CS 161 (Stanford, Winter 2025) 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 that doesn't use any hashing.

   Solution

   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. Assume that we have access to a hash function for hashing individual characters. What is the running time of this solution?

   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:

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

   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**.

   Solution

   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 $O(n)$.

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

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

Solution

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 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.)

Solution

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 ← h_x(p)
for all substrings s_k ∈ T do
    if k = 0 then
        hash ← h_x(s_k)
    else
        hash ← (hash − s_{k−1}[0])/x + s_k[m] · x^{m−1}
    if hash = p_h then
        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 hash for each subsequent substring $O(1)$, 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?

Solution

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?

Solution

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 Hash Tables with Linear Probing

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

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. (Warmup) Can we ever output **Fail** while inserting these m elements?

   Solution

   No, as there are n spots in the table.

2. Above, we gave an informal algorithm for inserting an element u. Your next task is to give algorithms for searching and deleting an element u from the table.

   Hint: Make sure that your search and delete algorithms work together - specifically, what should happen when you search after deleting an item?

   Solution

   When deleting, we should take special care that a previously occupied bucket is still marked as "previously occupied". Here's why: suppose $n = 3$ and $h(0) = 0, h(1) = 1, h(2) = 0$. Suppose elements were inserted in the order $0, 1, 2$. Then, $H = [0, 1, 2]$. What happens if we delete 1 and then search for 2? Well, after deleting 1, $H = [0, \square, 2]$ and so naively searching for 2 would return false, as the spot after 0 is empty.
   To get around this, we mark such deletions with a "tombstone" value so that search treats those as elements.

```
def Search(H, u):
    start = h(u)
    if H[start] == u:
        return True
    current = start + 1
    while current != start:
        if H[current] == u:
            return True
        elif H[current] == empty: # X is not empty!
            return False
        current = current % n + 1 #adds 1 mod n

def Delete(H, u):
    start = h(u)
    if H[start] == u:
        H[start] = X (tombstone)
        return
```

```
    current = start + 1
    while current != start:
        if H[current] == u:
            H[current] = X
            return
        elif H[current] == empty:
            return # u is not in H
        current = current % n + 1 #adds 1 mod n
```

3. In this part, we will analyze the expected runtime of linear probing assuming that $m = n^{1/3}$ and that 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 $\mathbf{P}[\textit{at least one of } E_1, \ldots, E_k \textit{ happens}] \leq \sum_{i \in [k]} \mathbf{P}[E_i]$ *given any random events* $E_1, \ldots, E_k$.

   Solution

   Number the elements $u_1, u_2, \ldots, u_m$. The probability that any pair of elements collide with each other is $\frac{1}{n}$, and hence, by union bound over all possible pairs, an upper bound on the probability of any collision is $\frac{m^2}{n} = n^{-1/3}$.

   (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 $h(u), \ldots, h(u)+2$ were occupied but $h(u)+3$ was not then we would have to check 4 buckets.

   Prove that the expected number of total probes done when inserting $m = n^{1/3}$ elements is $O(m)$.
   Hint: Consider using the law of total expectation with two cases: when there are no collisions, and when there is at least one collision.

   Solution

   Let X be a random variable corresponding to the number of total probes done, and notice that $X \leq m^2$ (each inserted element can only be compared against the other inserted elements' positions, so we have m elements with $\leq m$ probes each).
   Furthermore, let E be the event that there is at least one collision. Then, by conditional expectation:

   $$\mathbb{E}[X] = \mathbb{E}[X \mid E]\mathbf{P}[E] + \mathbb{E}[X \mid \overline{E}]\mathbf{P}[\overline{E}] \leq \mathbb{E}[X \mid E] \cdot \frac{m^2}{n} + \mathbb{E}[X \mid \overline{E}] \leq \frac{m^4}{n} + \mathbb{E}[X \mid \overline{E}]$$

   We upperbounded the probability of E by $\frac{m^2}{n}$ and the probability of $\overline{E}$ (the

complement of E) by 1, since probabilities are always ≤ 1. Then, we applied $\mathbb{E}[X \mid E] \leq m^2$. Finally, if there are no collisions, the total number of probes done is m: we only have to check one bucket per insertion. Hence, overall we have $\mathbb{E}[X] \leq \frac{m^4}{n} + m = 2m = O(m)$ (remembering that $n = m^3$).

