1 Pattern Matching with Rolling Hash

In the Pattern Matching problem, the input is a text string $T$ of length $n$ and a pattern string $P$ of length $m < n$. Our goal is to determine if the text has a (consecutive) substring\(^1\) that is exactly equal to the pattern (i.e. $T[i \ldots i + m - 1] = P$ for some $i$).

1. Design a simple $O(mn)$-time algorithm for this problem.

   ```
   Compare $P$ to each length $m$ substring of $T$ starting from index 0 to $n - m$. Return true if any substring matches exactly, and false otherwise.
   
   This algorithm iterates through $O(n)$ substrings of $T$, and each check against $P$ takes $O(m)$, making the algorithm $O(mn)$.
   ```

2. Can we find a more efficient algorithm using hash functions? One naive way to do this is to hash $P$ and every length-$m$ substring of $T$. What is the running time of this solution?

   ```
   Hashing every length-$m$ substring of $T$ takes $O(m)$ for each substring, with a total of $O(n)$ substrings. This overall is still $O(mn)$.
   ```

3. Suppose that we had a universal hash family $H_m$ for length-$m$ strings, where each $h_m \in H_m$ the sum of hashes of characters in the string:

   $$h_m(s) = h(S[0]) + \cdots + h(S[m - 1]).$$  \hspace{1cm} (1)

   Explain how you would use this hash family to solve the pattern matching problem in $O(n)$ time.

   (Hint: the idea is to improve over your naive algorithm by reusing your work.)

   ```
   Each time we hash the next substring, subtract the hash of the character that was removed and add the hash of the character that was added. This takes $O(1)$ for each substring, so the overall runtime becomes $O(n)$.
   ```

4. Unfortunately, a family of “additive” functions like the one in the previous item cannot be universal. Prove it.

   ```
   Consider a 2 character string $S$, and another 2 character string $S'$ with the characters of $S$ in reverse order. For any $h_m \in H_m$ we have $P(h_m(S) = h_m(S')) = 1$ and $S' \neq S$.
   ```

5. The trick is to have a hash function that looks almost like (1): the hash function treats each character of the string is a little differently to circumvent the issue you discovered in the previous part, but

---

\(^1\)In general, subsequences are not assumed to be consecutive, but a substring is defined as a consecutive subsequence.
they’re still related enough that we can use our work. Specifically, we will consider hash functions parameterized by a fixed large prime \( p \), and a random number \( x \) from \( 1, \ldots, p - 1 \):

\[
h_x(S) = \sum_{i=0}^{m-1} S[i] \cdot x^i \pmod{p}.
\]

For fixed pair of strings \( S \neq S' \), the probability over random choice of \( x \) that the hashes are equal is at most \( m/p \), i.e.

\[
\Pr[h_x(S) = h_x(S')] \leq m/p.
\]

(This follows from the fact that a polynomial of degree \((m-1)\) can have at most \( m \) zeros. Do you see why?)

Design a randomized algorithm for solving the pattern matching problem. The algorithm should have worst-case run-time \( O(n) \), but may return the wrong answer with small probability (e.g. \( < 1/n \)). (Assume that addition, subtraction, multiplication, and division modulo \( p \) can be done in \( O(1) \) time.)

Our algorithm uses the same idea from part 3, but applies this polynomial rolling hash function instead. The key insight is that if we have \( h_x(T[k \ldots k + m - 1]) \) then we have

\[
h_x(T[k + 1 \ldots k + m]) = (h_x(T[k \ldots k + m - 1]) - T[k])/x + T[k + m] \cdot x^{m-1} \pmod{p}
\]

**Algorithm 1: PatternMatch\((T, P)\)**

\[
p_h \leftarrow h_x(P)
\]

for all substrings \( s_k \in T \) do

\[
\text{if } k = 0 \text{ then } \text{hash} \leftarrow h_x(s_k)
\]

\[
\text{else } \text{hash} \leftarrow (\text{hash} - s_{k-1}[0])/x + s_k[m] \cdot x^{m-1}
\]

\[
\text{if hash} = p_h \text{ then } \text{return True}
\]

return False

**Runtime:** Computing the hash for a substring from scratch takes \( O(m) \) time. However, we compute the entire hash only for \( P \) and the first substring of \( T \). Remaining hashes requires computing \( x^m \), but we can precompute and store this value. This makes computing hashes for subsequent substrings \( O(1) \).

The algorithm iterates over \( O(n) \) substrings and each iteration is \( O(1) \), making the algorithm \( O(n) \).

6. How would you change your algorithm so that it runs in expected time \( O(n) \), but always return the correct answer?

Modify the algorithm so that whenever the hashes match, before returning “True” it also checks that the pattern \( P \) actually matches to the substring (and if not continue the loop).

Checking takes \( O(m) \) time, and in expectation we would only have to check \( O(n \cdot m/p) \) times. (That’s \( O(n) \) hash comparisons \( \times \) probability \( m/p \) of false positive each hash comparison). When \( p = \Omega(n \cdot m) \), that’s \( O(1) \) checks.

