CS648 Assignment - 2

Team Name:

wPygRi7M8Fy CYZwK2p

Soham Samaddar 200990

Aditya Tanwar 200057

February 2023

Q1. Randomized Quick Select

Let S be a set of n real numbers. Consider the randomized algorithm `Rand-QSelect(k, S)` described below that finds the k^{th} smallest element from the set S.

```
x ← uniformly randomly selected pivot from S
r ← rank of x in the set S
if (r = k) report x
if (r > k) report Rand-QSelect(k, S_{<x})
if (r < k) report Rand-QSelect(k − r, S_{>x})
```

Where $S_{<x}$ and $S_{>x}$ are the sets consisting of all those elements that are respectively smaller and greater than the element x. Prove that the expected number of comparisons is at most $3.5\,n$.

Solution: We follow an approach similar to the one followed in the lectures to find the expected number of comparisons made by the randomized quick sort algorithm. Specifically, we shall analyse comparisons from the perspective of a pair of elements.
We use the same notation as used in the lecture, where e_i denotes the i^{th} smallest element in the set. Further, let the element of interest be e_k.

We are interested in knowing when the element e_i is compared with the element e_j ($i < j$ for the rest of the solution). The analysis from the lectures cannot be borrowed directly, since we run into a problem in cases when a pivot x ($e_i < e_j < x < e_k$) is picked; where after the "pruning" of the set, both the elements, e_i and e_j will be disposed. But, there are some important observations that still hold from the analysis done in lectures-

- Suppose e_i and e_j both exist in the set currently being examined, and $x = e_i$ is picked as the pivot. Then the two elements' comparison is guaranteed. Similarly, for $x = e_j$. Thus, e_i or e_j being picked as the first element among all the elements between them (and including them), continues to be a necessary condition. However, it is not guaranteed that any element in this range will be picked throughout the algorithm.
- Picking a pivot lesser than e_i, or greater than e_j does not affect their chances of being compared, as long as they remain in the set S after the "pruning".

However, these complicacies can be solved by redirecting our focus from the subarray $[e_i, e_j]$ (by subarray here, we mean the subset of S containing elements greater than or equal to e_i and and smaller than or equal to e_j) to the smallest subarray containing e_i, e_j, and e_k. All the pairs have been split into three categories, depending on where e_k lies with respect to them. Further, let P_{ij} be the probability that the two elements e_i and e_j are compared. Then, we have:

- $e_k < e_i < e_j$: If an element outside this range is picked, then clearly, all the three elements shall continue to lie in the same set. Thus, the chances of e_i and e_j getting compared is not changed. From analysis similar to that done in the lectures, if any one of e_i or e_j is picked, then they are guaranteed to be compared, and if any pivot $e_i < x < e_j$ is picked, then it is guaranteed that they shall not be compared.
  The only minor difference is when a pivot $e_k \leq x < e_i$ is picked, in which case, both the elements e_i and e_j will be disposed and will not be compared.
  In essence, our "sample space" has changed from previously being the range of indices $[i, j]$ to $[k, j]$ in this case, while the favourable outcomes remain the same, e_i or e_j being the first elements to be picked in the range. To summarise, the probability of the elements being compared in this case is $P_{ij} = 2/(j - k + 1)$.
- $e_i < e_k < e_j$: The analysis of this case is similar to the one done in lectures, where a pivot outside this range will not affect the chances of comparison, the pivot being e_i or e_j guarantees comparison, and the pivot being between e_i and e_j completely disallows them from being compared. Thus, the sample space remains the range of indices $[i, j]$, and there are two favourable cases. Thus, the probability of comparison in this case is $P_{ij} = 2/(j - i + 1)$.
- $e_i < e_j < e_k$: The analysis of this case is similar to the first case. Thus, the probability of comparison in this case is $P_{ij} = 2/(k - i + 1)$.

