Algorithms

September-11-12 3:03 PM

Terminology

Problem
Given a problem instance, carry out a particular computational task.

Problem Instance
Input for the specified problem

Problem Solution
Output (correct answer) for the specified problem instance.

Size of a problem instance
Size(I) is a positive integer which is a measure of the size of the instance I.

Example: Sorting
Problem instance I: 5, 1, 4, 3, 7
Output: 1, 3, 4, 5, 7
Size(I) = 5

Example: Matrix Multiplication
Matrices

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$

$C = AB$
$I = (A, B)$
$size(I) = n$

Algorithm
An algorithm is a step-by-step process for carrying out a series of computations, given an arbitrary problem instance I.

Algorithm solving a problem
An Algorithm A solves a problem Π if for every instance I of Π, A finds a valid solution for the instance I in finite time.

Program
A program is an implementation of an algorithm using a specified computer language

In this course, our emphasis is on algorithms (as opposed to programs or programming)

Algorithms and Programs
For a problem Π, we can have several algorithms.
For an algorithm A solving Π, we can have several programs (implementations)

Algorithms in practice: Given a problem Π
- Design an algorithm A that solves Π → Algorithm Design
- Assess correctness and efficiency of A → Algorithm Analysis
- If acceptable (correct and efficient), implement A.

RAM (Random Access Machine)
- Machine has only CPU and RAM
- Any memory access is constant time
- load/store/compare/add/multiply data stored in cells in constant time
  - Arrays work as expected
  - Linked lists are possible
- Infinite amount of memory
- Program is stored in memory

Example: Pseudocode of Matrix Multiplication
```
for i from 1 to n do
    for j from 1 to n do
        C[i,j] := A[i, 1] * B[1, j]
        for k from 2 to n do
            C[i, j] = C[i, j] + A[i, k] * B[k, j]
        od
    od
od
```
How many primitive operations?
n^3 multiplications
$n^2(n-1)$ additions
Total $2n^3 - n^2$ arithmetic operations
Using order notation, running time is $O(n^3)$

A better algorithm: Strasen69 $< 42n^{2.8074}$

Order Notation

September-13-12 2:39 PM

Order notation

Know all of the order notation symbols
The functions are asymptotically non-negative

O-notation

$f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$
$f(n)$ grows no faster than $g(n)$

Ω-notation

$f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$
$g(n)$ grows no faster than $f(n)$

Θ-notation

$f(n) \in \Theta(n)$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

o-notation

$f(n) \in o(g(n))$ if for all constants $c > 0$ there exists a constant n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$
$f(n)$ grows strictly more slowly than $g(n)$

ω-notation

$f(n) \in \omega(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$
$f(n)$ grows strictly more rapidly than $g(n)$

Analysis

Create formula expressing program. If can simplify exactly then get Θ running time. Using inequalities gives O or Ω and need both to get Θ

Work inside out of loops

Properties

Assume $f(a) \ge 0, g(n) \ge 0\ for\ all\ n \ge 0$

1. $f(0) \in O(af(n))$ for any constant $a > 0$
   $c = \frac{1}{a}, \quad n_0 = 1$
2. If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$
   - $f(n) \le c_1 g(n)\ \forall\ n \ge n_1$
   - $g(n) \le c_2 h(n)\ \forall\ n \ge n_2$
   - $f \le c_1 g(n) \le c_1 c_2 h(n)\ \forall\ n \ge \max(n_1, n2)$
   - $c = c_1 c_2, \quad n_0 = \max(n_1, n2)$
3. $\max(f(n), g(n)) \in O(f(n) + g(n))$
   $c = 1, \quad n_0 = 1$
   Exercise: Show $O(f(n) + g(n)) \in O(\max(f(n), g(n)))$
4. $\sum_{i=1}^{n} a_i x^i \in O(x^n), \quad a_n > 0$
   Exercise
5. $n^x \in O(a^n), \quad x > 0, a > 1$
6. $(\log n)^n \in O(n^y)$

Big Ω

$f(n)$ grows no slower then $g(n)$

Example

$n^3(\log n) \in \Omega(n^3)$ since $\log n \ge 1\ \forall n \ge 2$
$c = 2, n_0 = 2$

Big θ

$f(n)$ grows at the same rate as $g(n)$

Directly from definition:
$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $g(n) \in O(f(n))$
N.B. to show both in proofs

