CS 161 (Stanford, Winter 2023) Section 5

1 Pattern matching with a 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. $T[i \dots i+m-1] = P$ for some i).

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

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

   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**.)
4. Unfortunately, a family of "additive" functions like the one in the previous item cannot be universal. Prove why.
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 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 $1, \dots, p-1$:

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

   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?)

[1] In general, *subsequences* are not assumed to be consecutive, but a *substring* is defined as a consecutive subsequence.

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.)

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

7. Suppose that we had one fixed text T and many patterns $P_1, \dots P_k$ that we want to search in T. How would you extend your algorithm to this setting?

2 Graph Algorithms: True or False

Determine if the two statements below are True or False. If a statement is true provide an explanation; if it is false provide a counter-example. You may find useful to draw small graphs to illustrate your examples.

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

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

3 Identifying 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) connects a vertex from U to V or a vertex from V to U. A bipartite graph is possible if the graph coloring is possible using two colors 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 an undirected graph is bipartite.

4 Hashing with Linear Probing

In this problem, we will explore *linear probing*. Suppose we have a hash table H with n buckets, universe $U = \{1, 2, \dots, n\}$, and a *uniformly random* hash functions $h : U \to \{1, 2, \dots, n\}$.

When an element u arrives, we first try to insert into bucket $h(u)$. If this bucket is occupied, we try to insert into $h(u) + 1$, then $h(u) + 2$, and so on (wrapping around after n). If all buckets are occupied, output **Fail** and don't add u anywhere. If we ever find u while doing linear probing, do nothing.

Throughout, suppose that there are $m \leq n$ distinct elements from U being inserted into H. Furthermore, assume that h is chosen *after* all m elements are chosen (that is, an adversary cannot use h to construct their sequence of inserts).

1. Above, we gave an informal algorithm for inserting an element u. Your next task is to give algorithms for **SEARCH(u)** and **DELETE(u)** an element u from the table.

2. In this part, we will analyze the runtime of linear probing, assuming no deletions occur.

   (a) Give an upper bound on the probability that $h(u) = h(v)$ for some u, v that are a part of these first m elements, assuming that $m = n^{1/3}$.

   Hint: You may need that for any $x > 0$, $(1-x)^n \geq 1 - nx$.

   (b) When inserting an element, define the number of *probes* it does as the number of buckets it has to check, including the first empty bucket it looks at. For example, if $h(u), \ldots, h(u) + 10$ were occupied but $h(u) + 11$ was not then we would have to check 12 buckets.

   Prove that the expected number of total probes done when inserting $m = n^{1/3}$ elements is $O(m)$.