Let Y_{ij} be an indicator random variable which takes the value 1 when the elements e_i and e_j are compared and 0 otherwise. Clearly, $\mathbb{E}[Y_{ij}] = \mathbb{P}[Y_{ij} = 1] = P_{ij}$. Then, we are interested in the expected value of the random variable, $Y = \sum_{i<j} Y_{ij}$.

$$
\begin{aligned}
\mathbb{E}[Y] &= \sum_{i<j} \mathbb{E}[Y_{ij}] && \text{(From Linearity of Expectation)} \\
&= \sum_{k\leq i<j} \mathbb{E}[Y_{ij}] + \sum_{i<k<j} \mathbb{E}[Y_{ij}] + \sum_{i<j\leq k} \mathbb{E}[Y_{ij}] && \text{(Using the cases described above)} \\
&= \sum_{k\leq i<j} \mathbb{P}[Y_{ij} = 1] + \sum_{i<k<j} \mathbb{P}[Y_{ij} = 1] + \sum_{i<j\leq k} \mathbb{P}[Y_{ij} = 1] \\
&= E_1 + E_2 + E_3 \\
E_1 &= \sum_{k\leq i<j} \mathbb{P}[Y_{ij}] = \sum_{k\leq i<j} \frac{2}{j-k+1} = \sum_{j=k+1}^{j=n} \sum_{i=k}^{i=j-1} \frac{2}{j-k+1} \\
&= \sum_{j=k+1}^{j=n} \frac{2(j-1-k+1)}{j-k+1} && \text{(Since } 2/(j-k+1) \text{ is independent of } i\text{)}
\end{aligned}
$$

$$\leq \sum_{j=k+1}^{j=n} 2 = 2 \cdot (n-k-1+1) = 2 \cdot (n-k)$$

$$E_3 = \sum_{i<j\leq k} \mathbb{P}[Y_{ij}] = \sum_{i<j\leq k} \frac{2}{k-i+1} = \sum_{i=1}^{i=k-1} \sum_{j=i+1}^{j=k} \frac{2}{k-i+1}$$

$$= \sum_{i=1}^{i=k-1} \frac{2(k-i-1+1)}{k-i+1} \qquad \text{(Since } 2/(k-i+1) \text{ is independent } j\text{)}$$

$$\leq \sum_{i=1}^{i=k-1} 2 = 2 \cdot (k-1-1+1) = 2 \cdot (k-1)$$

$$E_2 = \sum_{i<k<j} \mathbb{P}[Y_{ij}] = \sum_{i<k<j} \frac{2}{j-i+1} = \sum_{i=1}^{i=k-1} \sum_{j=k+1}^{j=n} \frac{2}{j-i+1}$$

$$= 2 \cdot \sum_{i=1}^{i=k-1} (H_{n-i+1} - H_{k-i+1}) \qquad (H_n = \textstyle\sum_{i=1}^{i=n}(1/i) \text{ is the } n^{th} \text{ Harmonic number})$$

$$= 2 \cdot \sum_{i=1}^{i=k-1} (\ln(n-i+1) - \ln(k-i+1)) \qquad \text{(Using } H_n \approx \log(n) + 0.58\text{)}$$

$$= 2 \cdot (\ln(n!) - \ln(n-k+1)! - \ln(k!))$$

$$\leq 2 \cdot (n\ln(n) - n + \mathcal{O}(\ln(n)) - (n-k+1)\ln(n-k+1)$$
$$+ n - k + 1 - \mathcal{O}(\ln(n-k+1))$$
$$- k\ln(k) + k - \mathcal{O}(\ln(k))) \qquad \text{(Using Stirling's Approximation[1])}$$

$$\leq 2 \cdot (n\ln(n) - (n-k+1)\ln(n-k+1) - k\ln(k) + \mathcal{O}(\ln(n)))$$

$$\leq 2 \cdot (n\ln(n) - (n+1)\ln((n+1)/2) + \mathcal{O}(\ln(n))) \qquad \text{(Proof provided later)}$$

$$\leq 2 \cdot (n\ln(n) - n\ln(n/2) + \mathcal{O}(\ln(n)))$$

$$= n\ln(4) + \mathcal{O}(\ln(n))$$

$$\approx n\ln(4) \qquad \text{(Ignoring the logarithmic term asymptotically)}$$

$$\mathbb{E}[Y] = E_1 + E_2 + E_3$$

$$\leq 2 \cdot (n-k) + n\ln(4) + 2 \cdot (k-1)$$

$$\leq 3.38n$$

$$\boxed{< 3.5n}$$

Thus, the result follows.

