Computer Science 70

Discrete Mathematics and Probability Theory

Hashing

Lecture 23

2003-10-22

Dan Garcia

(www.cs.berkeley.edu/~ddgarcia)

`inst.eecs.berkeley.edu/~cs70/`

1 Handout: notes

Big Idea: memoization

- General principle: store rather than recompute.
- Context is a tree-recursive algorithm with lots of repeated computation, e.g. Fibonacci:

```
int Fib (int n) {
  if (n==0 || n==1) {
    return n;
  } else if (we've computed n's value already) {
    return that value;
  } else {
    int value = Fib(n-1) + Fib(n-2);
    store (n, value);
    return value;
  }
}
```

- Pairs (`n`, value of `Fib(n)`) are stored in the table.

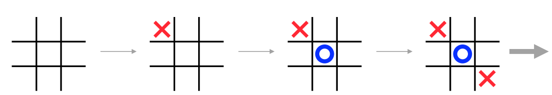
Hash Function

- If what we want to memoize isn't a simple number, how do we convert it to a number to easily store it into a table?
- We need something that can help us map this data into an integer, to serve as an index into an array (used to store the table).
- This mapping function is called a hash function

`http://en.wikipedia.org/wiki/Hash_function`

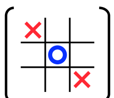
Writing hash functions - TTT (1)

- Let's consider Tic-Tac-Toe:
  - One player chooses X, the other chooses O
  - They take turns placing their piece on the board
  - Assume X goes first
  - Once a piece is placed, it isn't moved
  - The player who first gets 3-in-a-row wins
  - If the board gets filled up and nobody wins, it's a tie

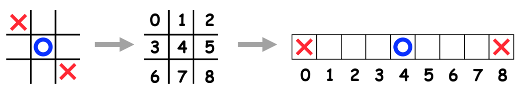


Writing hash functions - TTT (2)

- **Writing a Tic-Tac-Toe hash function:**

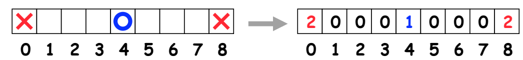
h  = 13,205

- **One idea is to ignore the 2D nature of the game and make it a 1D array of slots**



Writing hash functions - TTT (3)

- **Think of each of the 9 slots as 1 of 3 values**
  - Blank, O and X
  - Let's assign values 0, 1 and 2 to these



Writing hash functions - TTT (4)

- **Analysis of ternary polynomial hashcode:**
  - What's the smallest #? 0
  - What's the biggest #? 3^9-1
  - Is this as optimal (I.e., tightly-packed) as possible? No!
  - Any suggestions for making this more optimal?

Writing hash functions - TTT (5)

- **Optimizing the Tic-Tac-Toe hash function**
  - This involves understanding the rules of placement
    - » The players take turns & X goes first!
  - Let's consider some small 1D boards (S = # of slots)
    - » S=1: 2 boards (- | X) **We'll use "|" to separate groups**
    - » S=2: 5 boards (-- | -X, X- | XO, OX)
    - » S=3: 13 boards (--- | --X, -X-, X-- | -OX, -XO, O-X, OX-, X-O, XO- | OXX, XOX, XXO) = (1 + 3 + 6 + 3)
    - » S=4: __ boards (__ + __ + __ + __ + __)
    - » …pattern?

Remember your Combinatorics!

- **Let's figure out `numBoards(s)`, s = # slots**
- **For n=5, we had:**

  # ways to rearrange 0 Xs, 0 Os in 4 slots +
  # ways to rearrange 1 Xs, 0 Os in 4 slots +
  # ways to rearrange 1 Xs, 1 Os in 4 slots +
  # ways to rearrange 2 Xs, 1 Os in 4 slots +
  # ways to rearrange 2 Xs, 2 Os in 4 slots

- **Generalizing from this example (p=# pieces):**

$$numBoards(s) = \sum_{p=0}^{\lceil s/2 \rceil} rearrange(p,p,s) + \sum_{p=1}^{\lfloor s/2 \rfloor} rearrange(p,p-1,s)$$

- **But what is rearrange(x,o,s)?**
  - # of ways to rearrange x Xs, o Os in s slots?