Little o

$f(n)$ grows slower then $g(n)$

Example

$n \in o(n^2)$

Example

$2008n^2 + 1388n \in O(n^3)$

Let $c > 0$ be given
$2008n^2 + 1388n \le 5000n^2\ \forall n \ge 1$
$= \left(\frac{5000}{n}\right) n^3 \le cn^3\ \forall\ n \ge n_0 = \frac{5000}{c}$

Abstract Data Type

September-18-12 2:40 PM

Abstract Data Type (ADT)

A description of information and a collection of operations on that information

The information is accessed only through the operations

We can have various realizations of an ADT, which specify:

- How the information is store (data structure)
- How the operations are performed (algorithms)

Notation for Trees

Height

The height of a node is the number of edges in the longest single path down to leaf

Depth

Number if edges in single path up to root

⇒ root has maximal height = height of tree

Dynamic Arrays

Linked lists support O(1) insertions, deletions but element access costs O(n)
Arrays support O(1) element access, but insertion/deletion cost O(n)

Dynamic arrays offer a compromise:
O(1) element access, and O(1) insertion/deletion at the end.
Two realization of dynamic arrays:

- Allocate one HUGE array, and only use the first part of it
- Allocate a small array initially and double its size as needed.
    (Amortized analysis is required to justify the O(1) cost for insertion/deletion at the end— CS341/466)

Stack ADT

Stack: an ADT consisting of a collection of items with operations:

- push: inserting an item
- pop: removing the most recently inserted item
- Items are removed in LIFO order. We can have extra operations: size, isEmpty, and top.

Applications: Addresses of recently visited sites in a Web browser, procedure calls.

Realizations of Stack ADT

- Using arrays
- Using linked lists

Queue ADT

Queue: an ADT consisting of a collection of items with operations:
enqueue: inserting an item
dequeue: removing the least recently inserted item
Items are removed in FIFO order.
Items enter the queue at the rear and are removed from the front.
We can have extra operations: size, isEmpty, and front.

Realizations of Queue ADT

- Using (circular) arrays
- using linked lists.

Priority Queue ADT

Priority Queue: An ADT consisting of a collection of items (each having a priority) with operations:

- insert: inserting an item tagged with a priority
- deleteMax: removing the item of highest priority
    - Also called extractMAx

Applications: typical "todo" list, simulation systems

The above definition is for a maximum-oriented priority queue. A minimum-oriented priority queue is defined in the natural way, by replacing the operation deleteMax by deleteMin.

Heap

Lemma

Height of a heap with n nodes is $\theta(\log n)$

Proof:
Suppose that the height of the tree is h.
$n \geq 2^0 + 2^1 + \cdots + 2^{h-1} + 1 = 2^h$
$\Rightarrow h \leq \log n$ so $h \in O(\log n)$

$n \leq 2^0 + 2^1 + \cdots + 2^{h-1} + 2^h = 2^{h+1} - 1$
$\Rightarrow h \geq \log(n+1) - 1$ *so* $h \in \Omega(\log n)$

Heap

September-25-12 2:38 PM

Node is larger than its children
Binary tree is nearly complete, last row is filled on the left.
Can be stored in an array row by row.

HeapInsert

Insert in last position and 'bubble up'. Swap new node with parent if it is larger. Keep doing this until parent is larger or the node becomes the root (largest). $O(\log n)$

HeapDeleteMax

- Replace root node with right most node on level h
- 'bubble down'
  - If node is smaller than any children, swap with the larger child.
  - Repeat until larger than the children or is a leaf
- $O(\log n)$

Heapify

How to initialize a heap from arbitrary array

Top-Down creation of heap

- Put items in nearly complete binary tree
- For i = 0 to n-1, bubble up at position i
- Runtime:
- Upper bound
  - Cost of bubble up is proportional to depth of node
  - Depth of each node is $O(\log n)$
  - Number of nodes to process is n
  - $O(n\, log\, n)$ runtime
- Lower Bound (On Worse case)
  - Worst case when initial ordering is in increasing order
  - need to bubble-up to root each time
  - height of tree is $h := \lfloor \lg n \rfloor$
  - Lemma: At least $\frac{n}{2} + 1$ nodes have depth $\geq h - 1$
  - In worse case # swaps for bubble-up at level h-1 or h is h-1
  - overall # swaps is $\geq \frac{n}{2}(h-1) \in \Omega(n \log n)$