We now show the analysis for

$$n\ln(n) - (n-k+1)\ln(n-k+1) - k\ln(k) \leq n\ln(n) - (n+1)ln(n+1)/2$$

Let $f(k) \coloneqq n\ln(n) - (n-k+1)\ln(n-k+1) - k\ln(k)$. We apply first derivative test and second derivative test on it,

$$f'(k^*) = 0$$

$$
\begin{aligned}
0 + \ln(n - k^* + 1) + 1 - \ln(k^*) - 1 &= 0 \\
\ln(n - k^* + 1) &= \ln(k^*) \\
n - k^* + 1 &= k^* \\
k^* &= (n+1)/2 \\
f''(k = k^*) &= \frac{-1}{n - k^* + 1} - \frac{1}{k^*}\Big|_{k^* = \frac{n+1}{2}} \\
&= \frac{-4}{n+1} < 0
\end{aligned}
$$

The second derivative being negative affirms that k^* is a maxima. Thus, the maximum value of $f(k)$ is realised for $k^* = (n+1)/2$ and this value is $f(k^*) = n\ln(n) - (n+1)(\ln(n+1))/2$.

Observation: There is another way to heuristically/intuitively "argue" why the maximum value of $f(k)$ ought to be realised at $k^* = (n+1)/2$, the **mid-point** of the array. To see it, observe that k "negatively" affects all such pairs which lie completely to its left, or to its right by reducing the probability of their comparison. The other pairs, which have one index on either side of k are not affected by it. Thus, we can attempt to minimize the count of all such pairs whose comparison probabilities are decreased by k, i.e.,

$$
\begin{aligned}
k^* &= \underset{k}{\operatorname{argmin}}(k-1) * (k-2)/2 + (n-k) * (n-k-1)/2 \\
&\Rightarrow 0 = (2k^* - 3)/2 + (-2n + 2k^* + 1)/2 \\
&\Rightarrow 2n - 2k^* - 1 = 2k^* - 3 \\
&\Rightarrow k^* = (n+1)/2
\end{aligned}
$$

Second derivative test can be taken to confirm that k^* does minimize count of such pairs.

Q2. An interesting application of partition theorem

Elgoog, a very reputed company, is going to visit IITK to hire the best qualified (excellent knowledge of the fundamentals of computer science, excellent analytical and creative skills) student in his/her final year. There are n applicants and n is obviously huge since Elgoog is offering a huge package. They will select a person based totally on his/her qualification which can be revealed only through interview. Assume that there is a total order among all n applicants as far as their qualifications are concerned. Since n is huge, it is not possible to interview every applicant. Furthermore, the placement office requires that each applicant should be informed about his/her selection or rejection immediately after the interview. Therefore, the following strategy is followed by Elgoog. They fix a number $k < n$. They interview and reject first k applicants. After that they continue taking interviews and stop as soon as they find an applicant better than the first k applicants. If they don't find any applicant better than the first k applicants, they return without hiring any one.

1. Assuming the applicants appear in a uniformly random order (all permutations are equally likely), what is the probability in terms of k and n that Elgoog will be successful in selecting the best qualified applicant?

[1] Stirling's Approximation: $\ln(k!) \approx k \cdot \ln(k) - k + \mathcal{O}(\ln(k))$

2. For what value of k, is the probability of selecting the best qualified applicant maximum?

Solution: Let $\mathbb{B}[l, r]$ denote the best candidate among the candidates at positions between l and r (both inclusive). Let A be the event that Elgoog will be successful in selecting $\mathbb{B}[1, n]$. Let ξ_i denote the event that $\mathbb{B}[1, n]$ is at the i^{th} position in the applicant queue. Note that the events ξ_i are **mutually exclusive, and partition the entire sample space**. Now,

$$\mathbb{P}[\xi_i] = \frac{1}{n}$$

because the applicant queue is uniform and random and hence each position in the queue is equally likely to host the best candidate. Also,

$$\mathbb{P}[A \mid \xi_i] = 0, \quad i \in \{1, 2, \ldots, k\}$$

since if $\mathbb{B}[1, n]$ is one of the initial k interviewed candidates, they would be rejected immediately. Now, for $i > k$, we claim that

$$\mathbb{P}[A \mid \xi_i] = \frac{k}{i-1}$$

To see why, we prove the following lemma:

Lemma. Given that $\mathbb{B}[1, n]$ is at the i^{th} position in the queue ($i > k$), they are selected by Elgoog **if and only if** $\mathbb{B}[1, i-1]$ is at one of the first k positions in the queue.

