EECS 376: Foundations of Computer Science **Due 8:00pm, May 16**
University of Michigan, Spring 2024 **Homework 1** **(accepted until 9:59 pm, no credit after)**

This homework has 10 questions, for a total of 100 points and 10 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on Piazza, and we do not guarantee anything about the difficulty of these questions.
We strongly encourage you to typeset your solutions in LaTeX.
If you collaborated with someone, you must state their name(s). You must *write your own solution* for all problems and *may not use any other student's write-up*.

(0 pts) 0. **Before you start; before you submit.**

(a) Carefully read Handout 1 before starting this assignment, and apply it to the solutions you submit.

(b) If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

1. **Practice with asymptotics ("big-Oh, big-Omega, big-Theta").**

For the following pairs of functions, state with justification whether or not each of the following hold: $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$.

You can find the definitions of asymptotic notations (O, Ω, Θ) in Handout 0.

(5 pts) (a) $f(n) = n^3 + 2n + 8$, $g(n) = 4n^3$.

Solution: By noticing that the dominant terms of f and g, ignoring constant factors, are both n^3, we can conjecture that $f(n) = \Theta(g(n))$. This is indeed the case, which we formally prove by showing both required conditions.

To prove that $f(n) = \Omega(g(n))$, we can simply take constants $c = \frac{1}{4}$ and $n_0 = 1$; it is immediate that $f(n) \geq \frac{1}{4} \cdot g(n)$ for all $n \geq 1$.

To prove that $f(n) = O(g(n))$, we can take constants $c = 1$ and $n_0 = 2$. Combining $2n \leq n^3$ and $8 \leq n^3$ for $n \geq 2$, we have that $n^3 + 2n + 8 \leq 4n^3$, i.e., $f(n) \leq g(n)$.

Alternatively, we could have used limits to very concisely prove both required conditions. Applying L'Hôpital's Rule three times, we get that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'''(n)}{g'''(n)} = \lim_{n\to\infty} \frac{6}{24} = \frac{1}{4},$$

which is finite (implying that $f(n) = O(g(n))$) and nonzero (implying that $f(n) = \Omega(g(n))$), as needed.

More generally, if $f(n), g(n)$ are polynomials in n of the same (finite) degree, then $f(n) = \Theta(g(n))$. Try to prove this!

(5 pts) (b) $f(n) = (2 + (-1)^n)n^2 + 1$, $g(n) = n^2$.

Solution: By noticing that the dominant terms of f and g, ignoring constant factors, are both n^2, we can conjecture that $f(n) = \Theta(g(n))$. This is indeed the case, which we formally prove by showing both required conditions.

To prove that $f(n) = \Omega(g(n))$, we can simply take constants $c = n_0 = 1$; it is immediate that $f(n) \geq g(n)$ for all $n \geq 1$.

To prove that $f(n) = O(g(n))$, we can take constants $c = 4$ and $n_0 = 1$. Combining $(-1)^n n^2 \leq n^2$ and $1 \leq n^2$ for $n \geq 1$, we have that $(2 + (-1)^n)n^2 + 1 \leq 4n^2$, i.e., $f(n) \leq 4 \cdot g(n)$.

In this case, we cannot use limits to prove required conditions because the limit $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist.

(5 pts) (c) $f(n) = \log_2(n^{10})$, $g(n) = \log_{10}(n^2)$.

Solution: All three conditions hold, i.e., $f(n) = \Theta(g(n))$. By the property of logarithms, $f(n) = 10 \log_2 n$ and $g(n) = 2 \log_{10} n = 2 \cdot \frac{\log_2 n}{\log_2 10}$. So $f(n)/g(n) = 5 \cdot \log_2 10$ is a nonzero constant, and therefore $f(n) = \Theta(g(n))$.

More generally, if $f(n), g(n)$ are logarithms, with *any* (not necessarily the same) constant bases greater than one, of *any* polynomials in n (not necessarily of the same degree), then $f(n) = \Theta(g(n))$. Try to prove this!

(5 pts) (d) $f(n) = 2^{10n}$, $g(n) = n^{10} \cdot 10^{2n}$.

Solution: Notice that $f(n) = 1024^n$ and $g(n) = n^{10} \cdot 100^n$; we will use these forms in what follows.

