

Hashing

Biostatistics 615 / 815

Lecture 10

Problem Set 3 Notes

- **Dynamic Programming**

- **Top Down**

- Recursive implementation, with additional code to store results of each evaluation (at the end) and to use previously stored results (at the beginning)

- **Bottom Up**

- Evaluate small values of the function and proceed to successively larger values.

Problem 1

- Using top-down dynamic programming, evaluate the beta-binomial distribution
 - Like other recursive functions, this one can be very costly to evaluate for non-trivial cases
- Must initialize matrix or results could be wrong
 - Use global variable for intermediate results

Problem 2

- Using bottom-up dynamic programming, evaluate:

$$C(N) = \begin{cases} N + \frac{1}{N} \sum_{k=1}^N (C(k-1) + C(N-k)) & N \geq 2 \\ 0 & N \leq 1 \end{cases}$$

- Calculation is still slow, due to nested sum... but this can be simplified

Speedy Solution ...

```
int comparisons[Nmax];  
double inner_sum = 0.0;  
  
comparisons[1] = comparisons[0] = 0;  
  
for (i = 2; i < Nmax; i++)  
{  
    inner_sum += 2 * comparisons[i - 1];  
    comparisons[i] = i + inner_sum / i;  
}
```

Last Lecture

- Merge Sort
 - Bottom-Up
 - Top-Down
- Divide and conquer sort with guaranteed $N \log N$ running time
 - Requires additional auxiliary storage

Today

- Hashing Algorithms
- Fast way to organize data prior to searching
- Trade savings in computing time for additional memory use

Almost Trivia

- Short detour... Finding primes
- How do we find all prime numbers less than some number?

Eratosthenes Sieve

- List all numbers less than N
 - Ignore 0 and 1
- Find the smallest number in the list
 - Mark this number as prime
 - Remove all its multiples from the list
- Repeat previous step until list is empty

The Sieve in C

```
void list_primes()
{
    int i, j, a[N];

    for (i = 2; i < N; i++)
        a[i] = 1;

    for (i = 2; i < N; i++)
        if (a[i])
            for (j = i * i; j < N; j += i)
                a[j] = 0;

    for (i = 2; i < N; i++)
        if (a[i]) printf("%4d is prime\n", i);
}
```

Notes on Prime Finding

- The algorithm is extremely fast
 - Takes <1 sec to find all primes <1,000,000
- Performance can be improved by tweaking the inner loop
 - Can you suggest a way?
- Illustrates useful idea:
 - Use values and indices into an array where items denote presence / absence of the value in a set.

Idea

- If all items are integers within a short range...
 - ... speed up search operations
 - ... avoid having to sort data
- How?

Even better!

- With this strategy...
 - Adding an item to the collection takes constant time
 - Searching through the collection takes constant time
 - Independent of the number of objects in the collection!

Previous Search Strategies

- Place data into an array
 - $O(N)$
- Sort array containing data
 - $O(N \log N)$
- Search for items of interest
 - $\log N$ per search

Using Items as Array Indexes

- Place data into an array
 - $O(N)$
- ~~Sort array containing data~~
- Search for items of interest
 - $O(1)$ per search

Another Example ...

- Consider a sorted array with N elements
 - Assume elements uniformly distributed between 0 and 1
- Using binary search, checking for a specific element is about $O(\log_2 N)$
- Can we do better by taking into account data is uniformly distributed?

Improving on Binary Search...

- Let $N = 100$
- What are likely locations for the values
 - 0.0001
 - 0.4281
 - 0.9941
- In fact, we can improve on binary search so that we need only about $\frac{1}{2}\log_2 N$ comparisons.

General Principle

- If the elements we are searching for provide information about their location in the array, we can reduce search times
- We will describe a way to convert an arbitrary element of interest into a likely array location
 - In fact, a series of locations!

Hashing

- Method for converting arbitrary items into array indexes
 - Items can always serve as array indexes...
- A different approach to searching
 - Not (primarily) based on comparisons