- So in the worse case, has running time $\Theta(n \log n)$
- We don't qualify "worst case". Just "running time" is assumed to mean "worst case running time"

Bottom-Up creation of a heap

- Starting from end of the array, bubble-down each node in turn.
- Consider node, by level

level	# nodes	height of nodes
0	2^0	h
1	2^1	$\leq h-1$
2	2^2	$\leq h-2$
...		
h	$\leq 2^h$	0

Lemma: $n \leq 2^{h+1} - 1$
Lemma: For $i \in \{0, 1, 2, \ldots, h\}$ number of nodes with height i is $\leq 2^{h-i}$
Total number of swaps is

$$\sum_{i=0}^{h} i 2^{h-i} \leq n \sum_{i=0}^{h} \frac{i}{2^i} < n \sum_{i=0}^{h} \frac{i}{2^i} = 2n$$

- Lower bound is $O(n)$ because the loop iterates n times

  So the running time of bottom-up heap creation is $\Theta(n)$

Heapsort

```
heapify(A, n)
for i=0 to n-1 do
    A[n-i-1]=heapDeleteMax(A, n-i)
```

$O(n \log n)$

Selection Problem

September-27-12 2:33 PM

Selection

Find element at position k
Suppose $A[0 \dots n-1]$ contains distinct keys
Position of $A[i]$ =# of keys in $A[0, \dots i-1]$

Quick Select Algorithm

Find pivot, center around the pivot, and recurse on one of the halves (or return the pivot)

Best case: $O(n)$
Worse case $O(n^2)$

Find k^th largest in array $A[0 \dots n-1]$ of numbers

1) Scan array k times, delete max each time
   $O(kn)$
2) Sort numbers, then return $A[n-k]$
   $O(n \log n)$
3) Heap: Build a min-heap of size k
   a. If the next element in A is larger than min of heap
   b. Remove min from heap
   c. Insert element of A

   Return minimum element of heap, this is the k^{th} largest
   $O(n \log k)$
4) Heapify, call delete max k times
   $O(n + k \log n)$

Average Case Analysis

Example

$1 \le k \le 6$

```
foo1(k)
    for i = 1 to k do
        print "Hello world!"
```

What is the average case number of calls to print? Assuming uniform distribution of k

$$\frac{1}{6}\sum_{i=1}^{6} i = \frac{7}{2}$$

Average Case Running Time of Quick Select

Assumption 1: Keys are distinct
Observation: Behaviour of algorithm depends on relative ordering, not actual values
Therefore, can assume that inputs are integers 1 … n
Need to consider all $n!$ possible orderings.

Assumption 2: Uniform distribution

Let $T(n)$ be the cost of a quick select
There are n possible choices of pivots

$$T(n,k) = cn + \frac{1}{n}\left(\sum_{i=0}^{k-1} T\big(n-i-1, k-(i+1)\big) + \sum_{i=k+1}^{n-1} T(i,k)\right)$$

At least half of all the $n!$ problem instances will have $piv \in S = \left[\frac{n}{4}, \frac{3n}{4}\right]$ and hence will have recursive call with size at most $\frac{3n}{4}$

$$T(n) \le cn + \frac{1}{2}\left(T\left(\left\lfloor\frac{3n}{4}\right\rfloor\right) + T(n)\right)$$

$$T(n) \le 2cn + T\left(\left\lfloor\frac{3n}{4}\right\rfloor\right) \le 2cn + 2c\left(\frac{3n}{4}\right) + 2c\left(\frac{9n}{16}\right) + \cdots + d \le d + 2cn\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i \in O(n)$$

Randomized Algorithms

October-02-12 2:33 PM

Average Running Time
Average over all inputs

Expected Running Time
Average over all possible random choices

Worst-Case Expected Runtime
$T^{(\exp)}(n) = \max_{size(I)=n} T^{(\exp)}(I)$

Example

```
foo2 ()
    k = value of a fair die toss
    for i = 1 to k
        print("Hello World!")
```

Here the expected runtime is

$$\sum_{k=1}^{6} \frac{k}{6} = \frac{7}{2}$$

```
pre: A(0..n-1) is a permutation 1,…,1,n
    A has (n-1) 1's and 1 n

foo3(A)
    k = 0
    for i = 1 to A[k] do
        print("Hello World!")
```

prints in best case: 1
prints in worst case: n

Average runtime

$$\frac{1}{n}(1 \times n + (n-1) \times 1) = \frac{2n-1}{n} = 2 - \frac{1}{n} < 2$$

```
pre: A[0...n-1] is permutation of 1, 1, …, 1, n
foo4(A)
    k= random integer in range 0...n-1 // uniform
    for i = 1 to A[k] do
        print("Hello World!")
```

