ASCII characters or pairs. (e.g., with key $k = 5$, "A&W" (`65 38 87`) becomes 108332).

The ciphertext is:

400644095036345936514356093209515259423656091750325118

Recover the original message. You may use a computer or the ASCII code to string converter. Include an explanation of what you did and clearly state the secret key.

Hint: First determine all the possible secret keys. Then, perform a brute-force attack.

Solution: Possible keys are $k \in \{0, \ldots, 9\}$.

For each possible key k:

For each number in the ciphertext, compute new digit $d_i = (c_i - k) \bmod 10$. Then, group the digits into pairs, and convert each pair to its ASCII character representation.

Key 7 yields:

I'M SECRETLY A TURKEY (SAT).

or,

73 39 77 32 83 69 67 82 69 84 76 89 32 65 32 84 85 82 75 69 89 32 40 83 65 84 41

(5 pts) (d) Recall the following extended Euclidean algorithm from Homework 1.

Input: integers $x \geq y \geq 0$, not both zero
Output: a triple (g, a, b) of integers where $g = \gcd(x, y)$ and $ax + by = g$

```
1: function ExtendedEuclid(x, y)
2:     if y = 0 then
3:         return (x, 1, 0) // Base case: 1x + 0y = gcd(x, y) = x
4:     else
5:         Write x = qy + r for an integer q, where 0 ≤ r < y
6:         (g, a', b') ← ExtendedEuclid(y, r)
7:         a ← b'
8:         b ← a' − b'q
9:         return (g, a, b)
```

Show that if $\gcd(x, y) = 1$ (in other words, x and y are coprime), then $ax \equiv 1 \pmod{y}$ and $by \equiv 1 \pmod{x}$. Then, calculate $32^{-1} \bmod 263$ and $263^{-1} \bmod 32$ using ExtendedEuclid. Show you work by filling in the table below:

input		division		rec ans		output	
x	y	q	r	a'	b'	a	b
263	32	8	7	2	−9	−9	74
32	7	4	4	−1	2	2	−9
7	4	1	3	1	−1	−1	2
4	3	1	1	0	1	1	−1
3	1	3	0	1	0	0	1
1	0					1	0

Solution: If $\gcd(x, y) = 1$, then $ax + by = 1$ for some integers a and b.

$$ax + by \equiv 1 \pmod{y} \implies ax \equiv 1 \pmod{y},$$

$$ax + by \equiv 1 \pmod{x} \implies by \equiv 1 \pmod{x}.$$

$$32^{-1} \bmod 263 = 74, \qquad 263^{-1} \bmod 32 = -9 \equiv 23 \pmod{32}.$$

(5 pts) (e) Recall the following (iterative version of) fast modular exponentiation (repeated squaring) algorithm from Homework 3.

Input: positive integers a, b, and modulo $m \geq 2$
Output: $a^b \bmod m$

```
1: function RepeatSquare(a, b, m)
2:     r ← ⌈log_2 b⌉
3:     P ← Initialize an empty table of size r // To store P[i] = a^{2^i} mod m
4:     P[0] ← a mod m
5:     for i = 1, ..., r − 1 do
6:         P[i] ← (P[i − 1] · P[i − 1]) mod m
7:     E ← 1
8:     for i = 0, ..., r − 1 do
9:         if ⌊b/2^i⌋ is odd then // The i-th bit of b in binary is 1
10:            E ← (E · P[i]) mod m
11:    return E
```

Compute $6^{25} \bmod 22$ using RepeatSquare. Show your work by first filling out the table P below (using lines 3-6). Then, list all the $P[i]$'s that you included in your product to calculate $6^{25} \bmod 22$ (using lines 8-10) in your calculation leading to the final answer.

i	2^i	$P[i]$
0	1	6
1	2	14
2	4	20
3	8	4
4	16	16

Solution: $25 = 11001_2$; thus include $P[0]$, $P[3]$, $P[4]$.

$$6^{25} \bmod 22 \;=\; (P[0] \cdot P[3] \cdot P[4]) \bmod 22 \;=\; (6 \cdot 4 \cdot 16) \bmod 22 \;=\; 10.$$

3. **Trading space for time in skip lists.**

Recall that a skip list is essentially a linked list with multiple levels. To add an item, we first find the appropriate place to insert it in the first level, and do so. Then, we run the following loop: flip a fair coin; if it comes up heads, insert the item at the next-higher level and repeat; if it comes up tails, stop.

