String search algorithms

Key-indexed counting

> Basis for more complex algorithms, specialized sorting which works best with the following conditions

- Input consists of a collection of n items.
- Maximum possible value of each of the individual item is

ASSUMPTION: Keys are integers between - and R - 1

IMPLICATION: Can use key as an array index.

APPLICATIONS:

- Sort string by first letter
- Sort class roster by section
- Sort phone numbers by area.
- Subroutine in a sorting algo.

```
int N = a.length
int[] count = new int[R+1];
//NOTE Count the frequencies of each letter using key as index(offset by 1)
for(int i = 0; i < N; i++)
    count[a[i]+1]++;

//NOTE Compute frequency cumulates which specify destinations
for(int r = 0; r < R; r++)
    count[r+1] += count[r];


//NOTE Access cumulates using key as index to move items
for(int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

//NOTE Copy back to original array.
for(int i = 0; i < N; i++)
    a[i] = aux[i];
```

Radix Sorts - LSD and MSD

Radix sorts

> Non-Comparative integer sorting algorithm Sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

LSD radix key sort

- Short keys come before longer keys
- Keys of the same length are sorted lexicographically
- Normal order of integer representations such as 1,2,3,4,5,6,7....

MSD radix sort

- Lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations.
- A sequence such as "b,c,d,e,f,g,h,i,j,ba" would be lexicographically sorted as "b,ba,c,d,e,f,g,h,i,j"
- (1 to 10 would be output as 1,10,2,3,4,5,6,7,8,9)

LSD sort

PROPOSITION: LSD sorts fixed-length strings in ascending order.

PF. [by induction on i]

After pass i, strings are sorted by last i characters.

- If two strings differ on sort key, key-indexed sort puts them in proper relative 1.
- If two strings agree on sort key, stability keeps them in proper relative 1. first

```
Public class LSD {
    Public static void sort(String[] a, int W) {
        int R = 256;
        int N = a.length;
        String[] aux = new String[N];

        for(int d = W -1 d >= 0; d--){
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;
            for (int i = 0; i < R; r++)
                count[r+1] += count[r];
            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]++] = a[i];
            for (int i = 0; i < N; i++)
                a[i] = aux[i];
        }
    }
}
```

MSD - Sort by most significant Digit first

> Similar to quicksort

Partition array into R (radix) pieces according to the first character (most significant digit) using key-indexed counting. Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort)

Boyer - Moore

When a character not in the pattern is found skip up to M characters (where M is less than the length of the pattern) (no need to loop through characters)

```
- Uses mismatched character heuristic.
- Don't look through characters in order, start from the back and look at the last character in t
e pattern first to see if its a match.
- Uses backup.
```

```java
public int search(String txt){
    int N = txt.length();
    int M = pat.length();
    int skip;
    for(int i = 0; i < N-M; i += skip){
        skip = 0;
        for(int j = M-1; j >= 0; j--){
            if(pat.charAt(j) != txt.charAt(i+j)){
                //NOTE max between 1 and positive int incase of a negative skip result
                skip = Math.max(1,j-right[txt.charAt(i+j)]);
                //Compute skip value

                break;
            }
        }
        if(skip == 0) return i;
    }
    return N;
}
```

For each mismatched character increment i (the skip) by one

Mismatched character heuristic

Q. How much to skip

A. Pre compute index of rightmost occurrence of character c in pattern (-1 if not in pattern);

```java
//Position of the rightmost occurrence of c in the pattern
right = new int[R]
for(int c = 0; c < R; c++)
    right[c] = -1;
for(int j = 0; j < M; j++)
    right[pat.charAt(j)] = j;
```

Rabin-Karp

Comparison based with modular hashing. $h(k) = k \mod m$ for some m. the value k is an integer hash code generated from the key (generally used with positive ints)

```java
int h(int k, int M){
    return k % M;
}
```

To prevent an overflow for large search substrings take intermediate results e.g

$(a + b) \mod Q = ((a \mod Q) + (b \mod Q)) \mod Q$

$(a * b) \mod Q = ((a \mod Q) * (b \mod Q)) \mod Q$

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

> $x_i = t_i R^{m-1} + t_{i+1} R^{m-2} + ... + t_{i+M-1} R^0 (\mod Q)$

Implementation

```java
private long hash(String key, int M) {
    long h = 0;
    for(int j = 0; j < M; j++)
        h = (h * R + key.chatAt(j)) % Q;
    return h;
}
```

```java
public class RabinKarp
{
    private long patHash; // pattern hash

    private int M; // pattern length
    private long Q; // modulus
    private int R;  // radix
    private long RM1; //R^(M-1) % Q

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = longRandomPrime();
        RM1 = 1;
        for (int i = 1; i <= M-1; i++)
            RM1 = (R * RM1) % Q;
        patHash = hash(pat, M);
    }
    private long hash(String key, int M)
        { /* as before */ }
    public int search(String txt)
        { /* see next slide */ }
}
```

Possible approaches

Monte Carlo version - Return match if the hash matches (more risk if poor hash)

Las Vegas version - Check for substring match if hash match (less risk)

```java
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
        for (int i = M; i < N; i++)
        {
            txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
            txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
        }
    return N;
}
```

Trade offs

MonteCarlo

- Always runs in linear time
- Extremely likely to return the right answer (not certain)

Las Vegas

- Always returns the right answer
- Extremely likely to run in linear time (worst case is *M N*)

Rabin-Karp analysis

Advantages

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup
- Poor worst-case guarantee.