Let I_1 be input $A = [1, 1, 4, 1]$. Expected running time is

$$T^{(\exp)}(I_1) = \frac{1}{4}(3+4) = \frac{7}{4}$$

Let I_n be input $A = [n, 1, \dots, 1]$

$$T^{(\exp)}(I_n) = \sum_{k=0}^{n-1} T(I_{n,k}) \times \frac{1}{n} = \frac{1}{n}\big((n-1) \times 1 + n \times 1\big) = \frac{2n-1}{2} < 2$$

Does not matter what array is past in, worst case expected runtime is

$$T^{(\exp)}(n) = n \times \frac{1}{n} + 1 \times \frac{n+1}{n} = \frac{2n-1}{n} < 2$$

Choose-Pivot 3
Non-random pivot selection with good worst-case

1) Group elements in $k = \left\lceil \frac{n}{5} \right\rceil$ groups of at most 5 elements
2) Get median g_i of each group $1 \le i \le k$
3) Recursively compute the median g of $g_1, \dots, g_k$
   If $n = 5k$ for odd k
   $\Rightarrow \frac{k-1}{2}$ of g_i are $> g$
   $\Rightarrow$ at least $k-1$ elements in A are $> g$
   $\Rightarrow$ at leas $k-1$ elements in A are $<$ g
4) $i = partition(A, p)$

Runtime

$$T(n) \le T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{4n}{5} \right\rceil\right) + O(n)$$

$$T(n) \in O(n)$$

Sorting Algorithms

October-04-12 2:35 PM

Theorem

The worst case runtime of every comparison based sorting algorithm is $\Omega(n \log n)$

Counting Sort

Count how many of each key value, then put elements in correct position.

```
0 ≤ A[i] < k,  0 ≤ i < n
(A really consists of key-value pairs)
C ← array of size k, = 0

for i = 0 to n - 1
    increment C[A[i]]
for i = 1 to k - 1
    C[i] ← C[i] + C[i - 1]

B ← copy(A)
for i = n - 1 to 0
    decrement C[B[i]]
    A[C[B[i]]] = B[i]
```

Runtime

cost is $\Theta(n + k)$ if $k \in O(n)$ then the runtime is $O(n)$

Last loop is backwards to ensure stability

Radix Sort

Idea

Modify counting sort to handle large keys

- Consider keys as d-digit base-k numbers

Fact

For $k > 1$, every integer $x, 0 \leq x \leq k^d - 1$ can be written as $x = x_0 + x_1 k + x_2 k^2 + \cdots + x_{d-1} k^{d-1}$
for unique $x_i, 0 \leq x_i \leq k - 1$

Write numbers as tuples, of x_i, left pad with 0's as needed. Sort from least significant place to most significant using stable sort (counting sort) on only that digit.

Comparison Model

Only data accesses are

- Comparing two elements
- Move elements around

Goal

Lower bound for # comparisons required by any comparison based sorting algorithm

Idea

Model execution of algorithm on all inputs using a decision tree

Structure

Each node has zero or two children

- # leaves = # internal nodes + 1

Example sorting decision tree

$n = 3$
$[x_1, x_2, x_3] \in \{[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]\}$

```
                      _ [x3,x2,x1]
                     / <        __[x2,x1,x3]
            _ x3:x2            / <
          / <          \_ x1:x3
x2:x1            <            \__[x2,x3,x1]
      \ x3:x1 —[x3,x1,x2]
             \_ x3:x2 — [x1,x3,x2]
                      \_ [x1,x2,x3]
```

Notes

- Internal nodes : comparison
- Result of comparison : edge
- Leaf nodes : result(sorted)
- Worst case : height of tree
- Average case : Average depth of leaves

Proof of Theorem

At least one leaf node for each possible input.
⇒ At least $n!$ leaf noes
⇒ at least $n!$ nodes in tree

⇒ height $\geq \lg n! \geq \lg \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \lg \frac{n}{2} \in \Omega(n \log n)$

Trees

October-16-12 2:31 PM

Multi-Way Search Tree

An ordered tree such that

- each node has at least 2 children
- each node with c children $t_1, t_2, \ldots, t_c$ stores $(c-1)$ keys s.t. $k_1 < k_2 < \cdots < k_{c-1}$
- for all (k, v) stored in the subtree rooted at t_i we have $k_{i-1} < k < k_i$