Proof. For $\Rightarrow$, since Elgoog selects $\mathbb{B}[1, n]$, they must have rejected every candidate at positions $(k+1), (k+2), \ldots, (i-1)$. Hence, $\mathbb{B}[k+1, i-1]$ must be less qualified than $\mathbb{B}[1, k]$. This forces $\mathbb{B}[1, i-1] = \mathbb{B}[1, k]$, otherwise the above condition would not hold. We are done with this direction.

For $\Leftarrow$, let $\mathbb{B}[1, k] = \mathbb{B}[1, i-1]$. This means that $\mathbb{B}[k+1, i-1]$ is less qualified than $\mathbb{B}[1, k]$. Hence all candidates in this range would be rejected. Finally, Elgoog would select $\mathbb{B}[1, n]$ at the i^{th} position. □

Now, with the above lemma at hand, it is easy to see that the probability that $\mathbb{B}[1, i-1]$ lies in the first k positions is simply $\frac{k}{i-1}$ since each position is equally likely. Finally, by the partition theorem:

$$\begin{aligned}
\mathbb{P}[A] &= \sum_{i=1}^{n} \mathbb{P}[A \mid \xi_i] \cdot \mathbb{P}[\xi_i] \\
&= \sum_{i=k+1}^{n} \mathbb{P}[A \mid \xi_i] \cdot \frac{1}{n} \\
&= \frac{1}{n} \cdot \sum_{i=k+1}^{n} \frac{k}{i-1}
\end{aligned}$$

$$= \frac{k}{n} \cdot \left(\sum_{i=k}^{n-1} \frac{1}{i} \right)$$

Hence,

$$\boxed{\mathbb{P}[A] = \frac{k}{n} \cdot \left(\sum_{i=k}^{n-1} \frac{1}{i} \right)}$$

Asymptotically, the harmonic numbers tend to the natural logarithm.

$$\begin{aligned}
\mathbb{P}[A] &= \frac{k}{n} \cdot \left(\sum_{i=k}^{n-1} \frac{1}{i} \right) \\
&= \frac{k}{n} \cdot \left(\sum_{i=1}^{n-1} \frac{1}{i} - \sum_{i=1}^{k-1} \frac{1}{i} \right) \\
&\approx \frac{k}{n} \cdot \ln \left(\frac{n-1}{k-1} \right) \\
&\approx \frac{k}{n} \cdot \ln \left(\frac{n}{k} \right)
\end{aligned}$$

Now, we simply differentiate this function with respect to k and set it to 0 to find the extreme value.

$$\begin{aligned}
\frac{\ln n}{n} - \frac{\ln k}{n} - \frac{1}{n} &= 0 \\
\ln n &= \ln k + 1 \\
k &= \frac{n}{e}
\end{aligned}$$

Taking a second derivative and inputting $k = \frac{n}{e}$ gives $-\frac{e}{n^2} < 0$ which verifies that this is indeed the maxima. Hence, the probability of selecting $\mathbb{B}[1, n]$ is maximized by taking

$$\boxed{k = \frac{n}{e}}$$

And this maximum probability is $\underline{1/e}$.

Q3. 2-Dimensional Pattern Matching

A different fingerprinting technique can be used to solve one-dimensional pattern matching problem. The idea is to map any bit string s into a 2×2 matrix $M(s)$, as follows:

- For the empty string ϵ, $M(\epsilon) = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$
- $M(0) = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$
- $M(1) = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$
- For non-empty strings x and y, $M(xy) = M(x) \times M(y)$

It can be shown that this fingerprint function has the following properties:

- $M(x)$ is well defined for all $x \in \{0,1\}^*$
- $M(x) = M(y) \Rightarrow x = y$

Starting with the two properties of M listed above, solve the problems that follow.

