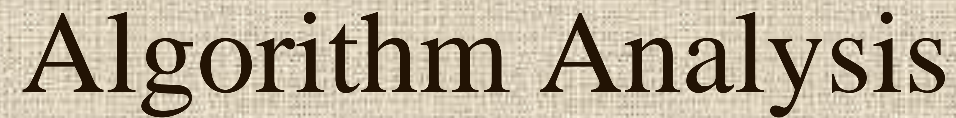


More details and examples

Another For Loop Example

```
for(int i = 1; i<= n; i++)              /*n times*/
        for(int j = 1; j<=i; j++)       /*?*/
            sum++                       /*fixed time for each iteration*/
```

1. The outer loop happens n times.
2. The inner loop happens 1 time when i = 1, then 2 times when i = 2, then 3 times when i = 3, etc.

   So sum++ is executed a total of 1 + 2 + 3 + 4 + 5 + … + n times

   or

$$T(N) = \sum_{j=1}^{j=n} j = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

“arithmetic” series
(Remember? oh, yeah…)

3. So the growth rate is $O(N^2)$.

Not **All** “For” Loops Are O(N)

```
for(int j = 1; j< n; j*=2)          /*how many times?*/
        sum++;                      /*fixed time for each iteration*/
```

1. Notice j doubles in size each time through loop.
   j=1, j=2, j=4, j=8… Grows exponentially
2. Let i be the number of times the loop has executed.
   Then $j = 2^{i-1}$.
3. So when does the loop stop? When $j = 2^{i-1} = n$.
4. $\log(2^{i-1}) = \log(n)$
   $(i-1) \log(2) = \log(n)$
   $i = (\log(n) / \log(2)) + 1$ (that’s the # of times the loop executed)
5. So sum++ is executed $(\log(n) / \log(2)) + 1$ times.
6. So the growth rate is O(logN). (…throw away the constants)

Cool, n’est pas?!

Remember this series?

- Recall this series from review? We'll need it for the next example.

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

Proof:

$$S = \sum_{i=0}^{N} A^i = 1 + A^1 + A^2 + A^3 + \ldots + A^N$$

$$AS = A^1 + A^2 + A^3 + \ldots + A^N + A^{N+1}$$

$$S - AS = 1 - A^{N+1}$$

$$S = \frac{A^{N+1} - 1}{A - 1}$$

(proof by construction)

Another For Loop Example

```
for(int j=1; j<n; j*=2)          /*already showed you this!*/
      for(int k=1; k<=j; k++)     /*runs how many times???*/
         sum++;                   /*fixed time for each iteration*/
```

1. The inner loop executes 1 time, then 2 times, then 4 times, then… A total of $1 + 2 + 4 + 8 + 16 + \ldots + 2^{i-1}$ where i is the total number of times the outer loop runs.
2. But wait! We just showed previously that the outer loop runs $i = \log(n)/\log(2) + 1$ times. So $i = \log_2 n + 1$. (…remember that math rule?)
3. So the total number of times that sum++ is executed is

$$T(n) = \sum_{i=1}^{i=1+\log_2 n} 2^{i-1}$$

ugh?

Another For Loop Example (cont.)

```
for(int j=1; j<n; j*=2)          /*already showed you this!*/
    for(int k=1; k<=j; k++)      /*runs how many times???*/
        sum++;                   /*fixed time for each iteration*/
```

So this runs in

$$T(n) = \sum_{i=1}^{i=1+\log_2 n} 2^{i-1} = \sum_{i=0}^{i=\log_2 n} 2^i$$

Recall

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1} = \frac{2^{N+1}-1}{2-1} = 2^{N+1}-1$$

$$= 2^{\log_2 n+1} - 1 = 2 \cdot 2^{\log_2 n} - 1$$

$$= 2n - 1 = O(n)$$

Compare Two Algorithms: Exponentiation #1

/*calculate x^n by brute force*/

```
public long pow(long x, int n)
{
    long product = 1;
    for(int i=1; i<=n; i++)
    {
        product *= x;
    }
    return product;
}
```

Growth rate is O(N). Why?