We first show that $f(n) = \Omega(g(n))$. We give a proof directly according to the definition, by giving suitable constants c, n_0 and showing that $1024^n \geq c \cdot n^{10} \cdot 100^n$ for all $n \geq n_0$. Letting $c = 1$ for simplicity, what we want to show is equivalent to $n^{10} \leq (1024/100)^n = 10.24^n$ for all $n \geq n_0$. We take $n_0 = 10$. Then we observe that beyond this threshold, the right-hand side grows much more quickly than the left-hand side: as n increases by one, the right-hand side grows multiplicatively by a factor of 10.24, while the left-hand size grows by a factor of *at most* $(11/10)^{10} \leq 10.24$. So, the inequality does indeed hold for all $n \geq n_0$.

Now we show that $f(n) \neq O(g(n))$. We give a proof according to the definition. We need to prove the logical negation of the statement "$1024^n = O(n^{10}100^n)$," which is: *for all* positive constants c, n_0, we have $1024^n < c \cdot n^{10}100^n$ for *some* $n \geq n_0$. So, letting $c, n_0 > 0$ be some arbitrary constants, we need to derive some $n \geq n_0$ *which may depend on* c, n_0, for which $n^{10} < c(1024/100)^n = c \cdot 10.24^n$. The key intuition is that the exponential on the right eventually grows arbitrarily larger than the polynomial on the left, so some large enough n does indeed do the job.

More rigorously, with the help of a computer algebra system like Mathematica, we can determine that the inequality $n^{10} < c10.24^n$ holds whenever n is greater than

some complicated-looking expression we call K_c, which involves various logarithms and esoteric-sounding "product log functions." But the key point is that K_c *depends only on* c, and hence is a constant because c is a constant. Therefore, by taking any $n > \max(n_0, K_c)$, we do indeed satisfy the inequality for some $n \geq n_0$, as needed.

Alternatively, we can approach this without much difficulty using limits. Note that in $f(n)/g(n)$, we can move factors between the numerator and denominator as is convenient, to get nicer derivatives when applying L'Hôpital's Rule. In particular,

$$\frac{f(n)}{g(n)} = \frac{2^{10n}}{n^{10} \cdot 10^{2n}} = \frac{10.24^n}{n^{10}}$$

so

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{10.24^n}{n^{10}} = \infty \, .$$

Since the limit exists and is infinity, we have that $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$. (To be precise, the problem preamble did not state that when the limit exists, a nonzero limit is *necessary* for $f(n) = O(g(n))$, only that it is *sufficient*. But this is indeed true, and it was stated in discussions and Piazza that it could be used.)

(5 pts) 2. **Comparing asymptotic running times.**

Suppose that Algorithm X has running time $T_X(n) = \Theta(n)$, Algorithm Y has running time $T_Y(n) = \Theta(n^2)$, and Algorithm Z has running time $T_Z(n) = O(n^2 \log n)$.

As usual, all running times are stated in terms of the *worst case* for inputs of size n. That is, $T_X(n)$ is the *maximum* number of steps for which X runs, taken over all inputs of size n (and similarly for T_Y, T_Z). Choose all claims that are necessarily true.

- ☐ On every input, X runs faster than Y.
- ☐ For all large enough n, X runs faster than Y on every input of size n.
- ■ **For all large enough n, there is an input of size n for which X runs faster than Y.**
- ☐ For all large enough n, there is an input of size n for which Y runs faster than Z.

Solution: Before considering the specific claims, we highlight the main conceptual points for this problem:

- An algorithm can take varying amounts of time on different inputs, even ones of the same size. (For example, consider insertion sort, which is fast when its input array is already sorted, but much slower when its input is far from sorted.) In this class, we care mainly about the *worst-case* running time over all inputs of a particular size, so that we have a runtime guarantee no matter what the input is. But on certain inputs, an algorithm might perform better than its worst-case behavior.

- An $O(\cdot)$ bound is merely an (asymptotic) *upper bound*. A function that grows much more slowly than $g(n)$ is still considered to be $O(g(n))$. Similarly, an $\Omega(\cdot)$ bound is merely an (asymptotic) *lower bound*; a function that grows much more quickly than $g(n)$ is still considered to be $\Omega(g(n))$. By contrast, a $\Theta(\cdot)$ bound is both an (asymptotic) upper *and* lower bound. Any function that is $\Theta(g(n))$ must be 'sandwiched' between two constant multiples of $g(n)$ (above a certain threshold for n), i.e., it must scale just as $g(n)$ does, as n grows large.
- Asymptotic bounds like $O(\cdot)$ and $\Omega(\cdot)$ say how a function (often, an algorithm's runtime) scales *as the input size n grows large*, beyond some threshold (denoted n_0). They do not say anything about the function's behavior for "small" values of n (below the threshold). Without any additional hypotheses, the function's behavior below the threshold can be arbitrary.

