Using Tables

• Problem: determine word frequency in a text message.

• e.g. given

' (mary had a little lamb, little lamb, little lamb. mary had a little lamb whose fleece was white as snow)

• Determine how many times each word occurs.

mary: 2, little: 4, lamb: 4, ...
Solution

- Maintain a table
  - Word is the key
  - Count is the value

- \texttt{cdr} down the message list
  - For each word \( w \)
    - If \( w \) exists in table, then increment its value
    - Otherwise, add entry \((w,1)\) to the table
  - Ignore punctuations
Queues using tables

• empty-Q = empty-table
• To insert item \( M \), insert entry \((N, M)\) into table.
  – \( N \) is a running number that we automatically generate. This serves as a unique key in the table.
    • when table is first created, \( N=0 \)
  – After inserting, we increment \( N \).
  – \( N \) also indicates the order in which the items were inserted.
Queues using tables

• To delete an item from the front of the queue,
  – Lookup key S in the table
  – S is the smallest key
  – Delete that entry
  – Increment S

• To think about:
  – How to implement front-queue?
  – What is the time complexity of these procedures?
Questions of the Day

• Can you implement tables using queues?
  – Hint: consider using 2 queues.
  – Think of how to represent an empty table.
  – How to implement table operations (e.g. lookup, insert!) using queue operations?

• Can you implement table more efficiently?
Recall: not smart fibonacci

```javascript
function fib(n) {
    if (n === 0 || n === 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Time complexity = \( O(\Phi^n) \) (exponential!)

Note: fib(5) computes fib(4) and fib(3), but fib(4) computes fib(3) and fib(2), etc.

Lots of duplicates!

How can we do better?
Computing Fibonacci

```
fib(5)
fib(4)  fib(3)  fib(3)
fib(3)  fib(2)  fib(2)  fib(1)
fib(2)  fib(1)  fib(1)  fib(1)
fib(1)  fib(1)  fib(0)  fib(0)
fib(1)  fib(0)
```

```
1  1  0  1  0
1  0
1  0
```
What’s the **obvious** way to do better?
Remember what you had earlier computed!
Memoization
Memoization

Procedure records, in a local table, values that have previously been computed.
A memoized procedure:

- Maintains a table in which values of previous calls are stored
- Use the arguments that produced the values as keys
When the memoized procedure is called, check table to see if the value exists:

– If so, return value.
– Otherwise, compute new value in the ordinary way and store this in the table.
Implementing Memoization

```javascript
function memoize(f) {
    var table = make_table();
    return function(x) {
        if (has_key(x, table)) {
            return lookup(x, table);
        } else {
            var result = f(x);
            insert(x, result, table);
            return result;
        }
    }
}
```

- **has_key**: checks if table already has key
- **lookup (key, table)**: looks up the value associated with key in table
- **insert (value, key, table)**: Associates value with key in table.
Fibonacci with Memoization

Now that we have memoize, the obvious thing to do is:

```javascript
var memo_fib = memoize(fib);
```

What’s the time complexity now?

Still exponential!

HUH??
Doing it Right!

var memo_fib = 
    memoize(function(n) {
        if (n === 0 || n === 1) { return n; }
        else {
            return memo_fib(n-1) + memo_fib(n-2);
        }
    });

What’s the time complexity now?

O(n) (linear)!

Why??
var memo_fib =
    memoize(function(n) {
        if (n === 0 || n === 1) { return n; }
        else {
            return memo_fib(n-1) + memo_fib(n-2);
        }
    });

Each fib(n) is computed only once!
var memo_fib =
    memoize(function(n) {
        if (n === 0 || n === 1) { return n; }
        else {
            return memo_fib(n-1) + memo_fib(n-2); 
        }
    });

• Efficiency of table lookup is important: Table lookup should be $O(1)$, i.e. hash table.
• What happens to time complexity if table lookup is not constant, say $O(n)$?
Analyzing Memoize

var memo_fib = memoize(function(n) { ... });

G.E. →

param: n
body: if(...)...
Analyzing Memoize

var memo_fib = memoize(function(n) {...});
Evaluating \((\text{memo-fib} \ 3)\)
Evaluating (memo-fib 3)

Memo-fib: memo-fib:

G.E. ->

n: 3

memo_fib(2) + ...

f:

x: 3

result: f(3)

Table:

x: 2

result: f(2)

param: n
body: if(...)...

param: x
body: ...

x: 3

result: f(2)
Evaluating (memo-fib 3)

\[ \text{memo\_fib}(3) = \text{memo\_fib}(2) + \text{memo\_fib}(1) \]

\[ f(3) = f(2) + f(1) \]
Evaluating (memo-fib 3)

if (n == 1) -> n

result: 1

insert(x, result, table)
Evaluating (memo-fib 3)

```javascript
G.E. →

memo_fib:

n: 0
if (n===0...) → n

f:

param: n
body: if(...)...