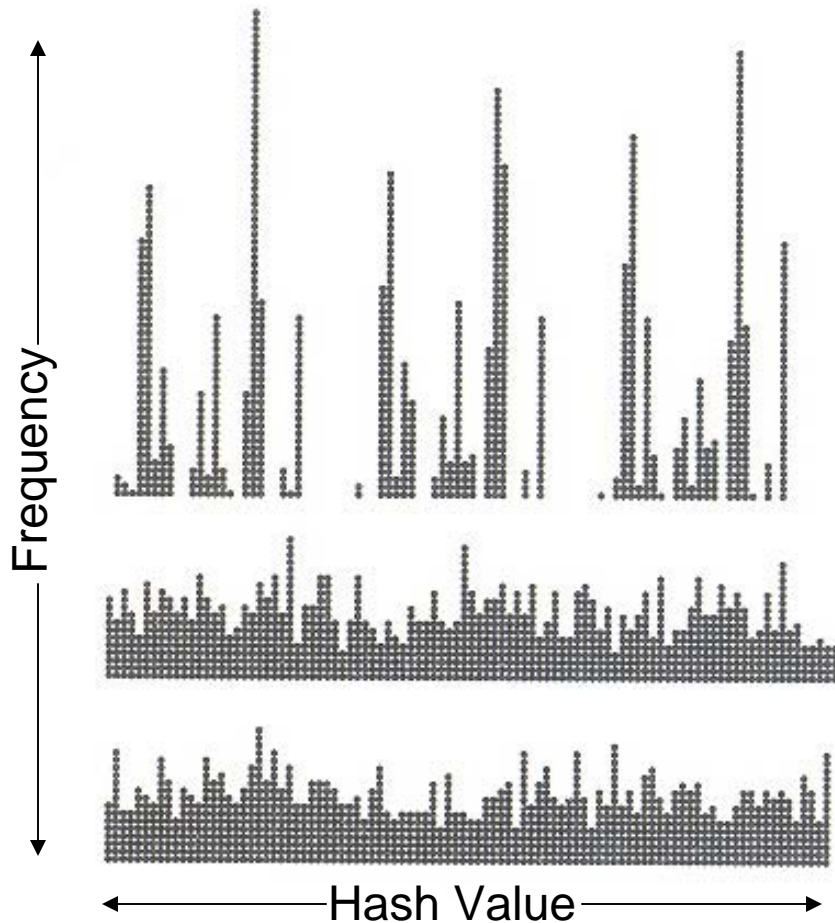
Time - Space Trade Off

- If memory were no issue...
 - Could allocate arbitrarily large array so that each possible item could be a unique index
- If computing time were no issue...
 - Could use linear search to identify matches
- Hashing balances these two extremes

Components of Hashing

- Hash Function
 - Generates table address for individual key
- Collision-Resolution Strategy
 - Deals with keys for which the Hash Function generates identical addresses

Desirable Hash Functions



Poor Hash Function:
Data is clustered

Good Hash Functions:
Data is Evenly Distributed

Hash Function #1

- Assume indexes are in range $[0, M - 1]$
- Items are floating point values between $0 .. 1$
- Multiply by M and round

Hash Function #1

- In general, if items take ...
 - Minimum value *min*
 - Maximum value *max*
- Define hash function as

$$(item - min) / (max - min) * M$$

Unfortunately ...

- If the items are not randomly distributed within their range...
- Hash function will generate a lot of collisions.
- Better strategies exist...

Hash Function #2

- For integers
- Ensure that table size M is prime
- Define hash function as

$$\textit{item modulus } M$$

Note: The modulus operators is % (in C)

Hash Function #2

- For floating point values
- First map item into 0 .. 1 range
 - As before ...
- Multiply result by large integer (say 2^k) and truncate
- Calculate modulus M (where M is prime)

Two Simple Hash Functions

```
int hash_int(int item, int M)
{ return item % M; }
```

```
int hash_double(double item, int M)
{
return (int) ((item-min)/(max-min)*
                LARGE_NUMBER) % M;
}
```

Hashing for Strings

- It is also possible to hash strings ...
- One way is to convert each string to a number
 - List all possible characters that could occur
 - Assign the value '1' to one of them, '2' to another, ...
- Although the conversion sounds cumbersome, it is built into the way C represents strings
 - Each character is also a number ...

A Hash Function for Strings

```
int hash_string(char * s, int M)
{
    int hash = 0, mult = 127, i;

    for (i = 0; s[i] != 0; i++)
        hash += (s[i] + hash*mult) % M;

    return hash;
}
```