Now, we explain each option:

- The **first option** is not necessarily true; it can be false since Y might be faster than X for some *small* inputs (i.e., small input size n). For example, it could be the case that $T_X(n) = 100n$ and $T_Y(n) = n^2$. Then on *any* input of size $n = 2$, Y runs in time *at most* $T_Y(2) = 2^2 = 4$. By contrast, on a *worst-case* input of size $n = 2$, X runs in time *exactly* $T_X(2) = 100 \cdot 2 = 200$. So, Y is faster than X on such an input.
- The **second option** is not necessarily true; it can be false since Y might be faster than X for some best-case inputs. For example, it could be the case that Y runs for constant time on inputs with the first character 0, while X runs for $T_X(n) = n$ time for such inputs. This still can satisfy $T_Y(n) = \Theta(n^2)$ if the worst case input exists as inputs with the first character not 0. Then, Y runs faster than X on inputs with the first character 0.
- The **third option** is necessarily true. For each value of n, there is some *worst-case* size-n input y_n for Y, i.e., Y run on y_n takes time exactly $T_Y(n)$. Then because $T_Y(n) = \Omega(n^2)$, there is a positive constant $c > 0$ such that for all large enough n, Y run on y_n takes time *at least* cn^2. Because $T_X(n) = O(n)$, there is a positive constant $c' > 0$ such that for all large enough n, X run on y_n takes time *at most* $c'n$. Finally, because $c'n < cn^2$ for all large enough n (namely, when $n > c'/c$), we conclude that X runs faster than Y on y_n for all large enough n.
- The **last option** is not necessarily true; it can be false because the $O(n^2 \log n)$ upper bound on $T_Z(n)$ could be very *loose*, and Z might actually run in much faster than quadratic time (for all inputs). For example, it could be the case that $T_Z(n) = 10 = O(n^2 \log n)$, whereas Y takes time $T_Y(n) = 20n^2 = \Theta(n^2)$ for *every* input of size n. Then, because $20n^2 > 10$ for every $n \geq 1$, Z is faster than Y on *every* input.

3. **EECS 376 lover.**

Consider the following algorithm:

```
1: function EECS376Lover(n, k)          ▷ n is a positive integer, and k ∈ {1, 2, ..., n}
2:     for i = 1, 2, ..., k do
3:         for j = 1, 2, ..., n − k do
4:             Print("I love EECS 376!")
```

(1 pt) (a) Identify the value of k that induces the **most** `"I love EECS 376!"` printed by the algorithm.

(2 pts) (b) Based on your answer in (a), give the **tightest correct asymptotic** (big-O) bound, as a function of n, on the number of `"I love EECS 376!"` printed by the algorithm.

(2 pts) (c) Is the algorithm *efficient*, i.e., runs in at most polynomial time with respect to the input size? Briefly explain your answer.

Solution:

(a) The number of `"I love EECS 376!"` printed is $k(n-k)$. Taking the derivative with respect to k, it is $n-2k$. Therefore, the worst case arises (i.e., this expression attains its maximum) when $k = \lfloor n/2 \rfloor$.

(b) Substituting $k = \lfloor n/2 \rfloor$ into $k(n-k)$, we get the number of prints $\lfloor n/2 \rfloor \lceil n/2 \rceil$. Therefore, the tightest asymptotic bound is $O(n^2)$ in terms of n.

(c) This is **not** polynomial—it is exponential—in the input size, because the input size is $\log n + \log k = \log(nk) = \Theta(\log n)$.

(10 pts) 4. **Power of induction.**

Consider the following algorithm to compute a^b, where a and b are some integers.

```
1: function Pow(a, b)
2:     if b = 0 then
3:         return 1
4:     if b is even then
5:         return (Pow(a, b/2))^2
6:     else
7:         return a · (Pow(a, (b − 1)/2))^2
```

Prove that the algorithm is correct by induction.

Solution: We will prove that the algorithm correctly returns a^b by induction on the value of b.

Base case: $b = 0$: The algorithm returns 1 which is the correct output a^0.

Inductive step: Suppose the algorithm correctly returns for $b = 0, 1, \cdots, k-1$. We need to show that the algorithm correctly returns for $b = k$.

