1.5em 0pt
CS 218, Spring 2025, Due: 11:59pm, Fri 05/23, 2025
Assignment #4 [1] Basic Programming Due: 11:59pm, Fri 05/16, 2025
Name: Lakhan Kumar Sunilkumar Student ID: 862481700

---

1 (2.5 pt) Being Unique

1. (0.5pts)

   Let us consider the following,

   Let there exists two integers $m' < m$ and some binary string

   $$t \in \{0,1\}^{m'}$$

   such that t does *not* occur as a contiguous substring of S.

   Now consider any extension

   $$u \in \{0,1\}^{m-m'}$$

   Consider the length-m string

   $$w \;=\; t \,\|\, u, \qquad \textit{where} \quad \| \quad \textit{denotes} \quad \textit{concatenation}.$$

   As we just assumed that m is *not* feasible, *every* binary string of length m, including w, *does* occur as a substring of S. Hence there exists an index i with

   $$S[i..\,i+m-1] \;=\; w \;=\; t\,u.$$

   But then the prefix

   $$S[i..\,i+m'-1]$$

   of that substring equals t, contradicting the assumption that t never appears in S. Therefore no such $m' < m$ can be feasible, and the non-feasibility property is monotonic.

2. (0.5pts)

   Let $n = |S|$. For any integer L, the string S has at most

   $$n - L + 1$$

   distinct substrings of length L (one for each starting position).
   However, there are

   $$2^L$$

   possible binary strings of length L.
   If

   $$2^L > n - L + 1,$$

   then by the pigeonhole principle at least one binary string of length L does *not* appear in S, so L is feasible and thus

   $$N_{\text{opt}} \le L.$$

   In particular, since

   $$n - L + 1 \le n,$$

[1] Some of the problems are adapted from existing problems from online sources. Thanks to the original authors.

it suffices to choose L so that

$$2^L > n,$$

i.e.

$$L > \log_2 n.$$

Hence the tightest simple bound is

$$N_{\text{opt}} \;\le\; \lfloor \log_2 n \rfloor + 1.$$

Also, one may observe that for

$$L = \lceil \log_2(n+1) \rceil,$$

we have

$$2^L \ge n+1 > n \ge n - L + 1,$$

so again $N_{\text{opt}} \le \lceil \log_2(n+1) \rceil$. This is the tightest upper bound in terms of n.

3. (1.5pts)

Algorithm (Binary-search over L + rolling hashes).
Let $n = |S|$. We know from part 2 that

$$1 \le N_{\text{opt}} \le M \quad \text{where} \quad M = \lceil \log_2(n+1) \rceil.$$

We perform a binary search on the interval $[1, M]$, testing feasibility of a candidate length L in $O(n)$ time using rolling hashing.

(a) **Precompute.** Choose two large moduli p_1, p_2 and a base B. In $O(n)$ time compute

$$H_{1,i} = \sum_{k=0}^{i-1} S[k]\, B^k \bmod p_1, \quad H_{2,i} = \sum_{k=0}^{i-1} S[k]\, B^k \bmod p_2,$$

and the powers B^i mod each modulus for $i = 0, \ldots, n$.

(b) **Binary search.** Initialize $\ell = 1$, $r = M$.

$$\textbf{while } \ell < r: \quad \begin{cases} m \;=\; \lfloor (\ell + r)/2 \rfloor, \\ \textbf{if } \text{Feasible}(m) \text{ then } r = m, \\ \textbf{else } \ell = m + 1. \end{cases}$$

Return $N_{\text{opt}} = \ell$.

(c) **Feasible**(L)**.**

i. For each $i = 1, \ldots, n - L + 1$, compute the double-hash

$$h_i = \big(H_{1,i+L-1} - H_{1,i-1}\, B^L\big) \bmod p_1,\ \big(H_{2,i+L-1} - H_{2,i-1}\, B^L\big) \bmod p_2.$$

ii. Insert each $(h_i^{(1)}, h_i^{(2)})$ into an unordered_set.

iii. Return $\big|\text{set}\big| < 2^L$.

Time Complexity.

- Precomputation of hashes and powers: $O(n)$.
- Each call Feasible(L) scans $n - L + 1 \le n$ positions, does $O(1)$ work per position (two modular subtractions, two lookups/insertions), for $O(n)$.
- Binary search over $[1, M]$ takes $O(\log M) = O(\log \log n)$ iterations.

Hence the total running time is

$$O\big(n + (\log \log n) \cdot n\big) \;=\; O\big(n \log \log n\big).$$

2 (3 pts) Weight-balanced trees

1. (0.3pts)

   Let $S(u)$ denote the number of nodes in the subtree rooted at u. By definition of a subtree,

   $$S(u) \;=\; 1 \;+\; S\big(\mathrm{lc}(u)\big) \;+\; S\big(\mathrm{rc}(u)\big).$$

   Since the weight is defined as

   $$w(u) \;=\; 1 \;+\; S(u),$$

   we have

   $$w(u) = 1 + \big(1 + S(\mathrm{lc}(u)) + S(\mathrm{rc}(u))\big) = (1 + S(\mathrm{lc}(u))) + (1 + S(\mathrm{rc}(u))) = w\big(\mathrm{lc}(u)\big) \;+\; w\big(\mathrm{rc}(u)\big).$$

   Hence for every node u, $w(u) = w(\mathrm{lc}(u)) + w(\mathrm{rc}(u))$.

