1 Overview

- Rolling Hash
- Sorting
- Master Theorem
- Universal Hashing

2 Rolling Hash

Idea: Hash functions can be related!

Example: Hashing strings “the” and “her”

Converting to numbers:

“the” = (t · (26)^2 + h · (26) + e)
“her” = (h · (26)^2 + e · (26) + r) = 26(“the” - t) + r

In general: Converting to base-b numbers using:

N(S) = S_0 b^L + S_1 b^{L-1} + S_2 b^{L-2} + ... + S_{L-1} b + S_L

Given S and S' = S_{0:L} and S'' = S_{n:L+M+n}

N(S'') = b^{M+n} (N(S') - b^{L-n} N(S'_{0:n})) + N(S''_{L+1:L+n+M})

Mod properties:

ab mod m = ((a mod m)(b mod m)) mod m
(a + b) mod m = ((a mod m) + (b mod m)) mod m
h_m(S) = N(S) mod m = (((S_0 mod m)(b^L mod m)) mod m)+...+S_L mod m) mod m

h_m(S'') = N(S'') mod m
= (b^{M+n} h_m(S') - b^{L-n} h_m(S'_{0:n}) + h_m(S''_{L+1:L+n+M})) mod m

Just store division hash!

One character move:

(b(h_m(S') - b^{L-1} h_m(S'_0)) + h_m(S''_{L+1})) mod m

Constant time hash calculation!
3 Sorting

Idea: Given list of numbers, sort them from smallest to largest.

MERGE SORT

1. One element, done
2. Merge-Sort(\(A[1 : n/2]\))
3. Merge-Sort(\(A[n/2 + 1 : n]\))
4. Merge two arrays

Two-Finger Algorithm

Idea: One finger in each list. Advance finger on smaller element.

Example:

\[
\begin{array}{cccccc}
1 & 2 & 5 & 3 & 19 & 18 \\
& & 21 & 25 & & \\
1 & 2 & 3 & 5 & 18 & 19 & 21 & 25
\end{array}
\]

Time: \(O(n)\) since you only touch each element once

Space: If you create a new array each time \(n \log n\) but can be done in place (complicated)

Best Case: \(O(n)\) if already sorted (yay good!)

4 Master Theorem

IDEA: Used to solve running time for recurrence relations. Like Merge Sort.

\[ T(n) = 2T(n/2) + O(n) \]

General form: \( T = aT(n/b) + f(n) \)

Think of recurrence as tree:

Height: \(\log_b(n)\)

Number of leaves: \(a^{\log_b(n)}\)
LOG PROPERTY:

\[ a^{\log_b(n)} = n^{\log_a(a)} \]

\[ \log_b(n) = \log_b(a^{\log_a(n)}) = \log_a(n) \log_b(a) \]

\[ \log_b(x^y) = y \log_b(x) \] because \( \log_b(x^y) \) is the number we must raise \( b \) to to get \( x^y \) and \( b^\log_b(x) = x^y \).

\[ a^{\log_a(n)} = (a^{\log_a(n)})^{\log_a(a)} = n^{\log_a(a)} \]

What is the work done?

That depends on what the work per level looks like.

We KNOW we do \( O(f(n)) \) work and \( O(a^{\log_a(n)}) \) work. Question: Which dominates?

CASES:

1. Leaves dominate. Implies that each level does an order of magnitude less work than the level below it. This is true when \( f(n) = O(n^{\log_a(a)-\epsilon}) \):

   Note: Clearly top level does order of magnitude less work than leaves.

   At level \( i \): \( a^i \) nodes do \( f(n/(b^i)) \) work

   \[ = a^i O((n \cdot b^{-i})^{\log_a(a)}) = a^i O(n^{\log_a(a) - \epsilon b^{-i \log_a(a)}}) \]

   \[ = a^i O(n^{\log_a(a) - \epsilon b^i / a^i}) \]

   \[ = O(n^{\log_a(a) - \epsilon i b^i}) \] (1)

So total work is

\[ O(n^{\log_a(a) - \epsilon}) + O(n^{\log_a(a) - \epsilon b^i}) + O(n^{\log_a(a) - \epsilon b^{2i}}) + ... + O(n^{\log_a(a) - \epsilon b^{\log_a(n) \epsilon}}) \]

\[ = O(n^{\log_a(a) - \epsilon n^\epsilon}) \]

\[ = O(n^{\log_a(a)}) \] (2)

2. Root node dominates. Implies that each level does order of magnitude less work than level below it. NOTE: third case from class

   Let \( f = O(n^{\log_a(a)+\epsilon}) \).