Recall Pascal's Triangle $\binom{5}{2}=10$

K

N	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

This table describes how to calculate combinations. I.e., "N choose K".

$$\binom{N}{K} = \frac{N!}{K!\,(N-K)!}$$

That is, the number of ways to rearrange 2 pieces in 5 slots is "5 choose 2", which is the expression at the top. 10 ways.

`rearrange(x,o,s) = r(x,o,s)`

- **How many ways to rearrange x Xs, o Os in s slots?**

X				O				X
0	1	2	3	4	5	6	7	8

- **Blur method**
  - First, blur eyes, how many ways to rearrange ALL (x+o) pieces in s slots? [stop blurring now] $\binom{s}{x+o}$
  - For EACH, how many ways to rearrange Xs in pieces? $\binom{x+o}{x}$
  - Answer is product of these

$$\frac{s!}{(s-x-o)!(x+o)!} \cdot \frac{(x+o)!}{o!\,x!} = \binom{s}{x+o}\binom{x+o}{x}$$

- **Overcount method**
  - Think of permuting all the elements; how many? $s!$
  - How many were overcounted? Xs, Os, spaces $o!\quad x!\quad (s-x-o)!$
  - Answer is quotient of these

$$\frac{s!}{o!\,x!\,(s-x-o)!}$$

Now we know our Hash Table size

- **Now we know `numBoards(s)`**
  - `numBoards(4)` ⇒ (1 + 4 + 12 + 12 + 6) = 35
  - `numBoards(9)` ⇒ (1+9+72+252+756+1260+1680+1260+630+126) = 6,046 < 19,683 = 3^9
- **Plotting `rearrange(x,o,4)`**

Note zig-zag pattern as a result of the alternating moves of each player! `numBoards` just sums 'em!

o

2			6
1		12	12
0	1	4	
s=4	0	1	2

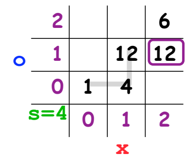
x

But what about the hash function?

- How do we write **the combinatorially optimal** `hash()`?
  - This take our board and generates a # between 0 and (`numBoards` - 1)
- Two steps (sum the following numbers)
  1. Finding out how many numbers there were in the zigzag up to our box (this is the BIAS, or OFFSET)
  2. Finding out our number REARRANGEMENT within our box
     - » Exactly same idea as the ternary polynomial hash code:
       - X counts as 2, i.e., $2 \cdot 3^i$, O counts as 1, I.e., $1 \cdot 3^i$, - = 0
     - » Here, we consider the leftmost slot & how much it's worth
       - X counts for all ways to rearrange if it were O & -
       - O counts for all ways to rearrange if it were -
       - - counts for 0
       - (Shortcut when a board has all the same piece, counts for 0)

Example TicTacToe hash function

- Let's hash XO-X = $X_3 O_2 -_1 X_0$
  - Must be a # between 0 and (`numBoards(4)` - 1) = 34
- Two steps: BIAS + REARRANGEMENT #
  - BIAS: X=2,O=1,S=4; Count buckets up to us:1+4+12=17
  - REARRANGEMENT #: [R(X,O,S)]
    - » X_3 = r(2,1,3) + r(2,0,3) = 3 + 3
    - » O_2 = r(1,1,2) = 2
    - » $-_1$ = 0
    - » X_0 = 0 (from shortcut)
    - » REARRANGEMENT # = 3 + 3 + 2 = 8
- Thus, combinatorially optimal `hash`(XO-X) = 17 + 8 = 25



Summary

- We showed how to calculate combinatorially optimal hash functions for a game
  - In real-world applications, we often find this useful
  - If it's too expensive, we usu. settle for sub-optimal
- A good hash function spreads out values evenly
- Sometimes hard to write good hash function
  - In 8 real applications, 2 had written poor hash funs
- Java has a great hash function for Strings
  - Strings are commonly used as the keys (the things you hash upon for a data structure)