Note: The arguments/parameters fed into H and H_p might look inconsistent. However, it is expected that the surrounding context will help resolve ambiguities, if any. Regardless, a brief description has been provided here:

- Part 1: A single integer denoting the index in a string is fed into $H(*)$. $H(i)$ for an index i, denotes the "hash" of the substring of length m starting at (and including) the index i.
- Part 3: Throughout the algorithm section, if a complete matrix, like $A[*][*]$ has been fed into H_p, then it denotes the "hash" of the whole $m \times m$ matrix, whereas if a row, like $A[i][*]$ has been fed into H_p, it denotes the "hash" of the i^{th} row of $A_{m\times m}$.
  In the implementation section, substrings (in the form of subrows) of size m have been fed into the input of H_p. For example, $H_p(X[i][j : j+m-1])$ denotes the indices from (j) to $(j+m-1)$ in the i^{th} row of the matrix X.
  Lastly, in the pseudocode section, $H_p(i,j)$ denotes the hash of the $m \times m$ contiguous submatrix of $X_{n\times n}$ starting at the cell (i,j) of X.

1. By using the matrix $M(x)$ for a bit-string x, design an algorithm for the pattern matching problem in 1-dimension (text and pattern are bit-strings of length n and m respectively). The algorithm should take $\mathcal{O}(n+m)$ time. You may make use of the following assumption.
   Assumption: Any arithmetic operation involving two numbers of arbitrary length can be executed in $\mathcal{O}(1)$ time.

Solution: Let the text be denoted by y, $|y| = n$, and the pattern to be matched be denoted by x, $|x| = m$. We begin by making the observation that both $M(0)$ and $M(1)$ are invertible, with

$$M^{-1}(0) \coloneqq (M(0))^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \qquad M^{-1}(0) \coloneqq (M(1))^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

From Linear Algebra, we know that these inverses are *unique*. Also, since the matrices we deal with in this part are all of size 2×2, multiplication of any two matrices takes at most 12 arithmetic operations. Thus, multiplication of a constant number of matrices takes at most $\mathcal{O}(1)$ time.
The algorithm we describe is almost identical in spirit to the one described in the lectures, wherein, a *"hash"* is computed for x, and for each (contiguous) substring of length m of y; and to save time, a relation is found between two adjacent substrings.

More specifically, let $1 \le i \le n-m+1$ be an index in the string y (1-based indexing), and let

$$H(i) \coloneqq M(y[i : i+m-1]) = \prod_{j=0}^{j=m-1} M(y_{i+j})$$

denote the "hash" of the substring $y[i : i + m - 1]$. Also, we know that the substring $y[i+1 : i+m]$ can be realised from $y[i : i+m-1]$ by simply removing the first character y_i, and appending the last character y_{i+m} to the remaining string. More abstractly, we need to "undo" the action of y_i and "do" the action of y_{i+m} to obtain $y[i + 1 : i + m]$.
This is exactly what we do to $H(i)$ as well, i.e., we pre-multiply it by $M(y_i)^{-1}$ and post-multiply it by $M(y_{i+m})$. Mathematically, we have the following relation:

$$H(i + 1) = M(y_i)^{-1} \times H(i) \times M(y_{i+m})$$

Algorithm: With this relation in hand, we are finally in a position to describe the algorithm. We compute $M(x)$ and store it, computation of which takes $\mathcal{O}(m)$ time (since there are $(m-1)$ matrix multiplications). We then compute $H(1)$ which again takes $\mathcal{O}(m)$ time. Thereafter, we compute $H(i)$ for $2 \leq i \leq n - m + 1$ using $H(i - 1)$. Computation of a single $H(i)$ shall take $\mathcal{O}(1)$ time (since there are just 2 matrix multiplications); however, since there are $\mathcal{O}(n)$ *"hashes"* to be computed, this step would take $\mathcal{O}(n)$ time in total. Thus, the total time complexity is $\underline{\mathcal{O}(n + m)}$.

2. The entries of $M(x)$ can be quite long; however, they will be bounded by Fibonacci number F_n for $|x| = n$. Therefore, taking inspiration from one of the lectures, we keep entries of M modulo a prime number p picked randomly uniformly from a suitable range. Do proper analysis to find the range from which you need to pick the prime number randomly to achieve error probability $< 1/n^4$.