Compare Two Algorithms: Exponentiation #2

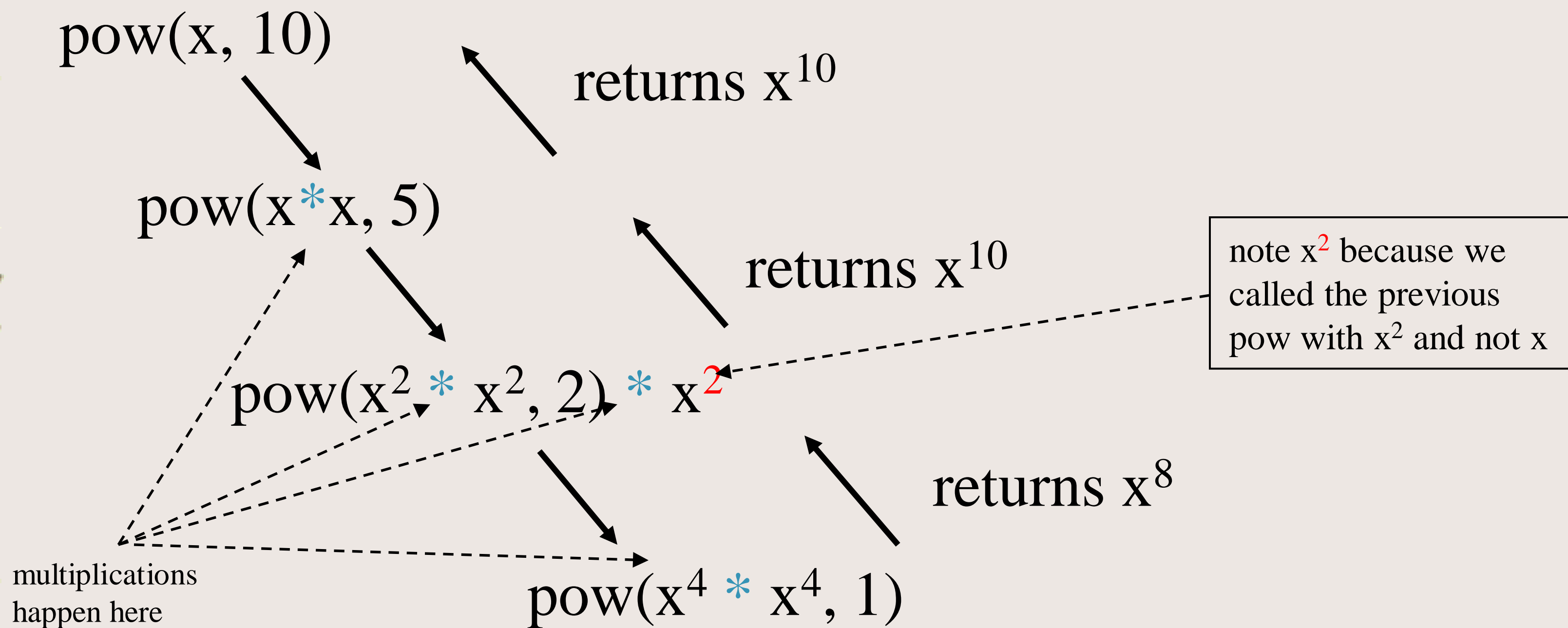
```
/*calculate x^n with recursion*/

public long pow(long x, int n)
{
    if(n==0)
        return 1;
    if(n==1)
        return x;
    if(n%2 == 0)     /*if n is even*/
        return pow(x*x, n/2);
    else
        return pow(x*x, n/2) *x;
}
```

Compare Two Algorithms: Exponentiation #2 (cont.)

- Example: pow(x, 1) … returns x.
  - » total of 0 multiplications
- Example: pow(x, 2)
  - calls pow(x^2, 1)
    - » calls returns x^2
    - » total of 1 multiplication
- Example: pow(x, 3)
  - calls pow(x^2, 1) * x
    - » pow(x^2, 1) returns x^2
    - » and that gets multiplied by x to return x^3
    - » total of 2 multiplications

Compare Two Algorithms: Exponentiation #2 (cont. 2)

Example: x^{10} takes only 4 multiplications.



- That’s 4 multiplications versus 10. (And 3 divisions.)
- Recursion algorithm is looking faster!

Compare Two Algorithms: Exponentiation #2 (cont. 3)