- If k is even, then the algorithm returns in line 5. Since the algorithm correctly returns for $k/2 < k$, the value in line 5 is $(a^{k/2})^2 = a^k$ which is the correct desired output.
- If k is odd, then the algorithm returns in line 7. Since the algorithm correctly returns for $(k-1)/2 < k$, the value in line 7 is $a \cdot (a^{(k-1)/2})^2 = a \cdot a^{k-1} = a^k$ which is the correct desired output.

In either case, the algorithm outputs a^k correctly.

Therefore, we conclude that the algorithm outputs a^b correctly for all non-negative integers b.

5. **Pigeons and the Contra-.**

Recall the Pigeonhole Principle, which states "If n pigeons are placed into m pigeonholes, where $n > m$, then at least one pigeonhole contains more than one pigeon."

Prove the Pigeonhole Principle using

(5 pts) (a) proof by contradiction, and

(5 pts) (b) proof by contrapositive.

Solution: Proof by contradiction: Suppose, for the sake of contradiction, that n pigeons are placed into m pigeonholes with $n > m$ but every pigeonhole contains at most one pigeon. Since there are m pigeonholes each containing at most one pigeon, the total number of pigeons placed into pigeonholes should be less than or equal to m. However, this contradicts that $n > m$ pigeons are placed into pigeonholes.

Proof by contrapositive: The contrapositive of the given statement is: "If each of m pigeonhole contains at most one pigeon with $n > m$, then not all n pigeons can be placed into pigeonholes." Since there are m pigeonholes each containing at most one pigeon, the total number of pigeons placed into pigeonholes should be less than or equal to m. Therefore, $n > m$ pigeons cannot all be placed into pigeonholes.

(15 pts) 6. **Potential method.**

Alice is playing a `FactorFinding` game with herself. The integer factorization is not exciting enough so she decides to consider another number system.

Alice thinks about the "complex integers", i.e. $a + bi$ where a, b are integers and $i^2 = -1$. For example,

- addition: $(1 + 2i) + (3 + 4i) = 4 + 6i$;
- multiplication: $(1 + 2i)(3 + 4i) = 3 - 8 + (6 + 4)i = -5 + 10i$

She then plays the game as follows. Alice starts with an arbitrary "complex integer" $a_1 + b_1 i$ and $k = 1$. Alice tries to find $a_{k+1} + b_{k+1} i$ that divides $a_k + b_k i$ which means there exist two

integers c and d such that $(a_{k+1} + b_{k+1}i)(c + di) = a_k + b_k i$. If $|c| + |d| \leq 1$, then the game terminates. Otherwise, she increments k and repeats this step.

Note: The initial complex number $a_1 + b_1 i \neq 0$; that is, at least one of a_i and b_i must be non-zero.

Question: Can Alice continue the game forever? If yes, give an example infinite run. If not, prove it by the potential-function method.

Hint: Try the potential function $\Phi(a + bi) = a^2 + b^2$.

Solution: No. The factorization process will terminate for any starting value.

Define $\Phi(a + bi) = a^2 + b^2$ as suggested by the hint. Observe that for $a, b, c, d \in \mathbb{Z}$,

$$\begin{aligned}\Phi((a + bi)(c + di)) &= \Phi((ac - bd) + (ad + bc)i) \\ &= (ac - bd)^2 + (ad + bc)^2 \\ &= a^2c^2 - 2abcd + b^2d^2 + a^2d^2 + 2abcd + b^2c^2 \\ &= (a^2 + b^2)(c^2 + d^2) \\ &= \Phi(a + bi)\Phi(c + di).\end{aligned}$$

Let the current number be $a_k + b_k i$. Suppose Alice finds some $a_{k+1} + b_{k+1}i$ such that $(a_{k+1} + b_{k+1}i)(c + di) = a_k + b_k i$ for some integers c, d. Then as above we have

$$\begin{aligned}\Phi(a_k + b_k i) &= \Phi((a_{k+1} + b_{k+1}i)(c + di)) \\ &= \Phi(a_{k+1} + b_{k+1}i)\Phi(c + di) \\ &= \Phi(a_{k+1} + b_{k+1}i)(c^2 + d^2)\end{aligned}$$

If $|c| + |d| \leq 1$, Alice loses and the game terminates. Otherwise, the potential drops by at least one half since for $c, d \in \mathbb{Z}$, we have $c^2 + d^2 \geq |c| + |d| \geq 2$. In addition, $\Phi(a_k + b_k i) \geq 1$ since at least one of a_k and b_k is non-zero. (If we start with a non-zero complex number, it can never have a factor that is 0, so at least one of a_k and b_k must be non-zero for every iteration.) Thus the game will end within $\log_2(a^2 + b^2)$ rounds if she starts with $a + bi$.