7. Suppose that we had one fixed text \( T \) and many patterns \( P_1, \ldots, P_k \) that we want to search in \( T \). How would you extend your algorithm to this setting?
We can extend our algorithm simply by hashing each of $P_1, \ldots, P_k$ and checking the hash of each substring against this set of hashes.

2 Depth First Search

1. What are all the strongly connected components? (i.e. groups of vertices such that there exists a path between any two vertices in the group)

   \{E\}, \{A, B, C, D, F, G, H, I, J\}

2. Perform DFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the start time and the finish time. Highlight the edges in the tree generated from the search.
3. Perform BFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the discovery order. Highlight the edges in the tree generated from the search.
3 True or False

1. If \((u, v)\) is an edge in an undirected graph and during DFS, \(\text{finish}(v) < \text{finish}(u)\), then \(u\) is an ancestor of \(v\) in the DFS tree.

   **True.** When we do DFS, we store “Visited” nodes in a stack to keep track of the order in which they were visited. Stacks, by nature, have a “last-in first-out” order, meaning the last node you added into the stack will be popped out before any of the nodes before it. Thus, we have a scenario where \(u\) was visited, then \(v\) was visited, then \(v\) was popped, then \(u\) was popped. This makes \(u\) an ancestor of \(v\). The only other scenario is if \(v\) was both visited and popped before \(u\) was visited and popped. However, since there is an edge between \(u\) and \(v\), this scenario would never happen in DFS since you explore all neighbors before popping yourself.

2. In a directed graph, if there is a path from \(u\) to \(v\) and \(\text{start}(u) < \text{start}(v)\) then \(u\) is an ancestor of \(v\) in the DFS tree.

   **False.** Consider the following case:

   ![Graph Diagram]

   Note: notice that part 1 had an edge between \(u\) and \(v\) while this does not specify such an edge (only a path).

4 Bipartite Graphs

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, \(U\) and \(V\) such that every edge \((u, v)\) either connects a vertex from \(U\) to \(V\) or a vertex from \(V\) to \(U\). A graph is bipartite if there is a 2-coloring such that vertices in a set are colored with the same color. In lecture, we saw an algorithm using BFS to determine where a graph is bipartite.

Design an algorithm using DFS to determine whether or not a graph is bipartite.

The algorithm is essentially the same as that of DFS, except at every node we visit, we either color it if it hasn’t been visited before, or check its color if it has been visited before. The rough algorithm is as follows:

1. Start DFS from any node and color it RED
2. Color the next node BLUE
3. Continue coloring each successive node the opposite color until the end of the tree is reached
4. If at any point a current node is the same color as one of its neighbors, then return false
5. If we have to restart DFS because not all nodes have been discovered, we pick a new node and check if all the node’s neighbors are the same color, and if so, start with the opposite color.
6. Once every node has been visited, if we haven’t returned false, then the graph is bipartite.
5 Source Vertices

A source vertex in a graph $G = (V, E)$ is a vertex $v$ such that all other vertices in $G$ can be reached by a path from $v$. Say we have a directed, connected, acyclic graph that has at least one source vertex.

1. Describe a naive algorithm to find a source vertex.

   You can simply perform DFS/BFS on every vertex and find whether we can reach all the vertices from that vertex. This approach takes $O(V(E + V))$ time, which is very inefficient for large graphs.

2. Describe an algorithm that operates in $O(V + E)$ time to find a source vertex.

   If there exists a source vertex (or vertices), then one of the source vertices is the last finished vertex in DFS. (Or a source vertex has the maximum finish time in DFS traversal).

   A vertex is said to be finished in DFS if a recursive call for its DFS is over, i.e., all descendants of the vertex have been visited.

   **How does the above idea work?**

   Let the last finished vertex be $v$. Basically, we need to prove that there cannot be an edge from another vertex $u$ to $v$ if $u$ is not another source vertex (Or there cannot exist a non-source vertex $u$ such that $u \rightarrow v$ is an edge). There can be two possibilities.

   (a) Recursive DFS call is made for $u$ before $v$. If an edge $u \rightarrow v$ exists, then $v$ must have finished before $u$ because $v$ is reachable through $u$ and a vertex finishes after all its descendants.

   (b) Recursive DFS call is made for $v$ before $u$. In this case also, if an edge $u \rightarrow v$ exists, then either $v$ must finish before $u$ (which contradicts our assumption that $v$ is finished at the end) OR $u$ should be reachable from $v$ (which means $u$ is another source vertex).

   **Algorithm :**

   (a) Do DFS traversal of the given graph, restarting at an unvisited vertex if DFS completes with vertices remaining. While doing traversal keep track of last finished vertex ‘$v$‘. This step takes $O(V + E)$ time.

   (b) If there exists a source vertex (or vertices), then $v$ must be one (or one of them). Check if $v$ is a source vertex by doing DFS/BFS from $v$. This step also takes $O(V + E)$ time.