3 True or False

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.

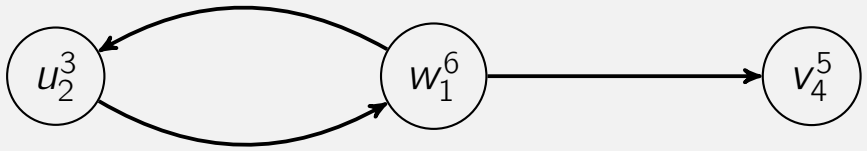
   Solution

   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 $start(u) < start(v)$ then u is an ancestor of v in the DFS tree.

   Solution

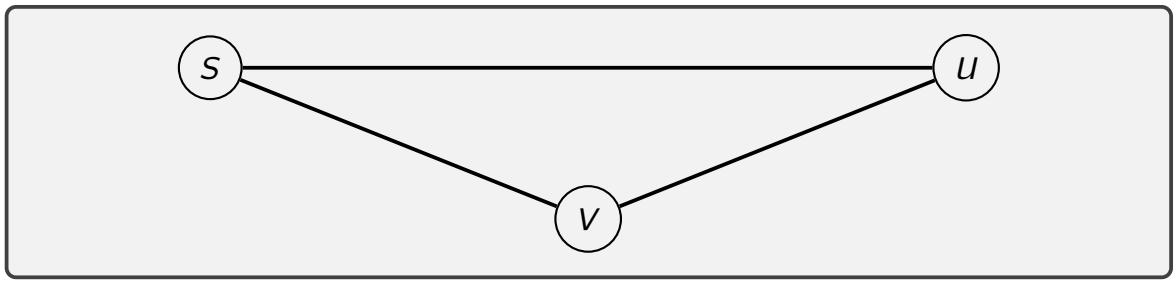
   False. Consider the following case:

3. In an undirected graph, if (u, v) is an edge and u is marked as visited before v during BFS, then u must be closer to the starting node s than v.

   Solution

   False.
   The order of node discovery in BFS depends on the queue processing, and marking an adjacent node as visited does not necessarily imply its distance from the start node. Consider the following case:



4 Matryoshka Dolls

You have n matryoshka dolls. The i-th doll has dimensions $w_i \times h_i$. Doll i can fit inside doll j iff $w_i < w_j$ and $h_i < h_j$. A sequence of dolls $b_1, b_2, ..., b_k$ form a chain if doll b_i fits inside doll b_{i+1} for each $1 \leq i < k$. Design an algorithm which takes as input a list of dimensions $w_i \times h_i$ and returns a longest possible chain of dolls. You must construct a directed graph as part of your solution.

Solution

Construct a directed graph whose vertices are dolls, and such that there is an edge (v_i, v_j) iff doll v_i fits inside doll v_j. Notice that this graph is a DAG (you can only go one direction between dolls). Our goal is now to find the longest path.
Linearize (topologically sort) the graph, so whenever there is an edge from v_j to v_i , $j < i$.
For every node v_i, let ℓ_i be the length of the longest path ending at v_i. We can compute ℓ_i as follows:

$$\ell_i = 1 + \max_{(v_j, v_i) \in E} \ell_j$$

Because we have linearized the graph, ℓ_i depends only on ℓ_j for $j < i$. So we can compute the ℓ_i values in order.
The answer is $\max_{i=1}^{n} \ell_i$.

5 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.

Solution

The algorithm is essentially the same as that of BFS, 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. Once every node has been visited, if we haven't returned false, then the graph is bipartite