Search

- Start at root
- if $k = k_i$ for some i then done
- else recursively search in subtree rotted at t_i s.t. $k_{i-1} < k < k_i$

2-3 Trees

- multi-way-search trees
- each non-leaf node has 2 or 3 children
  - allowed 1 or 2 key-value pairs per node
- all leaves at same level
- insert
  - always fill leaf nodes
  - if overflow, promote middle element
- height of tree grows iff root

Summary

- Search is easy
- Insert may involve splitting/promotion
- deletion may involve transfer or fusion
  - fusion may repeat up to root
- h increases only if root splits
- h decreases only if root's siblings fuse and root becomes empty

B-tree of min size d

- Same idea as a 2-3 tree
  - each node has $\leq 2d$ keys
  - each non-root node has $\geq d$ keys
- Implementation:
- Chose d so that a node with 2d KVPs fills a disk sector
- goal: minimize disk access
- keep root in RAM
- eg: $d = 256$, $n = 16$ million
  - 16 million $\cong 256^3 = (2d+1)^h - 1$
  - 3 disk accesses

Example Construction of a 2-3 Tree

- insert 1, 2
  - [1, 2]
- insert 3
  - [1, 2, 3] ⇒ [1] ← [2] → [3]
- insert 4
  - [1] ← [2] → [3, 4]
- insert 5
  - [1] ← [2] → [3,4,5] ⇒ [1] ← [2, 4]→ [5]
[3]
- insert 6
  - [1] ← [2, 4] → [5, 6]
[3]
- insert 7
  - [1] ← [2, 4] → [5, 6, 7]
[3]
  - [1] ← [2, 4, 6] → [7]
[3] [7]
  - [1] ← [2] ← [4] → [6] → [7]
[3] [5]

Example B-tree of min size 3

[1, 2, 3, 4, 5, 6] ← 7 → [8, 9, 10]
insert 0
[0, 1, 2, 3, 4, 5, 6] ← 7 → [8, 9, 10]
[0, 1, 2] ← [3, 7] → [8, 9, 10]
[4, 5, 6]
delete 10
[0, 1, 2] ← [3, 7] → [8, 9]
[4, 5, 6]
[0, 1, 2] ← [3] → [4, 5, 6, 7, 8, 9]

Hash Tables

October-25-12 2:31 PM

Cuckoo Hashing

- Open addressing
- two independent hash functions, h_1, h_2
- always insert k into $T[h_1(k)]$
    - may displace another item
        - If so, insert in alternate location
    - If get into a cycle then rehash

Extendible Hashing Delete

- Reverse of insert
- keep directory/root as small as possible
- merge with "buddy" if possible
    - Leaf node with same local depth
    - agree on first $k_B - 1$ bits.

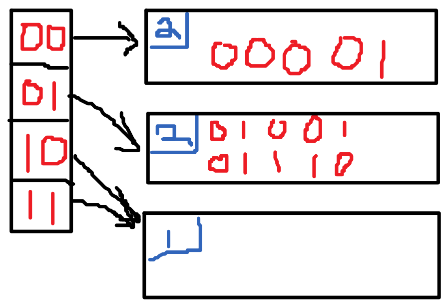
Example of Cuckoo Hashing

Insert k into T

- $h_1(k) = 3$, but $T[3]$ is occupied
- kick out $\bar{k}$, insert k into $T[3]$
- insert $\bar{k}$ into $h_1(\bar{k}) = 1$ or $h_2(\bar{k}) = 3$

0	1	2	3	4
			$\bar{k}$	
	$\bar{k}$		k	

Example of Extendible Hashing



k_b is the local depth
$k_B \leq d$ of each block

- keys in B had k_B leading bits in common

exactly e^{d-k_B} pointers to block B

Example (L=5, S=2)

insert 01001 into initially empty ext. hash table
[] → [0| 01001]

Insert 00001
[] → [0 | 01001]
[00001]

insert 01110
[0] → [1| 01001]
[00001]
[1] → [1|]

[00] → [2| 00001]
[01] → [2| 01001]
[| 01110]
[10] → [1|]
[11] ^

Insert 11100 and 10110
[00] → [2| 00001]
[01] → [2| 01001]
[| 01110]
[10] → [1| 10110]
[11] ^ [11100]