(1->1)

table:

x: 0
param: x
body: ...

result: 0

(0->0, 1->1)

insert(x,result,table)
```

Evaluating (memo-fib 3)
Evaluating (memo-fib 3)

```
(param: n
body: if(…)
)

 memo_fib:
  (table:
    (0->0, 1->1)
  )

(param: x
body: ...
)

x: 2

(result: 1

(insert(x, result, table)

G.E. -> n: 2 -> 1 + 0

(2->1, 0->0, 1->1)

(0->0, 1->1)
Evaluating \((\text{memo-fib} \ 3)\)

\[\text{memo\_fib:} \]

\[\text{f:} \]

\[\text{table:} \]

\[\text{param: } n \quad \text{body: if(\ldots)…} \]

\[\text{(2->1,0->0,1->1)} \]

\[\text{x:1} \]

\[\text{memo\_fib(1) + …} \]

\[\text{n:3} \]
Evaluating memo_fib(3)

- memo_fib(3)
  - memoized!
    - memo_fib(2)
      - memoized!
        - memo_fib(1)
          - found in table!
        - memo_fib(0)
          - memoized!
      - memo_fib(1)
        - memoized!
    - memo_fib(0)
      - memoized!
function choose(n, k) {
    if (k > 0) { return 0; }
    else if (k===0 || k===n) {
        return 1;
    } else {
        return choose(n - 1,k) +
            choose(n - 1, k - 1);
    }
}
Remember Count-Change?

Consider one of the elements x. x is either chosen or it is not.

Then number of ways is sum of:

– **Not chosen.** Ways to choose k elements out of remaining \( n-1 \) elements; and

– **Chosen.** Ways to choose \( k-1 \) elements out of remaining \( n-1 \) elements
Another Example:  
(n choose k)

function choose(n, k) {
    if (k > 0) { return 0; }
    else if (k===0 || k===n) {
        return 1;
    } else {
        return choose(n - 1,k) + 
        choose(n - 1, k - 1);
    }
}

• What is the order of growth?  
• How can we speed up the computation?
Memoization!
Recall

```javascript
function memoize(f) {
    var table = make_table();
    return function(x) {
        if (has_key(x, table)) {
            return lookup(x, table);
        } else {
            var result = f(x);
            insert(x, result, table);
            return result
        }
    }
}
```
Would a two-variable version help?

```javascript
function two_var_memoize(f) {
    var table = make_2d_table();
    return function(x, y) {
        if (has_key_2d(x, y, table)) {
            return lookup(x, y, table);
        } else {
            var result = f(x, y);
            insert(x, y, result, table);
            return result;
        }
    }
}
```
New memo-choose

```
var memo_choose
    = two_var_memoize(choose)
```

BUG!!

Just checking if people were paying attention 😊
Doing it Right …

```javascript
var memo_choose2 =
    two_var_memoize(
        function(n, k) {
            if (k > 0) { return 0; }
            else if (k===0 || k===n) {
                return 1;
            } else {
                return memo_choose2(n - 1,k) +
                    memo_choose2(n - 1,k - 1);
            }
        }
    );
```

• What is the order of growth?
Questions of the Day

• Recall the `count_change` problem
  – what is the order of growth?
• Can we use memoization to compute it more efficiently?
• If no, WHY? If yes, what is the new order of growth?
Summary

• Memoization dramatically reduces computation.
  – Once a value is computed, it is remembered in a table (along with the argument that produced it).
  – The next time the procedure is called with the same argument, the value is simply retrieved from the table.

• `memo_fib` takes time $= O(n)$

• `memo_choose` takes ?? time?
Have a great weekend!

Have fun building your robots …