6/15/2020 Recursion #1 - Part B

Monday, June 15, 2020 8:27 PM

Modular Exponentiation

Regular exponentiation problem is calculate $base^{exponent}$.
The modular exponentiation problem is calculate
$base^{exponent}$ mod n, where n is a pre-determined modulus value.
Answer to the first query is unbounded, but the answer to the second is always in between 0 and n-1, inclusive.

Why is this useful??? - This is one of the most important calculations that makes public key cryptography doable in practice. (RSA encryption amounts to calculating a modular exponent with integer values with hundreds (or even thousands) of digits.

Our previous recursive algorithm:

```
int modExp(int base, int exp, int mod) {

   if (exp == 0) return 1;

   return (modExp(base, exp-1,mod)*base)%mod;
}
```

Run-time: each time we subtract 1 off the exponent, so the run time is simply the time of the multiplication and mod, and that whole time multiplied by the VALUE of exp. O(exp*mult time)…the problem is that exp might be a number with hundreds of digits, so this would take, thousands of years to run….our computer would probably die before then…

Now, let's consider A DIFFERENT way to split up an exponentiation problem…

Pretend that exp is even.

$base^{exp} = base^{exp/2} * base^{exp/2}$

Why is this useful???

The reason is that the first and second term are the same value (on the right hand side, so I DON'T HAVE to recompute the second value…

In regular exponentiation, this is helpful but the calculation is prohibitive because the sheer size of the number is too big.

But in modular exponentiation, the result for the calculation must be no bigger than the mod value, so the size of the answer is limited and this results in a huge speed up.

Idea is as follows:

if the exponent is even, calculate the square root of the desired value, store the answer in a variable and square that variable.

2^{32} mod 19

2^{32} --> 2^{16} * 2^{16} (but we only make one recursive call)
--> 2^8 --> 2^4 --> 2^2 --> 2^1 (returns 2)
2^2 returns 4
2^4 returns 16
2^8 return 256 mod 19 --> returns 9
2^{16} returns 9*9 = 81 81 mod 19 --> return 5
2^{32} returns 5*5 mod 19 = 6

The savings pile up, particularly at the end, where instead of repeating 16 multiplications and mods, we did just one (times 5).

When the exponent is odd, we do the usual recursive breakdown…

Imagine breaking down 1000000
1000000 --> 500000 --> 125000 --> 62500 --> 31250 --> 15625 --> 15624 --> 7812 --> 3906 --> 1953 --> 1952 --> 976 --> 488 --> 244 --> 122 --> 61 --> 60 --> 30 --> 15 --> 14 --> 7 --> 6 --> 3 --> 2 --> 1 (25 steps total instead of 1,000,000…huge savings…)

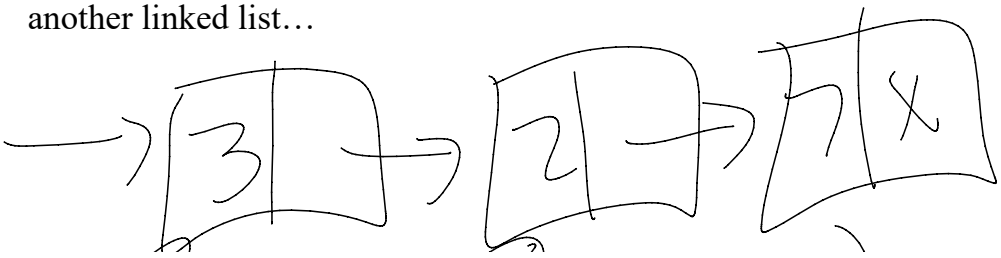
In general, we divide by 2 every other step, in the worst case, because when you subtract 1 from an odd number, you get an even number, so there are never two calls in a row with an odd number.

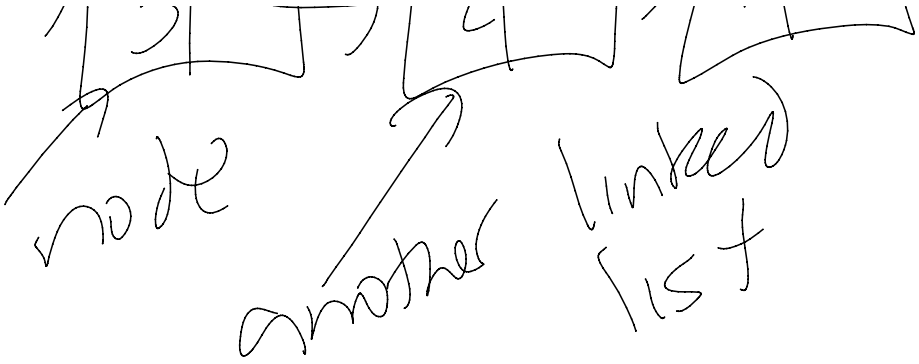
WE would have to divide exp by 2, log exp times to get down to 1.

the run time of this is O(log exp * mult time). Multiplication time is O(digits) in the mod value, which is log exp…