Solution: We define the matrix A_p for any matrix A, which is created by replacing each entry a_{ij} of A, by $a_{ij} \pmod p$. We claim that $M_p(0)$ and $M_p(1)$ are invertible under the field $\mathbb{Z}_p$ with the inverses as follows:

$$M_p(0) = M(0) \qquad M_p^{-1}(0) = \begin{bmatrix} 1 & 0 \\ p-1 & 1 \end{bmatrix}$$

$$M_p(1) = M(1) \qquad M_p^{-1}(1) = \begin{bmatrix} 1 & p-1 \\ 0 & 1 \end{bmatrix}$$

To see why, simply note that multiplying a matrix A_p with $M_p(0)$ simply preserves the first column of A and adds the first column to the second column (element wise) for the second column. This is obviously invertible by simply subtracting the first column from the second column in $A_p \cdot M_p(0)$, and can be attained by using $M_p^{-1}(0)$. A very similar argument holds for $M_p(1)$.

Similarly, the *"hash"*, when calculated under modulo p, is denoted by H_p, instead of H. Now, let A_p and B_p be hashes of two binary strings x, y with $x \neq y$. Also, a_{ij} and b_{ij} denote the elements of A_p and B_p, that is:

$$A_p = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad B_p = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

First, note that all a_{ij} and b_{ij} are bounded above by F_m for a binary string of length m as given in the question. Also, we know that $F_m \leq 2^m$ since the m^{th} Fibonacci number is

approximately the m^{th} power of the golden ratio (≈ 1.618). So $\log a_{ij},\ \log b_{ij} \leq m$.

The probability of an error is $\mathbb{P}[a_{11} = b_{11},\ a_{12} = b_{12},\ a_{21} = b_{21},\ a_{22} = b_{22}] \leq \mathbb{P}[a_{11} = b_{11}]$, since $\mathbb{P}[A \cap B] \leq \mathbb{P}[A]$, for any events A, B. Now, the analysis to calculate an upper bound on $\mathbb{P}[a_{11} = b_{11}]$ is exactly similar to that done in class. In a range $[2, t]$, the number of primes is approximately $\frac{t}{\log t}$ and the number of primes that can divide a_{11} and b_{11} is at most $\max(\log a_{11}, \log b_{11}) \leq m$. So, the error probability is bounded by $\frac{m \log t}{t}$. So, a bound of $\frac{1}{n^4}$ can be attained by the following choice of t (assuming n to be large enough):

$$
\begin{aligned}
t &= 8n^4 m \log(nm) < n^7 && (1)\\
\log(t) &= \log 8 + 4\log n + \log m + \log\log(nm) \\
&\leq \log n + 4\log n + 7\log m + \log\log(nm) && (\text{For } n > 8)\\
&\leq 7\log(nm) && (\log\log nm \leq \log n^2 \text{ as } m \leq n)\\
\log(t) &\leq 7\log(nm) && (2)\\
\frac{m\log(t)}{t} &\leq \frac{m \cdot 7\log(nm)}{t} && (\text{From } 2)\\
&\leq \frac{m \cdot 7\log(nm)}{8n^4 m \log(nm)} \\
&\leq \frac{7}{8n^4} \\
&\boxed{< \frac{1}{n^4}} && (3)
\end{aligned}
$$

Thus, the bound on error probability follows for choice of $p\,(= \mathcal{O}(n^{\mathcal{O}(1)}))$ in the range $[2, 8n^4 m \log(nm)]$.

Note: The four events $a_{11} = b_{11}, a_{12} = b_{12}, a_{21} = b_{21}, a_{22} = b_{22}$ may or may not be independent. Hence, we have not assumed $\mathbb{P}[a_{11} = b_{11},\ a_{12} = b_{12},\ a_{21} = b_{21},\ a_{22} = b_{22}] = \mathbb{P}[a_{11} = b_{11}] \cdot \mathbb{P}[a_{12} = b_{12}] \cdot \mathbb{P}[a_{21} = b_{21}] \cdot \mathbb{P}[a_{22} = b_{22}]$. Instead, we use a much safer inequality to come up with an upper bound.

