Before we begin the algorithm, let us note that the hash function is in essence treating the input submatrix as a binary number, where the number is represented as just the matrix as a 1d array. So while computing this hash for all the submatrices in T, we will refer to operations as bit shifts and masks, because they use this intuition of $h_p(x)$ as a binary number.

First, precompute all the $n' * m'$ powers of 2 mod p that we would need for the actual hash function calculation (trivially done in $O(mn)$ time). Next, we need to compute the inverse of each power of 2 mod p that we are calculating here. The inverse of 2 mod p is $\frac{p}{2}$. This is because $2 * \frac{p}{2}$ is $p + 1$, which is 1 mod p, therefore an inverse mod p. So we compute all these inverses for each power of 2 we need, trivially done in $O(mn)$ time as well. Next, for each row of length $m'$, calculate its value as a binary number. Using the same method as karp rabin for each of these rows, we can calculate these values for each row in $O(m)$ time. We will do this for all the rows in $O(mn)$ time total.

Now we will describe the method to move the hash window down a row, and how to recompute the hash at a given row column starting point.

To move the hash window down a row, we are basically shifting our binary number down $m'$ bits, and setting the top $m'$ bits of the number to be the new row we get. Since we pre-calculated the binary value of each row, we can do a shift down (multiplying by our precomputed inverse for the power of 2; If we shift down by i, we multiply by our precomputed inverse of $2^i$) to remove the row we are leaving behind, and can multiply this pre-calculated value by the appropriate power of two, and add it to our current (shifted down) hash to get the new value. Since operations modulo p take constant time, down 1 row movement takes $O(1)$ time.

Given a starting column and row, for each row that will be used in that hash, we look up its precomputed value. Then, for each of these we shift it up (multiply by the appropriate power of 2) to correspond to the set of bits that row corresponds to. So we get the hashed value by essentially replacing the inner summation with looking up results. Since there are at most $m$ rows, and all the arithmetic takes constant time, this computation takes $O(m)$.

Now we will show how to compute the hash of all submatrices in $O(mn)$ time using these procedures. We start with our window on the top left corner of the matrix, and compute its hash value as above - $O(m)$ time. Next, we move down the window by 1 row until the bottom, using the procedure above. This takes $O(n)$ time. Then, we go back to the top of the matrix, and move over one column, and repeat the process. We will make a full down pass $m$ times, and since each takes $O(n)$ time, the total time for all the down passes is $O(mn)$. Next, we will only perform a full hash computation $n$ times - one for each column, and since each computation takes $O(m)$ time, the total time is $O(mn)$. Since no operation in this sequence took more than $O(mn)$ time, the total time to compute all of these hash values is $O(mn)$.

From part A, we know that the hash function $h_p(x)$ represents is treating the submatrix X like a binary number modulo p.

Following this procedure, consider two sub matrices A, B, and let the numbers they represent be a, b. For them to hash to the same value a - b mod p must equal 0. If we let a - b = d, then to find the probability of a false positive, we want to find the probability that we pick a prime that is a divisor of d. The maximum number of bits in d can be $m*n$, because a and b can be at most $m*n$ bits long. Following similar analysis in the lecture, then the number of prime divisors d can have is at most $m*n$. The probability that p is one of them is

$$\frac{mn}{\text{number of primes under } M}$$

From lecture notes, we know this is equal to

$$\frac{mn}{\text{prime}} = \frac{mn * \ln(M)}{M}$$

So we know the probability of two matrices colliding is this. If we are trying to match to a specific submatrix P, then there are $O(mn)$ submatrices to compare against, so by the union bound, our total probability is at most

$$\frac{(mn)^2 * \ln(M)}{M}$$
We know that every edge in the graph corresponds to an element in the hash table. Next, we also know that since the vertices in the graph are the indices in the hash table, and the edges represent the different hash values for elements, if we have a connected component, then the possible indices that can be swapped out during the hashing procedure are only the elements in this connected component, because we follow the edges when doing a swap, we can only reach indices connected in this component. So, for this comparison, we can consider the graph to be just this connected component. Next, if there are only \( K \) nodes in the connected component, then we know there are \( K \) spaces for elements to be. If we have two cycles, then there must be \( > K \) elements in the table, because if there were exactly \( K \), we would have at most 1 cycle. So if there are at least two cycles, we have more elements than buckets to put the elements, so we will always fail.