   Work at level \( i \) is:

   \[ a^i O(n^{\log_a(a)+\epsilon} b^{-i \log_a(a) - i \epsilon}) \]

   \[ = O(n^{\log_a(a)+\epsilon} b^{-i \epsilon}) \] (3)

Total work is

\[ O(n^{\log_a(a)+\epsilon}) + O(n^{\log_a(a)+\epsilon} b^{-\epsilon}) + ... + O(n^{\log_a(a)}) \]

\[ = O(n^{\log_a(a)+\epsilon}) = f(n) \] (4)
3. What if \( f(n) = O(n^{\log_b(a) \log^k(n)}) \)?

Why \( \log^k(n) \)? Because a log is the largest order of magnitude function that cannot be expressed as \( n^c \) and we’ve covered that case.

At level \( i \) work

\[
= a^i O(n^{\log_b(a) b^{-i \log_b(a)} \log^k(n/b^i)}) \\
= O(n^{\log_b(a) \log^k(n/b^i)}) \quad (5)
\]

Total work:

\[
= O(n^{\log_b(a) \log^k(n)}) + O(n^{\log_b(a) \log^k(n/b)}) + \ldots + O(n^{\log_b(a)}) \\
= O(\text{treeheight} \cdot n^{\log_b(a) \log^k(n)}) \\
= O(n^{\log_b(a) \log^{k+1}(n)}) \\
= \log(n) f(n) \quad (6)
\]

NOTE: Changing bases in a log is just multiplying by a constant:

\[
\log_b(x) = \log_c(x)/\log_c(b)
\]

EXAMPLES:

- **MergeSort:**
  \[ T(n) = 2T(n/2) + O(n) \]
  \( a = 2, b = 2, n^{\log_b(a)} = n \) Case \( f(n) = O(n^{\log_b(a)}) \). Work is \( n \log n \).

- **T(n) = 8T(n/2) + O(n^2)**
  \( a = 8, b = 2, n^{\log_b(a)} = n^3 \) Case \( f(n) < O(n^{\log_b(a)}) \). Work is \( n^3 \).

- **T(n) = 3T(n/2) + n \log n** Case \( f(n) > O(n^{\log_b(a)}) \). Work is \( n \log n \).

- \( 2^n T(n/2) + n^n \) can’t be solved. \( a \) is not constant!

- \( 0.5 T(n/2) + n \) doesn’t have a recursion.

5 Universal Hashing

**Definition:** A family of hash functions \( H = \{h_0, h_1, \ldots\} \) is *universal* if, for a randomly chosen pair of keys \( k, l \in U \) and randomly chosen hash function \( h \in H \), the probability that \( h(k) = h(l) \) is not more than \( 1/m \) where \( m \) is the size of the hash table.
This is useful because if you pick a hash function from $H$ when your program begins in such a way that an adversary cannot know in advance which function you will pick, the adversary cannot in advance guess two keys that will map to the same value.

**Example:** The family of hash functions

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

where $0 < a < p$, $b < p$, $m < p$, and $|U| < p$ for prime $p$ is universal.

**Proof:** Consider $k,l \in U$ with $k \neq l$. For a given $h_{a,b}$ let

$$r = (ak + b) \mod p$$
$$s = (al + b) \mod p$$

Note that $r \neq s$ since

$$r - s \equiv a(k - l) \mod p$$

cannot be zero since $0 < a < p$, $k < p$, and $l < p$ so $a(k - l)$ cannot be a multiple of $p$.

Now consider

$$a = ((r - s)((k - l)^{-1} \mod p)) \mod p$$
$$b = (r - ak) \mod p.$$  \hspace{1cm} (10)

Now since $r \neq s$, there are only $p(p - 1)$ possible pairs $(r, s)$. Similarly, since we require $a \neq 0$, there are only $p(p - 1)$ pairs $(a, b)$. Equations 10 and 10 give a one-to-one map between pairs $(r, s)$ and pairs $(a, b)$. Therefore, each choice of $(a, b)$ must produce a different $(r, s)$ pair. If we pick $(a, b)$ uniformly, at random then $(r, s)$ is also distributed uniformly at random.

The probability that two keys $k$ and $l$ with $k \neq l$ have the same hash value is the probability that $r \equiv s \mod m$. Therefore, we must have that

$$r - s \in \{m, 2m, ...,qm\}$$  \hspace{1cm} (11)

where $qm < p$. This gives us at most $[p/m] - 1 \leq (p - 1)/m$ possible values for $s$ such that $s$ can collide with $r$. Since the pairs are distributed at random, and $s \neq r$, we have $p - 1$ values for $s$ that are all equally probable. Thus

$$Pr[s \equiv r \mod m] = \frac{p - 1/m}{p - 1} = \frac{1}{m}$$

$$\Rightarrow Pr[h(k) = h(l)] = \frac{1}{m}$$  \hspace{1cm} (12)

This proof was taken from CLRS Section 11.3.3.