3. Consider the two-dimensional version of the pattern matching problem. The text is an $n \times n$ bit-matrix X, and the pattern is an $m \times m$ bit-matrix Y. Of course, $m \leq n$. A pattern match occurs if Y appears as a (contiguous) sub-matrix of X. Get inspired from the randomized algorithm for part (b) above to design a randomized Monte Carlo algorithm for this 2-dimensional pattern matching problem. The running time should be $O(n^2)$ and the error probability should be $< 1/n^4$.

Solution: We shall first describe the "essence" of the algorithm, followed by its implementation details to achieve the required running time of $\mathcal{O}(n^2)$. Again, multiplication of any two 2×2 matrices can be done in at most 24 arithmetic operations (addition, multiplication, and modulo). Thus, the multiplication of a constant number of 2×2 matrices only takes $\mathcal{O}(1)$ time.

Algorithm: We pick a prime number p in the range *range* and do arithmetic modulo p throughout the rest of the problem. To compute the 2×2 *"hash"*, $H_p(A[*][*])$ of a binary matrix $A_{m \times m}$, we first compute the row-wise *"hash"* $H_p(A[i][*])$ of each row $A[i][*]$. Then, all these *"hashes"* are multiplied together, to obtain $H_p(A[*][*])$, i.e.,

$$H_p(A[*][*]) = \prod_{i=1}^{i=m} H_p(A[i][*]) = \prod_{i=1}^{i=m} \prod_{j=1}^{j=m} M_p(A[i][j])$$

Additionally, we will also require the inverse matrices for efficient calculation (details elaborated in implementation section). We shall only require the inverse matrices in the context of a single element, and in rows of size m. The inverse matrices, $M_p^{-1}(0)$ and $M_p^{-1}(1)$ have already been provided above. Now, to obtain the inverse of the row $A_{m \times m}[i][*]$, we use the following relation:

$$H_p^{-1}(A[i][*]) = \prod_{j=1}^{j=m} M_p^{-1}(A[i][m+1-j])$$

Implementation : To avoid recalculation of $H_p(A[*][*])$ for each contiguous $m \times m$ sub-matrix A of $X_{n \times n}$ (and effective $\mathcal{O}(nm^2)$ runtime), we make use of the inverse matrices. We first tell how to efficiently calculate *"hashes"* of each sub-row of size m in the i^{th} row of $X_{m \times m}$. The idea is the same as part 1 where we compute the *"hash"* of the first m characters ($X[i][1:m]$) of the row, then to compute the *"hash"* of $X[i][2:m+1]$, we pre-multiply by $M_p^{-1}(X[i][1])$, and post-multiply with $M_p(X[i][m+1])$. More generally, for row *hashes*, we have the following relation:

$$H_p(X[i][j+1:j+m]) = M_p^{-1}(X[i][j]) \times H_p(X[i][j:j+m-1]) \times M_p(X[i][j+m])$$

However, we also require the inverses of each *hash* in order to efficiently calculate the *hash* of the whole (sub-)matrix. The inverse matrices are calculate in much the same fashion, except the order of multiplication is reversed, i.e.,

$$\begin{aligned} H_p^{-1}(X[i][1:m]) &= M_p^{-1}(X[i][m]) \times M_p^{-1}(X[i][m-1]) \cdots M_p^{-1}(X[i][1]) \\ &= \prod_{j=1}^{j=m} M_p^{-1}(X[i][m+1-j]) \\ H_p^{-1}(X[i][j+1:j+m]) &= M_p^{-1}(X[i][j+m]) \times H_p^{-1}(X[i][j:j+m-1]) \times M_p(X[i][j]) \end{aligned}$$

These tools allow us to calculate *hashes* (and their inverses) of any row, in time $\mathcal{O}(n)$. Since there are n rows, the calculation of all such *hashes* (and their inverses) takes $\mathcal{O}(n^2)$ time. These are stored in a table to be used later.

Now, each contiguous sub-row of size of m can be treated as a single "unit", and the process used "horizontally" on each element of the matrix, can be repeated on these *units* "vertically". To avoid verbosity, pseudo-code has been provided in the next section.