A skip list keeps duplicate copies of items, which consumes extra memory. In this problem, we will obtain a trade-off between the amount of memory used and the running time of the 'find' operation, by adjusting the probability with which we "promote" items to higher levels. That is, instead of using a fair coin, we use one that comes up heads with some probability $p \in (0, 1)$.

Throughout this problem, assume that there are n distinct items in the data structure.

(4 pts) (a) For any particular item in the data structure, derive the expected number of copies of it that appear in the structure, in terms of p.

Solution:

$$\sum_{k=0}^{\infty} p^k = \frac{1}{1-p}.$$

p^k is the probability of the item appearing at level k. The expected number of copies of an item is the sum of the probabilities of the item appearing at each level, starting with 1 (100%) for the base level.

(4 pts) (b) Derive the expected size of the structure, i.e., the total number of copies of all the items, in terms of n and p. Also briefly describe how this behaves as p decreases and approaches 0.

Solution:

$$E[\text{size}] = n \sum_{k=0}^{\infty} p^k = \frac{n}{1-p}.$$

Thus, as $p \to 0$, $E[\text{size}] \to n$.

(5 pts) (c) Derive the expected number of items on level i (where $i = 0$ corresponds to the first level), in terms of n, p, and i.

Hint: Determine the probability that a particular item appears on level i.

Solution:

$$E[\#\text{items on level } i] = n\,p^i.$$

(7 pts) (d) Let Z be random variable denoting the total number of occupied levels in the structure. Show that

$$\mathbb{E}[Z] \leq \frac{1}{1-p} + \frac{\log n}{\log(1/p)}.$$

Also briefly describe how this bound behaves as p decreases and approaches 0.

Hint: Define an indicator random variable that indicates whether level i has *at least one* item, and use the previous part to analyze its expectation.

Solution:

$$\mathbb{E}[Z] = \sum_{i=0}^{\infty} \Pr[\text{level } i \text{ occupied}] = \sum_{i=0}^{\infty} \big(1 - (1-p^i)^n\big)$$

$1 - p^i$ is the probability that a particular item is NOT in level i. So, $(1-p^i)^n$ is the probability that any item is NOT in level i, or, the probability that level i is unoccupied. Thus, $1 - (1-p^i)^n$ is the probability that level i is occupied. Summing this for all levels gives the expected number of occupied levels.

$$\sum_{i=0}^{\infty} \big(1 - (1-p^i)^n\big) \leq \sum_{i=0}^{\lfloor \frac{\ln n}{\ln(1/p)} \rfloor} 1 + \sum_{i=\lfloor \frac{\ln n}{\ln(1/p)} \rfloor + 1}^{\infty} n\, p^i \leq \frac{\ln n}{\ln(1/p)} + \frac{1}{1-p}.$$

As $p \to 0$, $\frac{\ln n}{\ln(1/p)} \to 0$ and $\frac{1}{1-p} \to 1$.

(7 pts) (e) Consider any particular level i and an item y that appears in that level. Derive a fairly tight upper bound (in terms of any of the relevant quantities) on the expected number of items preceding y **on level** i that are accessed when running FIND(y). Also briefly describe how this bound behaves as p decreases and approaches 0.

Hint: For each item on level i that precedes y, define and analyze an indicator random variable for whether that item is accessed.

Solution: Let the search on level i arrive at the unique node x whose key is the largest key strictly less than y. This node is either the leftmost sentinel or came from the level $i+1$. The algorithm repeatedly follows the "next" pointer on level i until y is reached. Each of the items that lie between x and y must already be present on level $i-1$. Every item on level i was promoted independently with probability p. Therefore, reading left-to-right, the index j (distance from x) of the *first* item in this block whose promotion succeeds is distributed with

$$\Pr[j] = p(1-p)^{j-1} \qquad (j \geq 1),$$

The algorithm visits exactly the first J items in that block on level i (the predecessors of y and y itself), so

$$\mathbb{E}[J] = \sum_{j=1}^{\infty} jp(1-p)^{j-1} = \frac{1}{p}.$$

So, the expected number of level-i predecessors of y examined by FIND(y) is at most $1/p$. As $p \to 0$, this bound grows like $1/p$ and then diverges.

