

More on Strings

Hsuan-Tien Lin

Dept. of CSIE, NTU

March 21–22, 2011

What We Have Done

- Various Implementations of 2D Array
- Row Major vs. Col Major Access
- Adding Sparse Polynomials
- Sparse Matrix by Ordered Triples
- Fast Transpose of Sparse Matrix
- Fast Multiplication of Sparse Matrix
- Reading Assignment:
Structures and Unions, Abstract Polynomials et al.,
Multi-Dim Arrays, String (ADT/C)

Homework 2

- Asymptotic Complexity:
“proof” from **only the known definitions/theorems**
- Structure:
simple, do after reading assignment
- Sparse Matrix:
explore another implementation that uses **(dense-bit-array + elements)** instead of (paired-indices + elements)
- Special Matrix:
much like method 1 of 2D Array, but on a **band matrix**
- Sparse Matrix Processing:
coding for **huge data set**

Some Hints on Sparse Matrix Processing

row = 18000, col = 480000, element = 100000000

- on your computer:
 - read in the file by your special program
—better than Notepad/VIM/etc.
 - no need to open the whole file before you can start
—check README, check first few lines, etc.
- on CSIE R217 workstation:
 - access the file /tmp2/DSA2011/data_2_5.txt
—better than a file in your home dir
 - no quota vs. quota
 - local copy vs. network file system
 - can use `head` or `tail`
- design before implement:
 - large data set: hard to “trial-and-error”
 - express your design on paper is important

More Hints on Sparse Matrix Processing

discuss with TAs/classmates/instructor and be creative

- do you need a 4-byte integer per element $\in \{1, 2, 3, 4, 5\}$?
- storing both R and R^T (some redundancy) so both $users(m, n)$ and $movies(u, v)$ would be fast?
- don't be afraid to try; don't settle for naive solutions

Strings: Pattern Matching (Subsec. 2.7.3)

find the position that *pat* (first) shows up in *string*

```
for  $i \leftarrow 0$  to  $\text{len}(\text{string}) - 1$  do  
  if pat matches  $\text{strings}[i, i + \text{len}(\text{pat}) - 1]$   
    matching found  
  end if  
end for  
matching not found
```



- the IF takes $O(m)$ for $m = \text{len}(\text{pat})$
—can use heuristic on comparing *begin* and *end* first, but still $O(m)$ in the worst case
- so total $O(n * m)$ for $n = \text{len}(\text{string})$

Slow (Naive) Pattern Matching

```
i, j ← 0
while i < len(string) and j < len(pat) do
  if pat[j] == string[i]
    i ← i + 1, j ← j + 1 (continue matching)
  else
    i ← i - j + 1
    j ← 0
    (fail and totally go back)
  end if
end while
check matching status
```

see demo

“Jump” Pattern Matching

```
i, j ← 0
while i < len(string) and j < len(pat) do
  if pat[j] == string[i]
    i ← i + 1, j ← j + 1 (continue matching)
  else
    i ← i - min(jump, j) + 1
    j ← 0
    (fail and go to next possible starting point)
  end if
end while
check matching status
```

see demo

Fast (Knuth-Morris-Pratt) Pattern Matching

```
i, j ← 0
while i < len(string) and j < len(pat) do
  if pat[j] == string[i]
    i ← i + 1, j ← j + 1 (continue matching)
  else
    i ← i
    decrease j such that pat[0, j - 1] matches string[i - j, i - 1]
    (fail but continue partially)
  end if
end while
check matching status
```

see demo



- Ph.D., Caltech Math
- Professor Emeritus, Stanford
- 1974 ACM A. M. Turing Award
(who is Turing and what is Turing Award?)
- 1995 IEEE John von Neumann Medal
(who is von Neumann?)

For his major contributions to the analysis of algorithms and the design of programming languages, and in particular for his contributions to “The Art of Computer Programming” through his well-known books in a continuous series by this title

KMP Pattern Matching

	a	b	c	a	b	c	a	b	c	a	c	a	b	c	a	b	c	a	a	d	a	b	c	a	b	c
a	■			■			■			■		■			■			■	■	■			■			
b		■			■			■			■			■			■		■	■	■		■			
c			■			■			■			■			■			■			■			■		
a				■			■			■			■			■			■			■			■	
b					■			■			■			■			■		■			■			■	
c						■			■			■			■			■			■			■		
a							■			■			■			■			■			■			■	
c								■			■			■			■			■			■			■
a									■			■			■			■			■			■		
b										■			■			■			■			■			■	
d											■				■			■			■			■		

- number of increase $i = O(\text{len}(\text{string})) = \text{number of increase } j$
- number of decrease $j = O(\text{number of increase } j)$ because $j \geq 0$
- total: $O(\text{len}(\text{string}))$ IF the decrease step is $O(1)$

How To Make the Decrease Step $O(1)$

- main tool: **pre-compute** how you decrease j
- how to decrease j to $t + 1$ such that $pat[0, t]$ matches $string[i - t - 1, i - 1]$
 - originally, $pat[0, j - 1] == string[i - j, i - 1]$
 - want, $pat[0, t] == string[i - t - 1, i - 1]$
 - note, $pat[j - t - 1, j - 1] == string[i - t - 1, i - 1]$
 - so, $pat[0, t] == pat[j - t - 1, j - 1]$

整串綠色
 $\oplus = \text{丙}$
 $\ominus = \text{丙}$

An $O(1)$ decrease step:

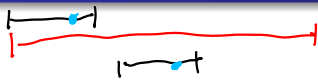
$$j \leftarrow f[j - 1] + 1$$

$$\text{where } f[j - 1] = \underset{t}{\operatorname{argmax}} \{ pat[0, t] == pat[j - t - 1, j - 1] \}$$

$$\text{or } f[j - 1] = -1 (\text{otherwise})$$

- a trivial algorithm of $O(len(pat)^2)$ for pre-computing f :
double for loops on j and t

Even Faster Algorithm for Pre-Computing f



$$f[j-1] = \underset{t}{\operatorname{argmax}} \{ \text{pat}[0, t] == \text{pat}[j-t-1, j-1] \}$$

or $f[j-1] = -1$ (otherwise)

- see textbook for something not so easy to understand
- a vernacular explanation: to get $f[j] = \text{res}$,
 - candidate 1: $\text{res} = f[j-1] + 1$, check if $\text{pat}[\text{res}] == \text{pat}[t]$
 - candidate 2: $\text{res} = f[f[j-1]] + 1$, check if $\text{pat}[\text{res}] == \text{pat}[t]$
 - candidate 3: $\text{res} = f[f[f[j-1]]] + 1$, check if $\text{pat}[\text{res}] == \text{pat}[t]$
 - ...
 - otherwise: $\text{res} = -1$

read the textbook for why it's faster!