Insert 01010, 11101, 01011, 00000 (exercise)
[0000] → [2| 00001]
[0001] ^ [00000]
[0010] ^
[0011] ^

[0100] → [4| 01001]
[0101] → [4| 01010]
[01011]
[0110] → [3| 01110]
[0111] ^
[1000] → [2| 10110]
[1001] ^
[1010] ^
[1011] ^
[1100] → [2| 11100]
[1101] ^ [11101]
[1110] ^
[1111] ^

delete 01011
[000] → [2| 00001]
[001] ^ [00000]
[010] → [3| 01001]
[01010]
[011] → [3| 01110]
[100] → [2| 10110]
[101] ^
[110] → [2| 11100]
[111] ^ [11101]

delete 01010
[00] → [2| 00001]
[00000]
[01] → [2| 01001]
[01110]
[10] → [2| 10110]
[11] → [2| 11100]
[11101]

Multidimensional Data

November-08-12 3:05 PM

kd-tree

Search Running Time

Runtime is bounded by number of calls to

Tries and String Matching

November-13-12 2:32 PM

Binary Tries (or Radix Trees)

- Stores a collection of binary strings
  - eg. {00, 110, 111, 01010, 01011}
- Assumption: Strings are prefix-free
- Runtime analysis takes length of string into account.

Prefix-Free

No string is a prefix of another

Compressed Trie (Patricia Tree)

KMP

Guess index: $i - j$
Check index i
Both monotonically nondecreasing
Main loop invariant: $P[0 \dots j-i] = T[i-j \dots i-1]$

T is string to be searched
P is desired substring.

Algorithm:

While $i < n$ do

- Case 1: $T[i] = P[j]$ and $j < m - 1$
  `i++; j++`
- Case 2: $T[i] = P[j]$ and $j = n - 1$
  `Return i-j;`
- Case 3: $T[i] \neq P[j]$ and $j = 0$
  `i++`
- Case 4: $T[i] \neq P[j]$ and $j > 0$
  Main idea of KMP: shift pattern certain amount right
  - Keep i the same
  - Decrement j by correct amount.

  Choose $j' < j$ maximal such that
  $P[0 \dots j'-1] = T[i-j' \dots i-1]$

  $j = F[j-1]$

KMP Failure Function

The failure function $F(j)$ for pattern $P[0, \dots, m-1]$

a) $F(0) = 0$
b) For $j > 0$, $F(j)$ is length of largest prefix of $P[0 \dots j]$ that is also a suffix of $P[1 \dots j]$

Usage

$j > 0$ and $T[i] \neq P[j] \Rightarrow j = F[j-1]$

Example of KMP Matching

T=bacbababaabbcbab
P=ababaca

```
b a c b a b a b a a b c b a b
a
  a b
    a
      a
        a b a b a c
           (a (b (a b
               (a b
                  a b a
                      a
                        a
                          a b
```

Example Construction of KMP Failure Function

Let
P = a b a c a b a

```
  j | 0 1 2 3 4 5 6
F(j)| 0 0 1 0 1 2 3
```

Compressed Multiway Trie

Store string to be searched in a suffix trie
For each suffix, store node in a compressed trie. The node contains the initial and final indices of the suffix. Internal nodes store initial and final indices of the substring represented by that node.

Compression

November-22-12 2:53 PM

Non-Prefix Free Example

Decoding Dictionary

$\Sigma^*_{\{0,1\}} \to \Sigma^*_{\{a,b,c,\ldots,A,B,C,\ldots\}}$

E	1010
S	11
O	1011
Y	01
N	0110

Coded Message: 01101011
Can be decoded as NO or as YES

Length of Trie Encoding

Let S be array of n characters and frequencies, $n \geq 2$
Let T be any encoding Trie
Length of encoded text is

$$\sum_{c \in S} (\text{\# occurences of } c) \times (\text{length of code for } c) = \sum_{c \in S} f(c) \times (\text{depth of leaf containing } c) = WPL(T)$$

= the weighted path length of T

Theorem (Length of Huffman Tree)

For a Huffman tree H, $WPL(H) \leq WPL(T)$

LZW Dealing with Full Dictionary

1. Stop adding
    a. bad if data changes structure
    b. fast
2. Clear and start with fresh dictionary
    a. temporary poor compression
3. Discard least frequently used
    a. maybe complicated or expensive
4. Increase k (size of output)