4. **Spot-checking polynomials.**

Suppose we are given two polynomials, each presented using the variable x, constants, additions, multiplications, and powers. For example, one polynomial might be presented as $p_1(x) = ((x-1)^2+2)^2$, and the other as $p_2(x) = (x-1)^4 + 4x(x-2) + 8$. Observe that these two polynomials are *identical*, i.e., $p_1(x^*) = p_2(x^*)$ for *all* x^*. Equivalently, $p(x) = p_1(x) - p_2(x)$ is the *zero* polynomial: $p(x^*) = 0$ for all x^*.

As another example, $q_1(x) = (x+1)^2$ and $q_2(x) = x^2 + 1^2$ are *not* identical: $q_1(x) - q_2(x) = 2x$, which is not the zero polynomial.

This suggests the natural "zero-testing" problem, of determining whether a given polynomial is the zero polynomial. This problem has many applications, like verifying polynomial multiplication—along with some graph problems that appear to have nothing to do with polynomials at all!

One way to solve the zero-testing problem is to simplify the given polynomial into "standard" form, as a sum of monomials, and test whether the result is zero (i.e., whether all the monomials cancel out). However, this can be complicated to implement and potentially slow, depending on how we do polynomial multiplication. For example, the naïve multiplication algorithm takes $\Theta(n^2)$ time to multiply two polynomials each having n monomials.

In this problem, you will consider a simple and fast randomized "spot checking" algorithm for zero testing, which relies merely on *evaluating* the polynomial. Assume that we have a subroutine that evaluates a given polynomial at a given value; this can be implemented straightforwardly by substituting the given value for x.

(5 pts) (a) Recall that the *degree* of a nonzero polynomial is the largest exponent that appears in its nonzero monomials; the degree of the zero polynomial is undefined. For example, the degree of $2x^3 + 8x^7 + 1x^6$ is 7, and the degree of 376 is 0.

Also recall that every nonzero degree-d polynomial $p(x)$ has at most d (complex) *roots*, i.e., values x^* such that $p(x^*) = 0$. So, this also holds when we restrict to integer roots.

All this suggests the following *incomplete* spot-checking algorithm for zero testing:

Input: a polynomial that is either nonzero and of degree d, or is the zero polynomial
1: **function** ZeroTest($p(x)$)
2: choose an integer $r \in \{1, 2, \ldots, \_\_\_\_\_\}$ uniformly at random
3: **if** $p(r) = 0$ **then** _____
4: **else** _____

State, with justification, how to fill in the three blanks above so that:

- If $p(x)$ is the zero polynomial, then ZeroTest($p(x)$) accepts with certainty (i.e., with probability 1).
- If $p(x)$ is nonzero, then $\Pr[\text{ZeroTest}(p(x)) \text{ rejects}] \geq 1/2$.

Solution: '$2d$', 'accept', and 'reject'.

A nonzero degree-d polynomial has at most d roots, so choosing r uniformly from $1, \ldots, 2d$ selects a root with probability at most $1/2$. Thus, the test rejects a nonzero polynomial with probability at least $1/2$, while the zero polynomial is always accepted.

(4 pts) (b) Accepting the zero polynomial with certainty is great, but rejecting a nonzero polynomial with probability only 1/2 is not so great. Describe, with brief justification, *two different ways* of modifying the algorithm so that the probability of rejecting a nonzero polynomial improves to at least 99% (and the zero polynomial is still accepted with certainty).

Solution:

1. Repeat the basic test independently k times. Reject if any evaluation is nonzero; otherwise accept. For a nonzero polynomial the acceptance probability is at most $(1/2)^k \leq 0.01$, so the rejection probability is at least 99% for, say, $k = 7$.
2. Keep a single evaluation but choose r uniformly from $1, \ldots, 100d$. A nonzero degree-d polynomial then has acceptance probability of at most $d/(100d) = 1\%$, yielding rejection probability of at least 99%.

