CSE 373: Review

Pete Morcos
University of Washington
3/29/2000

http://www.cs.washington.edu/education/courses/cse373/00sp

Important Math Stuff

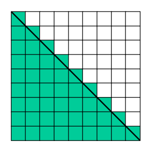
- You need to be familiar with:
  - Exponents
  - Logarithms
  - Modulo arithmetic
  - Series
  - Recursion
  - Proof techniques, especially induction
- We'll talk about some today, but read section 1.2 in book

Logs, Exponents

- Since we love binary numbers, we almost always want to think about things in base 2
- Thanks to the following formulas...
  - $A^B = (2^{\log A})^B = 2^{\log A * B}$
  - $\log_X Y = \log_2 Y / \log_2 X$
- ...we know that any base is equivalent to base 2 within a constant factor somewhere in the formula
- Base 2 is always assumed

Series - Arithmetic

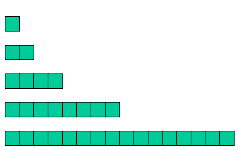
- $1 + 2 + 3 + 4 + ... + N = ?$



$$\sum_{i=0}^{N} i$$

Series - Geometric

- $1 + 2 + 4 + 8 + ... + 2^N = ?$



$$\sum_{i=0}^{N} 2^i$$

- These two series are very common—memorize them.

Recursion

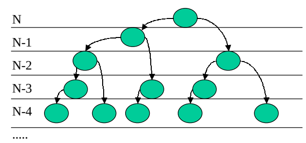
- A function that calls itself is said to *recurse*
- Sometimes a natural way to express a repetitive algorithm, as opposed to using explicit iteration (for loops, while loops)
- A classic example: the Fibonacci numbers
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
  - First two are defined to be 1
  - Rest are the sum of the preceding two: $F_i = F_{i-1} + F_{i-2}$

Recursive Fibonacci

```
int fib(int i) {
   if (i<0) return 0;  // error value
   if (i==0 || i==1) return 1;
   else return fib(i-1) + fib(i-2);
}
```

- Easy to write, looks a lot like the mathematical definition
- There is a big problem, though; what is it?

Fibonacci Calls



Iterative Fibonacci

```
int fib_it(int i) {
   int fib1 = 1, fib2 = 1, fibj = 1;
   if (i<0) return 0;  // error value
   for (int j=2; j<=i; j++) {
        fibj = fib1 + fib2;
        fib2 = fib1; // shift values for next iteration
        fib1 = fibj;
   }
   return fibj;
}
```

- We have to do more bookkeeping this way.

Recursion Summary

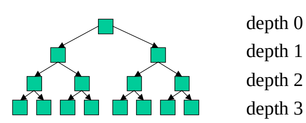
- Be sure to get the base case(s) correct!
- Each step must get you closer to the base case.
- Function calls aren’t free; actually a fairly expensive operation
- Recursion can be very neat, but beware of generating huge numbers of calls
- Also realize that there is a hidden space cost in the system’s stack; might be more than you need

Proof by Induction

- How do you create an infinite number of specific proofs? (often a function of $n \geq 0$)
- As with recursion: base case, self-referencing case
- Base case is “proven” by inspection
- All other cases proven in this standard way:
  - Assume all cases 1, 2, ..., k-2, k-1, k are true
  - Given that, show that case k+1 is true
- Together, these prove it for all values of n

Proof by Induction Example

- A complete binary tree of depth d contains 2^{d+1}-1 nodes



The Proof

- Base case, $k = 0$?
- Inductive step, given 1..k, show k+1?

- Other proof techniques: contradiction, counterexample, inspection

C Review

- C

```
typedef struct {
   int x,y,z;
} node;
```

function args that get changed:
pointer vars: `int *px`

`malloc(), free()`

`char name[100];`

```
printf("age:%d, name:%s\n",
   age, name);
```

- C++

```
class node {
public:
   int x,y,z;
};
```

reference vars: `int& x`

`new, delete`

`String name;`

```
cout << "age:" << age <<
   "name:" << name << "\n";
```


Pointers and Memory

- Recall that memory is a one-dimensional range of bytes, each with an address
- Pointer vars contain an address, rather than an int/char/float

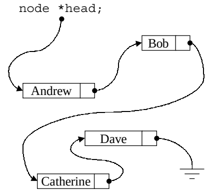
```
int *pint, y, *pint2;     // '*' needed on each ptr variable
y = 3;
pint = &y;                // assign address of y to pint
*pint = 42;               // put 42 in location pint points to
printf("%d", y);          // prints out 42
```

- Vital to know difference between address & value
- Ptrs to ptrs: "`int ***pppint`"
- What happens if you say "`*pint2 = 43;`" ?

Thinking about Pointers

- Arrows

`node *head;`



- Memory values

0	
1000	
2000	Bob, 52030
...	
12390	head = 21000
12391	
12392	
12393	
...	
21000	Andrew, 2000
...	
25007	Dave, 0
26010	
52030	Catherine, 25007

Ptrs take up 4 bytes

Memory Management

- When you declare a variable inside a procedure, space is allocated for it on the stack
- We'll often need to allocate an unknown number of variables at runtime

```
typedef struct _node {int value; struct _node *next;} node;
node *curnode = malloc(sizeof(node));
for (int i=0; i<1000; i++) {
   curnode->next = (node*)malloc(sizeof(node));
   curnode = curnode->next;
}
curnode->next = NULL;
```


malloc, free

- malloc allocates a specified number of bytes
- Use the `sizeof` operator to compute how many
- malloc returns a "`void *`", the generic pointer type
- Cast operation "`(node*)`" tells compiler to pretend this variable is a different type
- To deallocate, call free() and pass a pointer to an object allocated with malloc()
- Don't mix up new/delete and malloc/free pairs!
- You may use whichever style you prefer

Think about for next time

- Ops for linked lists:
  - add(char *newname)
  - remove(node *node_to_kill)
  - find(char *searchname)
  - removeAll(node *head_of_list)
  - getNext(node *current_node)
  - getPrev(node *current_node)
- What are the costs?