Because there is at most one cycle in the graph, we know there is room to place element \( x \). So, we need to show that the walk of the graph that the insertion method performs will eventually reach the open position in the hash table.

Consider the edges of the graph to have direction - if an edge is pointing into a node, that means that the element that edge corresponds to is in that node of the hash table. By construction, each node in the graph can only have one incoming edge, since a node can hold only one element.

Additionally because \( h_1(x) \) and \( h_2(x) \) can not point to the same node (last bit property), we know if we swap an element with the element at the node we swapped, we will traverse the edge to a different node. Also, we can only follow in edges back across them, because when we remove an element that corresponds to the in edge, we have to follow the in edge back across, because the evicted element has only one other position to be in.

If \( R \) is a tree, then consider running insert(\( x \)). If the algorithm ever reaches a leaf, then we know that leaf must be empty. This is true because consider the case when we actually try to swap an element into a leaf. That means there must be an edge from the left into the current node, else we would not be trying to swap into the leaf. Since the only edge from this leaf was pointing out, we know the leaf was empty, so we can successfully put our displaced element into this leaf and complete the insert. We know that the walk of the graph that the algorithm does must eventually reach a leaf, assuming there are no empty spaces inside the graph. This is because during swaps, we always traverse these back edges onto a new node. If we keep following these back edges, and there is no cycle, we must eventually reach a node with no more outgoing edges, the empty leaf. If the algorithm terminates before reaching a leaf, then it found an empty slot that wasn’t a leaf, which is fine, because we succeeded inserting \( x \) then.

If \( R \) contains a cycle, we must show that insert(\( x \)) always succeeds. Because there is at most one cycle, we know that every connection to a node in the cycle must be a tree. If we insert, and terminate before reaching the cycle, we are finished. If we enter the cycle, we know that the walk will exit the cycle where it entered. This is because the cycle must be a directed cycle, otherwise we wouldn’t uphold at most one element in each cell. So we will follow these directed edges and flip them as we traverse the cycle, and then when we get to where we entered the cycle, the edge we took to enter the cycle will be an in edge (because we swapped an element into the cycle), so when we reach this node, we will exit the cycle, because we have to follow this edge. After we follow this edge, we know we will be trying to insert into a tree, so we know from the previous part that we will succeed.

So if there is at most 1 cycle in \( R \cup \{x\} \), we will always be able to insert(\( x \)).
(a) We consider three values $x$, $y$, and $z$. We note that if $x$, $y$, and $z$ disagree in at least three distinct positions $(i, j, k)$, then proving 3-universality is straightforward. We fix matrices $A_1$ and $A_2$ except for columns $i$, $j$, $k$. We can assume that $x_i = 1$ and therefore $y_i = z_i = 0$ (the case where $x_i = 0$, $y_i = z_i = 1$ will be discussed as well, but it’s analogous). We can therefore set the $i$th column of $A_1$ as we please to determine our value of $h(x)$. The same can be done for $y$, and $z$.

We also note that if instead we had $x_i = 0$ and $y_i = z_i = 1$, we could be picking columns from $A_2$ instead. In each case, we independently have $2^m$ choices for the column in question, each one producing a unique selection of $h(x), h(y), h(z)$, giving us a probability of $1/(2^m)^3 = 1/2^{3m}$, therefore demonstrating 3-universality. We also note that when picking a column from one matrix we can simply ignore the column from the other matrix as it won’t be picked.

We now consider the more complicated case where $x$, $y$, and $z$ disagree in only two columns, specifically $i$, $j$. We can say without loss of generality $x_i \neq y_i$, $x_i = z_i$, $x_j = y_j$ and $y_j \neq z_j$. We therefore consider the following multiplications, where $A_{1k}$ refers to the $k$th column of $A_1$, and similar notation holds for $A_2$.

\[
A_1 x = \sum_{k \neq i, j} A_{1k} x_k + A_1 x_i + A_1 x_j
\]

\[
A_2 \bar{x} = \sum_{k \neq i, j} A_{2k} \bar{x}_k + A_2 \bar{x}_i + A_2 \bar{x}_j
\]

We note that there are 4 cases:

- $x_i = 0, x_j = 0 \rightarrow$ in this case, we use $A_{2i}, A_{2j}$
- $x_i = 1, x_j = 0 \rightarrow$ in this case, we use $A_{1i}, A_{2j}$
- $x_i = 0, x_j = 1 \rightarrow$ in this case, we use $A_{2i}, A_{1j}$
- $x_i = 1, x_j = 1 \rightarrow$ in this case, we use $A_{1i}, A_{1j}$