Both modifications accept the zero polynomial with certainty.

5. **Pattern-matching fingerprinting.**

In computational linguistics, a common task is to determine whether a short syllable appears as a contiguous part of a word. This problem can be formalized as identifying whether a smaller string Y is a substring of a larger string X.

Assume for simplicity that the strings are encoded in binary. Suppose n is the length of X and m is the length of Y. A simple brute force solution is by comparing Y with every possible substring of x, which takes $O(mn)$ time. In this problem, you will explore a randomized algorithm that has a faster running time, with a slight sacrifice in accuracy.

Consider the following randomized algorithm:

```
1: function PatternMatch(X, Y, k)
2:     Pick a random prime p in the range {1, ..., k}.
3:     for j = 1, ..., n − m + 1 do
4:         X_j ← x_j ··· x_{j+m−1} // Substring of length m starting at x_j
5:         if X_j mod p = Y mod p then
6:             accept
7:     reject
```

(2 pts) (a) Briefly explain why if Y is a substring of X, the algorithm will always accept.

Solution: If Y is a substring of X, some j satisfies $X_j = Y$, which implies that $X_j \bmod p = Y \bmod p$ for every prime p. The algorithm reaches line 5 and accepts.

(3 pts) (b) Suppose Y is not a substring of X. Derive an upper bound on the probability of a false positive in a single iteration of the algorithm. You may use the Prime Number Theorem which says the number of primes less than k is $\pi(k)$, where $\pi(k) \approx k/\ln k$.

Solution: Let $D_j = X_j - Y \neq 0$ for each j. A false positive arises when $p \mid D_j$ for some j. Each $D_j < 2^m$ contains at most m distinct prime divisors, so at most m primes in $\{1, \ldots, k\}$ divide a fixed D_j. The number of primes in that range is $\pi(k) \geq k/\ln k$. Applying the union bound over the $n - m + 1$ substrings gives

$$\Pr[\text{false positive}] \leq \frac{(n-m+1)m}{\pi(k)} \leq \frac{(n-m+1)m \ln k}{k}.$$

(3 pts) (c) In this part, you'll prove a result you need for the next part.

Let $h_j = X_j \bmod p$. Give a formula for h_j in terms of x_{j-1} (the leftmost bit in X_{j-1}), x_{j+m-1} (the bit right after X_{j-1}), and h_{j-1}.

Solution: Let $\beta = 2^{m-1} \bmod p$, then:

$$h_j = ((h_{j-1} - x_{j-1}\beta) \cdot 2 + x_{j+m-1}) \bmod p.$$

(5 pts) (d) It turns out that the current implementation of PatternMatch still runs in $O(mn)$ time. Using the result in Part c, modify the algorithm so that it uses at most $O(n \log k)$ numerical operations (addition, subtraction, multiplication, modulo operation, *bit* comparison[1]). Justify your answer.

Hint: You may want to precompute some values and reuse them in the for-loop.

Solution:

```
1:  function PatternMatchFast(X, Y, k)
2:      pick a random prime p ∈ {1, ..., k}
3:      β ← 2^(m−1) mod p
4:      h_Y ← Y mod p
5:      h ← X_1 mod p
6:      for j = 1 to n − m + 1 do
7:          if h = h_Y then
8:              if X_j = Y then
9:                  accept
10:         if j < n − m + 1 then
11:             h ← ((h − x_j β) · 2 + x_{j+m}) mod p
12:     reject
```

Lines 2-5 use $O(m + \log k)$ operations. The loop executes $n - m + 1 = O(n)$ iterations, each performing $O(1)$ modular operations. Each potential match triggers at most one $O(m)$ bit comparison, whose expected count is bounded by the probability in part (b); for $k = \Theta(nm \ln k)$ these comparisons contribute $O(1)$ expected cost per iteration. Hence the total number of numerical operations is $O(n \log k)$.

[1] This means comparing two bit strings of length ℓ uses $O(\ell)$ numerical operations.