7. **Master theorem.**

Consider the recurrence

$$T(n) = \sqrt{n} \cdot T(\sqrt{n}) + O(n).$$

(5 pts) (a) Explain why the master theorem cannot be applied *directly* to give a closed form for $T(n)$.

Solution: The master theorem requires the form $T(n) = aT(n/b) + O(n^c)$, where a is constant, but here $a = \sqrt{n}$. Thus a is not constant as it depends on n. Moreover the recursive call is not of the form $T(n/b)$ for a constant b.

(5 pts) (b) Define $S(n) = T(n)/n$. Using substitution, write a recurrence for $S(n)$.

Solution: We know

$$S(n) = \frac{T(n)}{n}$$

Expanding the definition of $T(n)$, we have

$$= \frac{\sqrt{n} \cdot T(\sqrt{n}) + O(n)}{n}$$

Simplifying, we obtain

$$= \frac{T(\sqrt{n})}{\sqrt{n}} + O(1)$$

Note that $S(n) = T(n)/n$, so $S(\sqrt{n}) = T(\sqrt{n})/\sqrt{n}$, and thus

$$= S(\sqrt{n}) + O(1)$$

This is not in the form for Master Theorem, so the Master Theorem cannot be applied to obtain a closed form solution.

(5 pts) (c) Let $n = 2^m$ and define $R(m) = S(2^m) = S(n)$. Using this substitution, write a recurrence for $R(m)$. **Hint:** You may need to use that $\sqrt{n} = \sqrt{2^m} = 2^{m/2}$

Solution: Applying definitions, we have

$$\begin{aligned} R(m) &= S(2^m) \\ &= S(\sqrt{2^m}) + O(1) \\ &= S((2^m)^{1/2}) + O(1) \\ &= S(2^{m/2}) + O(1) \\ &= R\Big(\frac{m}{2}\Big) + O(1) \end{aligned}$$

(5 pts) (d) Use the Master Theorem and the above recurrence to get an asymptotic expression for $R(m)$, then use it to get asymptotic expressions for $S(n)$ and finally $T(n)$.

Solution: From part (c), we know

$$R(m) = R\Big(\frac{m}{2}\Big) + O(1).$$

Per the Master Theorem, the complexity of this relation is $O(m^0 \log m) = O(\log m)$, so

$$R(m) = O(\log m).$$

Now we use the fact that $R(m) = O(\log m)$ and $R(m) = S(2^m)$ to get an asymptotic bound on $S(n)$.

$$S(2^m) = O(\log m)$$

Plugging in $\log n$ in for m on both sides:

$$S(2^{\log n}) = O(\log \log n).$$

Since $2^{\log n} = n$, we have that:

$$S(n) = O(\log \log n).$$

Now we use the fact that $S(n) = O(\log \log n)$ and $S(n) = T(n)/n$ to derive an asymptotic bound for $T(n)$.

$$\begin{aligned} T(n)/n &= O(\log \log n), \quad \text{so} \\ T(n) &= O(n \log \log n). \end{aligned}$$

(15 pts) 8. **Divide and conquer algorithm.**

The matrix H_t is a $n \times n$ matrix where $n = 2^t$. It is defined recursively, where $H_0 = (1)$, and in general,

$$H_{t+1} = \begin{pmatrix} H_t & H_t \\ H_t & -H_t \end{pmatrix}$$

For example,

$$H_2 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$$

Given a vector $x \in \mathbb{Z}^n$, there is a trivial algorithm that uses $O(n^2)$ operations to compute the matrix-vector product $H_t x$, by first computing the matrix H_t and then computing its product with x. Describe an algorithm that computes $H_t x$ in $O(n \cdot t) = O(n \log n)$ operations.

Note: Recall how matrix vector multiplication works. If we have a $n \times n$ matrix A and a vector $x \in \mathbb{Z}^n$, their product is calculated by taking the dot product of x with each row in A. (That is, if the row is $[a_1, a_2, ..., a_n]$ and $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, then the dot product is $a_1x_1 + a_2x_2 + ... + a_nx_n$).