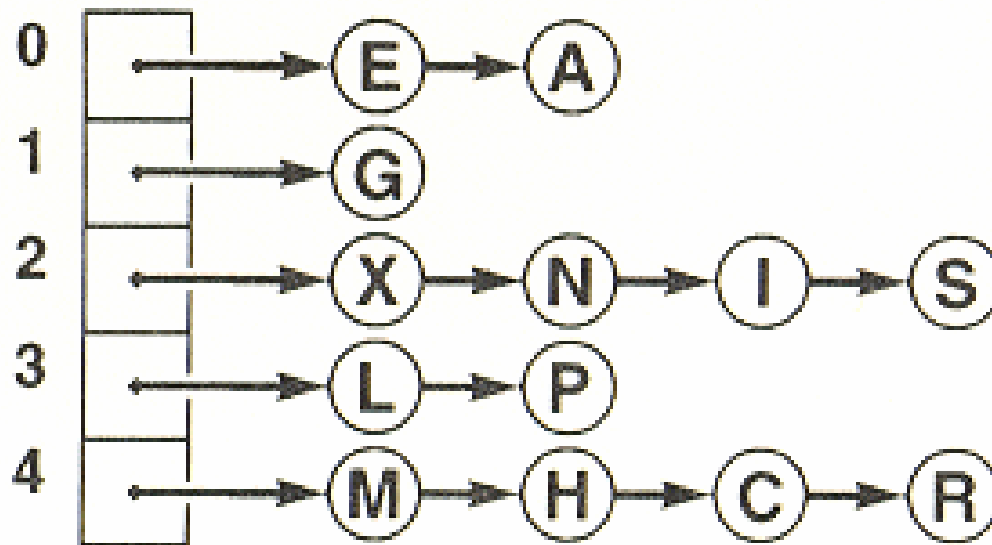
In C, characters can be treated as numbers. A string is an array of characters that terminates with the element zero.

Conflict Resolution: Separate Chaining

- What to do with items where the hash function returns the same value?
- One option is to make each entry in the hash table an array or list ...
 - Each entry corresponds to a "chain of items"

Separate Chaining Example

A	S	E	R	C	H	I	N	G	X	M	P	L	Key
0	2	0	4	4	4	2	2	1	2	4	3	3	Hash Value



Chains

Chaining in R (Relatively Simple Code)

```
# Create a hash table as a list of vectors  
table <- vector(M, mode = "list")  
  
for (i in 1:M) table[[i]] <- c()  
  
# Add an element to the table  
h <- hash(item, M)  
table[[h]] <- c(item, table[[h]])  
  
# Check if an element is in the table  
item %in% table[[hash(item, M)]]
```

Chaining in C++ (Certainly More Complicated!)

```
#include "stdio.h"
#include "stdlib.h"

#define EMPTY -1
#define M 997
#define N 50

class ValueAndPointer
{
public:
    int value;
    ValueAndPointer * next;
};

ValueAndPointer * hash[M];

void InitializeTable()
{
    for (int i = 0; i < M; i++)
        hash[i] = NULL;
}
```

Chaining in C++ (Adding an integer ...)

```
void Insert(int value)
{
    int h = value % M;

    ValueAndPointer ** pointer = &(hash[h]);

    while (*pointer != NULL)
    {
        if (**pointer).value == value)
            // Value is already in the chain ...
            return;

        pointer = &(**pointer).next;
    }

    // Value not found, add a new link to the chain
    *pointer = (ValueAndPointer *) malloc(sizeof(ValueAndPointer *));

    (**pointer).next = NULL;
    (**pointer).value = value;
}
```

Chaining in C++ (Finding an integer ...)

```
bool Find(int value)
{
    int h = value % M;

    ValueAndPointer ** pointer = &(hash[h]);

    while (*pointer != NULL)
    {
        if (**pointer).value == value)
            return 1;

        pointer = &(**pointer).next;
    }

    return false;
}
```

Chaining in C++ (Checking the previous functions)

```
int main(int argc, char ** argv)
{
    srand(123456);

    InitializeTable();

    for (int i = 0; i < N; i++)
    {
        int value = rand() % (N * 10);
        printf("Inserting the value %d into table ...\n", value);
        Insert(value);
    }

    for (int i = 0; i < N; i++)
    {
        int value = rand() % (N * 10);
        printf("Checking for value %d in table ...\n", value);
        if (Find(value))
            printf("    FOUND!!!\n");
        else
            printf("    Not found.\n");
    }
}
```

Properties of Separate Chaining

- If the hashing function results in random indexes...

$$\frac{N}{M}$$

expected number of entries in each chain

$$\binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}$$

number of entries follows Binomial distribution
(well approximated with Poisson distribution)

Interesting Known Properties

- In most slots, no. of entries is close to average

$$e^{-\alpha}$$

probability of an empty slot

$$\sim 1.25\sqrt{M}$$

number of items before first collision

"the birthday problem"

$$M \left(\sum_{i=1}^M \frac{1}{i} \right)$$

number of items before all slots have one item

"the coupon collector problem"

- $\alpha = N / M$ is the load factor...

Notes on Hashing

- **Good performance when**
 - Searching for elements
 - Inserting elements
- **Ineffective when**
 - Selecting elements based on rank
 - Sorting elements

Recommended Reading

- Sedgwick, Chapter 14
- Peterson W. W. (1957) *IBM Journal of Research and Development* 1:130-146