6. **Diffie–Hellman.**

Vikram and Eric are writing the solutions to Homework 11. Vikram has a draft of the solutions on his laptop and wants to send them confidentially to Eric. Since they have not had a private moment to choose a shared secret key together, they decide to use the Diffie-Hellman protocol to establish one over the public network.

(6 pts) (a) Suppose they use the Diffie–Hellman protocol with the small prime $p = 29$ and generator $g = 2$ for $\mathbb{Z}_p^*$. Vikram chooses secret exponent $a = 12$, and Eric chooses $b = 22$. Derive the values that they send each other and the secret key that each one computes, to conclude that they both compute the same value.

You may use a calculator for basic arithmetic (multiplication and division), but beyond that you should use the efficient algorithms that a computer would employ. Show your work (repeated squaring etc.), but you can omit the details of modular reductions. Keep numbers small by reducing modulo p where appropriate, and use negative number where they help.

Solution: Public values:

$$x = g^a \bmod p = 2^{12} \bmod 29 = 2^8 \cdot 2^4 \bmod 29 = (24)(16) \bmod 29 = 7,$$
$$y = g^b \bmod p = 2^{22} \bmod 29 = 2^{16} \cdot 2^4 \cdot 2^2 \bmod 29 = (25)(16)(4) \bmod 29 = 5.$$

Secret key:

$$K = y^a \bmod p = 5^{12} \bmod 29 = 5^8 \cdot 5^4 \bmod 29 = (24)(16) \bmod 29$$
$$= 7 = 7^{22} \bmod 29 = x^b \bmod p.$$

(6 pts) (b) After realizing that the parameters from the previous part are much too small (since they are breakable by hand), Vikram and Eric decide to use the Diffie–Hellman protocol with a different, huge prime p and generator g of $\mathbb{Z}_p^*$. As specified by the protocol, Vikram chooses secret $a \in \mathbb{Z}_p^*$ uniformly at random, and sends $x = g^a \bmod p$ to Eric. Similarly, Eric chooses secret $b \in \mathbb{Z}_p^*$ uniformly at random, and sends $y = g^b \bmod p$ to Vikram.

Unbeknownst to them, Lambda in the middle of their network, and is able to intercept *and undetectably replace* what they send over the network with other values of his choice. (In class we considered only an *eavesdropper* Eve who can merely read all the network traffic; this Lambda is more powerful.)

Specifically, Lambda chooses a secret $c \in \mathbb{Z}_p^*$ himself, and sends $z = g^c \bmod p$ to both Vikram and Eric in place of their messages to each other. That is, Vikram thinks that Lambda's message came from Eric, and Eric thinks that Lambda's message came from Vikram.

Give an expression for the key that Vikram K_V computes, and the key K_E that Eric computes, in terms of the secret exponents a, b, c and the public values p, g.

Solution:

$$K_V = (g^c)^a \bmod p = g^{ac} \bmod p, \qquad K_E = (g^c)^b \bmod p = g^{bc} \bmod p.$$

(6 pts) (c) Vikram and Eric work on the homework solutions by encrypting their messages using the keys they computed in the previous part (which they believe are the same key). Lambda knows the encryption scheme that Vikram and Eric are using. Describe how Lambda can decrypt all of Vikram and Eric's communications without being detected, i.e., so that they believe they are communicating confidentially, and when they decrypt the ciphertexts they receive, they get exactly the messages that were intended for them.

A valid solution should show the following:

- how Lambda can successfully decrypt all of the ciphertexts that Vikram and Eric send to each other;
- how Lambda can avoid detection by replacing the ciphertexts that Vikram and Eric send with ones that will yield the intended messages when they decrypt (using what they believe to be their shared key).

Solution:

- Intercept $x = g^a$ and $y = g^b$; send $z = g^c$ to both parties.
- Compute $K_V = g^{ac} \bmod p$ and $K_E = g^{bc} \bmod p$.
- Upon receiving a ciphertext from Vikram, decrypt with K_V; re-encrypt the plaintext with K_E and forward to Eric.
- Upon receiving a ciphertext from Eric, decrypt with K_E; re-encrypt the plaintext with K_V and forward to Vikram.

All messages decrypt to the intended plaintexts, so neither Vikram nor Eric detects the interception.