2. (0.4pts)

   Let the height of the tree be h, and consider a longest root-to-leaf path

   $$u_0, u_1, \ldots, u_h,$$

   where u_0 is the root and u_h is a leaf. By the weight-balance condition, for each $i = 0, 1, \ldots, h-1$,

   $$w(u_{i+1}) \;\le\; (1-\alpha)\, w(u_i).$$

   Further Computing,

   $$w(u_h) \;\le\; (1-\alpha)^h\, w(u_0).$$

   Since u_h is a (non-empty) leaf its weight satisfies $w(u_h) \ge 1$, and the root has weight

   $$w(u_0) \;=\; 1 + \big|\text{all nodes}\big| \;=\; n+1.$$

   Hence

   $$1 \;\le\; (1-\alpha)^h\,(n+1) \quad\Longrightarrow\quad (1-\alpha)^h \;\ge\; \frac{1}{n+1}.$$

   Taking logarithms (noting $0 < 1-\alpha < 1$ so $\log(1-\alpha) < 0$) gives

   $$h \;\le\; \frac{\ln\!\left(\frac{1}{n+1}\right)}{\ln(1-\alpha)} \;=\; \frac{-\ln(n+1)}{-\ln(1-\alpha)} \;=\; O(\ln n).$$

   Since $\ln n$ and $\log_2 n$ differ by a constant factor, the height is $O(\log n)$.

3. (a) (0.6pts)

   Let the size of the subtree be n'. We perform two steps:

   1. Flattening. Do an in-order traversal of the subtree rooted at t, appending each visited node to an array A. Since each of the n' nodes is visited exactly once, this step takes

   $$\Theta(n').$$

   2. Rebuilding. Given the sorted array $A[1..n']$, we build a perfectly balanced BST by the standard "divide-and-conquer" routine:

   $$\mathrm{Build}(i,j): \begin{cases} \text{if } i > j \text{ return } \texttt{null}, \\ m = \lfloor (i+j)/2 \rfloor, \\ \text{let } u \leftarrow \text{new node with key } A[m], \\ u.\mathrm{lc} = \mathrm{Build}(i, m-1), \quad u.\mathrm{rc} = \mathrm{Build}(m+1, j), \\ \text{return } u. \end{cases}$$

The recurrence for its running time $T(k)$ on a segment of length k is

$$T(1) = \Theta(1), \quad T(k) = 2\,T\big(\tfrac{k}{2}\big) + \Theta(1).$$

By the Master Theorem, $T(k) = \Theta(k)$. Hence rebuilding from $A[1..n']$ costs

$$\Theta(n').$$

Total cost. Adding the two steps,

$$\Theta(n') + \Theta(n') \;=\; \Theta(n').$$

Therefore, the optimal time to rebuild an (unbalanced) subtree of size n' into a perfectly balanced one is $\Theta(n')$.

(b) (1.7pts)

We'll apply a common technique called the *token-charging* (or potential method) to reason about the cost. Whenever we insert a new key into the tree, we follow the normal BST insertion procedure, which takes $O(h)$ time where h is the depth of the node being inserted.

As we walk down this path of length h, we imagine leaving one "token" at every node we pass. These tokens act like stored credits for future work. Because, as shown in part 2, the height of the tree is at most $O(\log n)$, we'll end up placing at most $O(\log n)$ tokens during this process.

$$O(h) = O(\log n)$$

After insertion, we return to the root looking for the *lowest* ancestor u whose weight-balance invariant will fails if no such u exists, the insertion is done.

$$\alpha < \frac{w(\mathrm{lc}(u))}{w(u)} < 1 - \alpha$$

Otherwise, let $n' = w(u)$; we *rebuild* the entire subtree at u in time $\Theta(n')$ (by part 3(a)).
We now show that the subtree at u has accumulated $\Theta(n')$ tokens since its last rebuild, so those tokens can pay for the $\Theta(n')$ rebuilding cost.

Right after a rebuild at u, both children satisfy

$$w(\mathrm{lc}(u)) \le (1-\alpha)\,w(u), \quad w(\mathrm{rc}(u)) \le (1-\alpha)\,w(u).$$

Thereafter, each new insertion into the subtree of u increments $w(u)$ by 1 (and increments exactly one of its children by 1). To reach a violation of the form $w(\mathrm{child}(u)) > (1-\alpha)\,w(u)$, the larger child must have gained more than

$$(1-\alpha)\,w(u) \;-\; (1-\alpha)\big(w(u)-1\big) \;=\; \alpha\,w(u) \;-\; (1-\alpha) \;=\; \Theta(n')$$

new nodes. Thus at least $\Theta(n')$ insertions have occurred *into* the subtree of u since its last rebuild. Each such insertion deposited exactly one token at u. Hence at rebuild time, u has accrued $\Theta(n')$ tokens, which we now *spend* to pay the $\Theta(n')$ cost of rebuilding.

Amortized cost. Each insertion does

$$\underbrace{O(\log n)}_{\substack{\text{search \&}\\ \text{token deposit}}} \;+\; \underbrace{O(1)}_{\substack{\text{possible}\\ \text{rebuild charge}}} \;=\; O(\log n)$$

amortized time. Thus although a single insertion might rebuild a large subtree in $O(n)$ worst-case time, the *amortized* cost per insertion is $O(\log n)$, matching classic balanced-tree schemes.