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" $= (t \cdot (26)^2 + h \cdot (26) + e)$

"her" $= (h \cdot (26)^2 + e \cdot (26) + r) = 26(\text{"the"} - t) + r$

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:

$ab \bmod m = ((a \bmod m)(b \bmod m)) \bmod m$

$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$

$h_m(S) = N(S) \bmod m = ((((S_0 \bmod m)(b^L \bmod m)) \bmod m) + ... + S_L \bmod m) \bmod m$

$$\begin{aligned} h_m(S'') &= N(S'') \bmod m \\ &= (b^{M+n}(h_m(S') - b^{L-n} h_m(S'_{0:n})) + h_m(S''_{L+1:L+M})) \bmod m \end{aligned}$$

Just store division hash!

One character move:

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

Constant time hash calculation!

Can be used for string matching (Rabin-Karp):

Given string S and text T

- Compute $h_m(S)$
- Compute hash for each string of length L in T
- If hash $= h_m(S)$, compare strings character-by-character $O(L)$

Time: $O(|S| + |T| - |S| + |S|c) = O(|T| + |S|c)$

Using signatures, c is $1/|T|$.

3 Sorting

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

Algorithms: INSERTION SORT (ONLY IF NECESSARY)

DIAGRAM.

LOOP PROPERTY: At iteration j, we have an array of $j-1$ sorted elements.

We put the jth element of the original list into the array in the correct place, creating an array of j sorted elements.

SPACE: $O(n)$ This can be done in place.

TIME: $O(n^2)$

EXAMPLE:

1. 9, 8, 7, 6, 5
2. 8, 9, 7, 6, 5
3. 7, 8, 9, 6, 5
4. 6, 7, 8, 9
5. 5, 6, 7, 8, 9

At each step j you must look through $j-1$ elements:

$\sum_{j=0}^{n-1} j = (n-1)(n-2)/2 = O(n^2)$

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 (If necessary ONLY)

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

Example:

1 2

5 3

19 18

21 25

1 2 3 5 18 19 21 25

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: $T = aT(n/b) + f(n)$

DIAGRAM.

Height: $\log_b(n)$

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

LOG PROPERTY:

$a^{\log_b(n)} = n^{\log_b(a)}$

$\log_b(n) = \log_b(a^{\log_a(n)}) = \log_a(n) \log_b(a)$

$log_b(x^y) = ylog_b(x)$ because $log_b(x^y)$ is the number we must raise b to to get x^y and $b^{y \log_b(x)} = x^y$.

$a^{\log_b(n)} = \left(a^{\log_a(n)}\right)^{\log_b(a)} = n^{\log_b(a)}$

What is the work done?

That depends on what the work per level looks like.

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

CASES: SHOW IN DIAGRAM!!

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

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

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

$$
\begin{aligned}
&= a^i O((n * b^{-i})^{\log_b(a)-\epsilon} = a^i O(n^{\log_b(a)-\epsilon} b^{-ilog_b(a)+\epsilon}) &(1)\\
&= a^i O(n^{\log_b(a)-\epsilon} b^{i\epsilon}/a^i) &(2)\\
&= O(n^{\log_b(a)-\epsilon} b^{i\epsilon} &(3)
\end{aligned}
$$

   So total work is

$$
\begin{aligned}
& O(n^{\log_b(a)-\epsilon}) + O(n^{\log_b(a)-\epsilon} b^{\epsilon}) + O(n^{\log_b(a)-\epsilon} b^{2\epsilon} + ... + O(n^{\log_b(a)-\epsilon} b^{\log_b(n)\epsilon}) &(4)\\
=\ & O(n^{\log_b(a)-\epsilon} n^{\epsilon}) &(5)\\
=\ & O(n^{\log_b(a)}) &(6)
\end{aligned}
$$

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

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

   Work at level i is:

$$
\begin{aligned}
& a^i O(n^{\log_b(a)+\epsilon} b^{-i\log_b(a)-i\epsilon} &(7)\\
=\ & O(n^{\log_b(a)+\epsilon} b^{-i\epsilon}) &(8)
\end{aligned}
$$

   Total work is

$$
O(n^{\log_b(a)+\epsilon}) + O(n^{\log_b(a)+\epsilon} b^{-\epsilon}) + ... + O(n^{\log_b(a)}) \quad = \quad O(n^{\log_b(a)+\epsilon}) = f(n) \qquad (9)
$$

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^{ϵ} and we've covered that case.

   At level i work

$$
\begin{aligned}
&= a^i O(n^{\log_b(a)} b^{-i\log_b(a)} \log^k(n/b^i)) &(10)\\
&= O(n^{\log_b(a)} \log^k(n/b^i)) &(11)
\end{aligned}
$$

Total work:

$$
\begin{aligned}
&= O(n^{\log_b(a)} \log^k(n)) + O(n^{\log_b(a)} \log^k(n/b)) + ... + O(n^{\log_b(a)}) && (12)\\
&= O(treeheight \cdot n^{\log_b(a)} \log^k(n)) && (13)\\
&= O(\log_b(n) n^{\log_b(a)} \log^k(n)) && (14)\\
&= O(n^{\log_b(a)} \log^{k+1}(n)) && (15)\\
&= \log(n) f(n) && (16)
\end{aligned}
$$

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!
- $0.5T(n/2) + n$ doesn't have a recursion.

5 Universal Hashing

Definition: A family of hash functions $H = \{h_0, h_1, ...\}$ 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

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m \tag{17}$$

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

$$r = (ak + b) \bmod p \tag{18}$$
$$s = (al + b) \bmod p \tag{19}$$

Note that $r \neq s$ since

$$r - s \equiv a(k - l) \bmod p \tag{20}$$

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

Now consider

$$a = ((r - s)((k - l)^{-1} \bmod p)) \bmod p \tag{21}$$
$$b = (r - ak) \bmod p. \tag{22}$$

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 21 and 22 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 $r \equiv s \bmod m$. Therefore, we must have that

$$r - s \in \{m, 2m, ..., qm\} \tag{23}$$

where $qm < p$. This gives us at most $\lceil p/m \rceil - 1 \leq (p-1)/m$ 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

$$Pr[s \equiv r \bmod m] = \frac{p - 1/m}{p - 1} = \frac{1}{m} \tag{24}$$
$$\Rightarrow \quad Pr[h(k) = h(l)] = \frac{1}{m} \tag{25}$$

This proof was taken from CLRS Section 11.3.3.