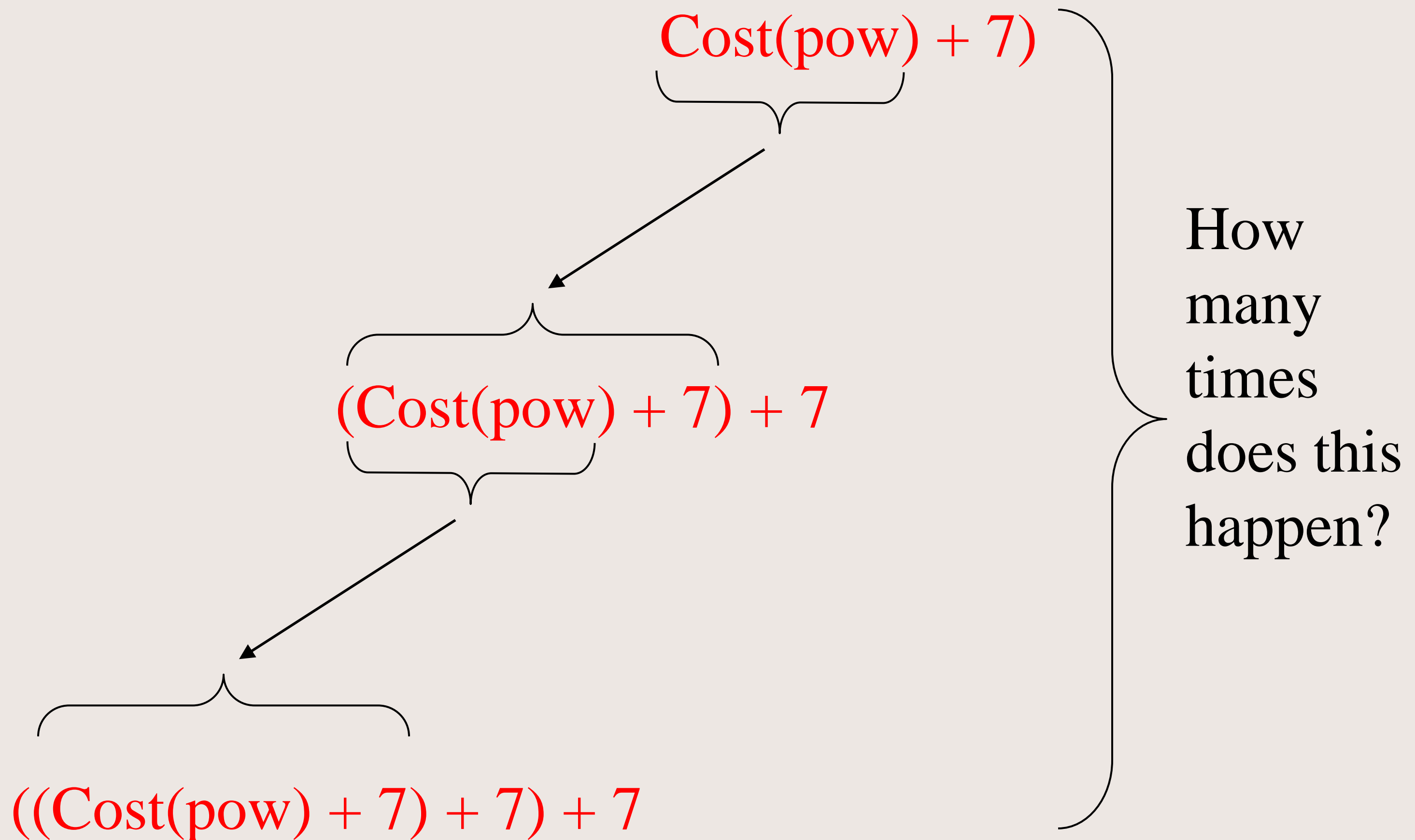
REMEMBER RULE: go with the more expensive of the if/else

Run time calculation:

Code	Cost
public long pow(long x, int n)	
{	
if(n==0)	cost 1
return 0;	cost 0
if(n==1)	cost 1
return x;	cost 0
if(n%2 == 0) /*if n is even*/	cost 2
return pow(x*x, n/2);	cost of pow + 1 mult + 1 div
else	cost 0
return pow(x*x, n/2) * x;	cost of pow + 2 mult + 1 div
}	
total cost:	cost of pow + 7

Compare Two Algorithms: Exponentiation #2 (cont. 4)

So the total cost is



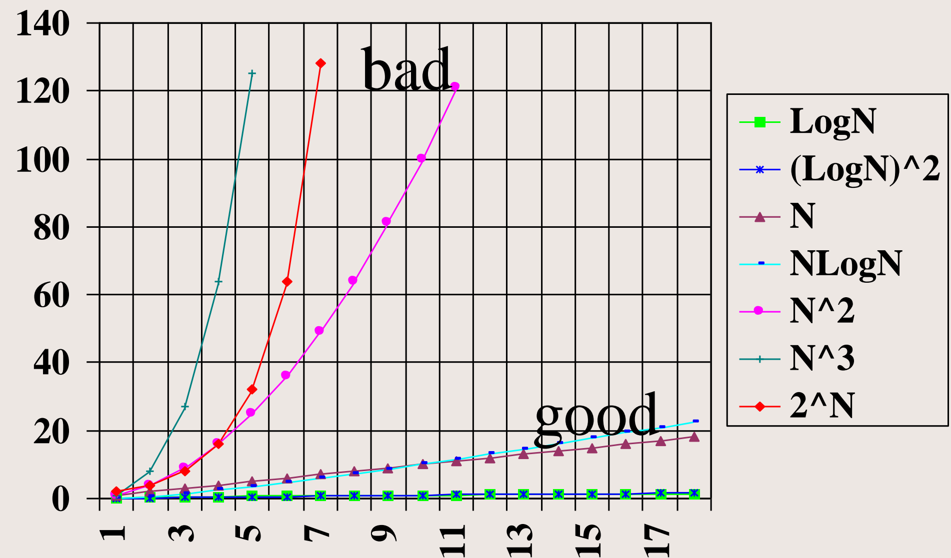
Compare Two Algorithms: Exponentiation #2 (cont. 5)

- So how many times does pow get called?
- Suppose start with n = 16.
- Then n/2 gives … n= 8, n= 4, n = 2, n = 1.
- Look familiar? That happens log(n) times!
  (go back and look at the for loop example)

- So, sum up 7 a total of log(n) times
- T(n) = 7 log(n) = O(log(n))

Compare Two Algorithms: Final Result

- O(N) versus O(log(N))
- Which is faster?
- Recursion wins!



Look at “Real” Code

- Code from Numerical Recipes.
    - In Pascal, C, Fortran!
- Calculate run time.
- Compare.