Pseudocode: An algorithm which takes as input the text X, the pattern Y, and a prime number p, and returns string-based answers (it can easily be modified to return indices) has been described below:

```
Data: X_{n×n}, Y_{m×m}, p                                   // Y is the pattern
H_Y ← I_{2×2}
H_p(i,j) ← I_{2×2}, ∀ i,j ∈ [1, n−m+1]                  // Array of matrices for X
H_p^{-1}(i,j) ← I_{2×2}, ∀ i,j ∈ [1, n−m+1]
        /*** All arithmetic operations are done  mod p  ***/
for i = 1 to m:
    for j = 1 to m:
        H_Y ← H_Y × M_p(Y[i][j])

for i = 1 to n:
    for j = 1 to m:
        H_p(i,1) ← H_p(i,1) × M_p(X[i][j])
        H_p^{-1}(i,1) ← M_p^{-1}(X[i][j]) × H_p^{-1}(i,1)

    for j = 2 to (n−m+1):
        H_p(i,j) ← M_p^{-1}(i,j−1) × H_p(i,j−1) × M_p(X[i][j+m−1])
        H_p^{-1}(i,j) ← M_p^{-1}(X[i][j+m−1]) × H_p^{-1}(i,j−1) × M_p(X[i][j−1])

for j = 1 to (n−m+1):
    H_X ← I_{2×2}                                        // Stores Window's Hash
    for i = 1 to m:
        H_X ← H_X × H_p(i,j)
    if (H_Y = H_X):
        return "Match Found!"
    for i = 2 to (n−m+1):
        H_X ← H_p^{-1}(i−1,j) × H_X × H_p(i+m−1,j)
        if (H_Y = H_X):
            return "Match Found!"

return "No Match!"
```

Analysis: The analysis is more or less similar to part 2 of the same problem, except that there are $m \times m = m^2$ square matrix multiplications taking place. Thus, for the matrices A_p and B_p denoting "hashes" of two (possibly sub-)matrices $R_{m\times m}$ and $S_{m\times m}$, with $R \neq S$, their entries a_{ij} and b_{ij} are bounded above not by F_m this time, but instead F_{m^2}. Thus, we have that

$$\max(\log a_{ij}, \log b_{ij}) \leq \log F_{m^2} \leq \log 2^{m^2} = m^2$$

meaning that the number of divisors of a_{ij} and b_{ij} can at most be m^2.
Now, if we pick a prime number p in the range $[2, t]$, we shall have error probability $\frac{m^2 \log t}{t} < \frac{1}{n^4}$ for the choice of t given below (along with analysis):

$$t = 8n^4m^2 \log(nm) \leq n^7 \tag{1}$$

$$
\begin{aligned}
\log t &= \log 8 + 4\log n + 2\log m + \log\log(nm) \\
&\leq \log n + 4\log n + 5\log m + \log\log(nm) \\
&= 5\log(nm) + \log\log(nm) \\
&= 5\log(nm) + \log(nm) \qquad (\log\log(nm) \leq \log(nm) \text{ as } 1 \leq m \leq n \Rightarrow 1 \leq nm) \\
&= 6\log(nm) \qquad (2)
\end{aligned}
$$

$$
\begin{aligned}
\frac{m^2 \log t}{t} &\leq \frac{m^2 \cdot 6\log(nm)}{8n^4 m^2 \log(nm)} \\
&\leq \frac{6}{8n^4} \\
&\boxed{< \frac{1}{n^4}}
\end{aligned}
$$

Thus, the bound on error probability follows for choice of $p\,(= \mathcal{O}(n^{\mathcal{O}(1)}))$ in the range $[2, 8n^4m^2\log(nm)]$.

Note: Again, the four events $a_{11} = b_{11}, a_{12} = b_{12}, a_{21} = b_{21}, a_{22} = b_{22}$ may or may not be independent. Hence, we have not assumed $\mathbb{P}[a_{11} = b_{11},\ a_{12} = b_{12},\ a_{21} = b_{21},\ a_{22} = b_{22}] = \mathbb{P}[a_{11} = b_{11}] \cdot \mathbb{P}[a_{12} = b_{12}] \cdot \mathbb{P}[a_{21} = b_{21}] \cdot \mathbb{P}[a_{22} = b_{22}]$. Instead, we use a much safer inequality to come up with an upper bound.