Most linked list stuff can be done recursively

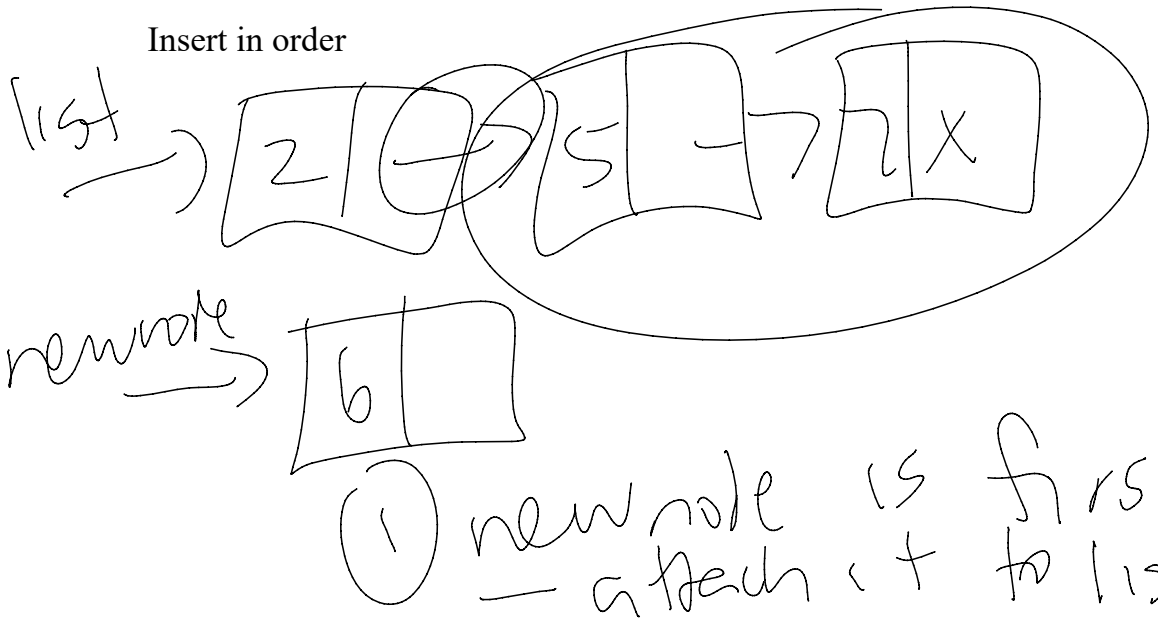
At the end of the day, a linked list is the first term, with a pointer to another linked list…





Most linked list code that is recursive, usually calls the recursive function on front->next….

Insert in order



① new node is first
— attach it to list

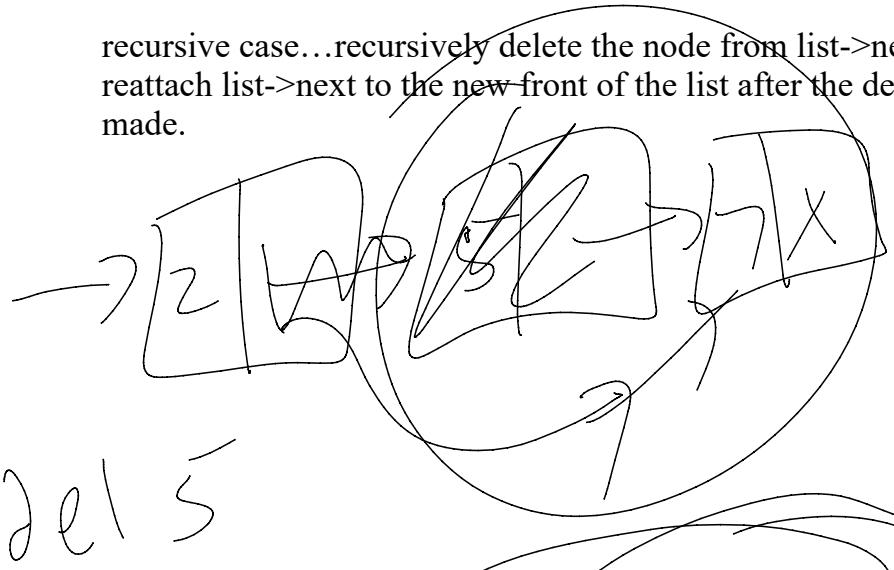
② recursively insert into list->next.
then reassign list->next to the new front of the list.

Recursive Delete

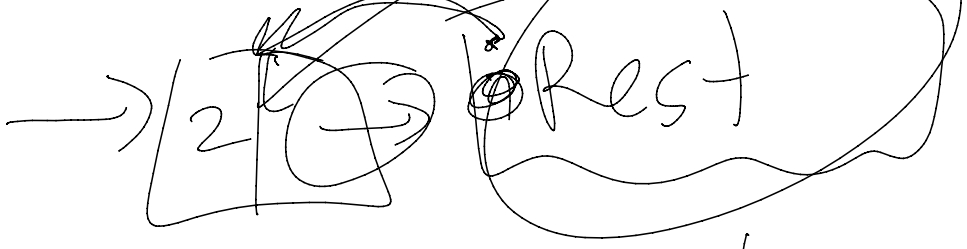
base case: delete the front node…we free it, and then we return the new front of the list.

base case: delete the front node…we free it, and then we return the new front of the list.

recursive case…recursively delete the node from list->next, then reattach list->next to the new front of the list after the deletion was made.



Recursive Reverse:



newfront = reverse(list->next)
attach end of the list to 2.
→ O(n) to get to end!

trick is as follows to avoid O(n) work:

Do front->next->next = front;