So in each case, our hash depends only on two vectors, each of which can be chosen in $2^m$ different ways, and is independent of $y$ and $z$. That is to say that in each case, we select two different columns so the choice of a single one of the columns does nothing in determining our range of final values. We still have one column we can adjust in $2^m$ different ways and in so doing get all $2^m$ possible values. As a result, the fact that each of $h(x), h(y), h(z)$ can be made whatever we want it to be independently gives us that we have $2^{3m}$ possibilities giving us a probability of $1/2^{3m}$ of choosing any particular one. Therefore, in this case as well we have 3-universality.
(b) For this, we need to show a counterexample. Let us consider $u = 2$. We can consider all 4 possible numbers within this universe, 00, 01, 10, and 11.

We note the following multiplications (simplified matrix multiplications with same column notation as in the previous part)

\[
\begin{align*}
  h \left( \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) &= A_{21} + A_{22} \\
  h \left( \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) &= A_{11} + A_{22} \\
  h \left( \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) &= A_{21} + A_{12} \\
  h \left( \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) &= A_{11} + A_{12}
\end{align*}
\]

We can see from this that $h(11) = h(10) + h(01) - h(00)$, which means that all 4 are not independent. Therefore, know that $(Pr(h(11) = v_1 \text{ and } h(10) = v_2 \text{ and } h(01) = v_3 \text{ and } h(00) = v_4)) > \frac{1}{2^m}$ and therefore we do not have 4-universality by counterexample.

**Problem 4 Double String**

**Solution 1: Simple $O(n^2)$ using fingerprinting:**

Preprocess the string in $O(n)$ time such that the Karp-Rabin fingerprint can be computed for any range in $O(1)$ time. This technique explained in recitation 5.

So you just try all lengths starting from $n/2$ down to 1 quitting when a solution is found. For a given length $\ell$ you just compare each block of length $\ell$ with the subsequent one of length $\ell$. So for each length value the time is $O(n)$.

**Solution 2 $O(n^2)$ solution using two fingers:**

For a given $\ell$ scan the string with two fingers, one at position $i$ and the other at $i + \ell$. Keep a counter $c$ which starts at 0. If $s[i] = s[i + \ell]$ then increment $c$. If not reset $c$ to zero. If $c$ reaches $\ell$ then you know you've found a pair of repeated strings of length $\ell$.

**Solution 3 Raymond Kang’s $O(n \log^2(n))$ algorithm.**

Given a length $\ell$, you want to test the string for a repeat of length $\ell$. If you could do it in time $O(n/\ell)$, then when you try it for all $\ell$, the total is $O(n \log n)$.

The way you do the test is to pick a bunch of anchor points at positions $\ell, 2\ell, 3\ell, \ldots$ in the string. For each pair of neighboring anchor points you look for a repeat work that touches both of them. If you use Karp-Rabin fingerprinting and apply binary search you can do this for each pair of neighboring anchor points in $O(\log \ell)$. 
Solution 4 He Pufan’s $O(n \log(n))$ algorithm.

This is the fastest one of all, in both theory and practice. It makes use of the Z algorithm, which is beautifully explained by Carl here:


So my $O(N \log N)$ repeated word algorithm is essentially divide and conquer + LCP by Z algorithm.

solve(L, R) only looks for repeated substring completely in s[L:R].

// l, l+1, l+2, ..., r-1

If the answer (longest repeated substring) does not contain the middle letter s[M] where $M = (L+R)/2$, then it must lie completely on the left half or the right half. We solve it in two recursions solve(L, M), solve(M, R).

Otherwise, s[M] must be in one of the two repeated words. We enumerate the position of M’, where M’ is the corresponding/mirrored M position in the other word. So the length of answer is $|M - M'|$. We only need to verify that

```
L....................M....................R
+--------(<<<<<<<<<<>>><<<<<<<<<<>>>)----+
.................................M'.......```

```
lcp(M to the right, M’ to the right) + lcp(M to the left, M’ to the left) >= |M - M’|
```

where lcp = longest common prefix.

Because M is stationary, we can prepare all lcp offline. Z algorithm is the simplest $O(N)$ approach to do that. Z algorithm solves lcp of a string against all its suffixes. Here we construct (s[M:R] + '#' + s[L:M]) to solve the left-to-right lcp. And similar for the inverse direction.