For example, here we have a 2×2 matrix A and a vector $x \in \mathbb{Z}^2$.

$$Ax = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} ax_1 + bx_2 \\ cx_1 + dx_2 \end{bmatrix}$$

Moreover, instead of multiplying each individual row and column, we can also multiply matrices in blocks. For example, if we have a 4×4 matrix A and a vector $x \in \mathbb{Z}^4$, we can break A into 4 smaller matrices, say 2×2 matrices H, I, J, K. We can also break x into 2 vectors $\vec{x}_1$ and $\vec{x}_2$ each of size 2. This would give the following formulation of the product Ax.

$$Ax = \begin{bmatrix} H & I \\ J & K \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} = \begin{bmatrix} H\vec{x}_1 + I\vec{x}_2 \\ J\vec{x}_1 + K\vec{x}_2 \end{bmatrix}.$$

Solution:

Require: A vector x of dimension 2^t
1: **function** ComputeHx(x)
2: $t := \log_2$(the dimension of x)
3: **if** $t = 0$ **then**
4: **return** x
5: **else**
6: $A =$ ComputeHx($x[0...2^{t-1} - 1]$)
7: $B =$ ComputeHx($x[2^{t-1}...2^t - 1]$)
8: **return** $\begin{bmatrix} A + B \\ A - B \end{bmatrix}$

Runtime analysis:
Let $T(n)$ be the runtime of ComputeHx with input size $n = 2^t$.
Lines 2-3 are $O(1)$.
Lines 6 and 7 are each $T(n/2)$, since we call ComputeHx with a vector that is half the size of x.
Line 8 is $O(n)$, since we add or subtract vectors of half the size of x twice.
Then the recurrence relation is $T(n) = 2T(n/2) + O(n)$.
Using the Master Theorem, since $2/2^1 = 1$, $T(n) = O(n \log_2 n)$.

Correctness Analysis:
For the base case $t = 0$, $\vec{x}$ is a 1x1 vector, and $H_t\vec{x} = [1]\vec{x} = \vec{x}$.
Otherwise,

$$H_t\vec{x} = \begin{bmatrix} H_{t-1} & H_{t-1} \\ H_{t-1} & -H_{t-1} \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \end{bmatrix} = \begin{bmatrix} H_{t-1}\vec{x}_1 + H_{t-1}\vec{x}_2 \\ H_{t-1}\vec{x}_1 - H_{t-1}\vec{x}_2 \end{bmatrix}$$

which is exactly what ComputeHx returns. □

(10 EC pts) 9. **Extra credit:** *You are not required to do this question to receive full credit on this assignment.* To receive the bonus points, you must typeset this **entire** assignment in LaTeX, and draw a table with two columns that includes the *name* (e.g., "fraction") and an *example* of each of the following:

- fraction (using `\frac`),

- less than or equal to,
- union of two sets,
- summation using Sigma ($\sum$) notation,
- the set of real numbers ($\mathbb{R}$); write a mathematically correct statement that applies to *all* real numbers $x \in \mathbb{R}$.

Solution: The following LaTeX code, which uses macros from the provided `header.tex`, was used to create the table below.

```
\renewcommand{\arraystretch}{2}
\begin{tabular}[t]{|c|c|}
  \hline
  fraction
  & $\frac{4}{8} = \frac{1}{2}$ \\
  \hline
  less than or equal to
  & $\abs{\inner{\vec{x}, \vec{y}}}^2 \leq
    \length{\vec{x}}^2 \cdot \length{\vec{y}}^2$ \\
  \hline
  union of two sets
  & $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$ \\
  \hline
  sum using Sigma notation
  & $\displaystyle
    \sum_{n=1}^\infty \frac{1}{n^2} = \frac{\pi^2}{6}$ \\
  \hline
  true statement $\R$, $\C$, and $\Z$
  & $\Z\subset \R\subset \C$ \\
  \hline
\end{tabular}
```

fraction	$\frac{4}{8} = \frac{1}{2}$
less than or equal to	$\lvert\langle \vec{x}, \vec{y}\rangle\rvert^2 \leq \lVert\vec{x}\rVert^2 \cdot \lVert\vec{y}\rVert^2$
union of two sets	$\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$
sum using Sigma notation	$\displaystyle\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$
true statement $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{Z}$	$\mathbb{Z} \subset \mathbb{R} \subset \mathbb{C}$