Lecture 12 - Divide & Conquer

Eric A. Autry

Last time: Review of Data Structures

This time: Recursion and Recurrence Relations

Next time: Sorting

HW 4 due today.

HW 5 assigned today and due next Thursday, the 25th.

Project 1 to be assigned next week.

Hash Tables

Hash Tables are used to map keys to values.

A hash function is used to compute the index of a 'bin' where a given value can be found.

- A good hash function should avoid collisions and clustering.

The load factor of a hash table is defined as

$$\lambda = \frac{n}{b},$$

where n is the number of keys, and b is the number of bins.

Want a good balance between taking up too much space (small load), and larger access times (large load)

- Ex: Java 10 HashMap load factor default is 0.75

Hash Tables - Implementation

There are two primary methods of implementing a hash table:

- Separate Chaining (typically with linked lists)
  - Each bin can store as many values as it wants.
  - When multiple values all correspond to the same key, all of those values are stored in the same bin.
  - Requires searching through the bin to find a given value.
- Open Addressing (typically with an array)
  - Each bin can hold only a single value.
  - If multiple values all correspond to the same key, a new bin has to be found.
  - When such a collision occurs, consider successive bins until there is an opening, then store the new value there.
  - To find a value, go to its 'original' bin, then search until you either find the value or find a blank space.
  - Good for storing 'small' value due to better locality for caching.
  - Requires a very good hash function and a low load factor.

Aside: Special Properties of Logarithms

- $\log x + \log y = \log(xy)$
- $\log x - \log y = \log(x/y)$
- $\log x^n = n \log x$
- $\log_a x = \frac{\log_b x}{\log_b a}$
- $a^{\log x} = x^{\log a}$

Aside: Geometric Series

We have already seen

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1.$$

Let's generalize this for all geometric series:

$$1 + a + a^2 + a^3 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1},$$

where

$$n \in \mathbb{Z}, \quad n \geq 0, \quad \text{and} \quad a \in \mathbb{R}, \quad a \neq 1.$$

Can be proved through induction on n.

Fibonacci Numbers

Fibonacci c. 1175 - c. 1250

Consider rabbit populations:

- Assume that rabbits can be children or adults.
- Assume that it takes 1 year for a rabbit to mature.
- Assume that each year, every adult pair of rabbits produces a pair of children rabbits.
- Assume that there are always exact male-female pairs.

How many rabbits are produced when starting from a single pair of children rabbits?

Year 1: 1 pair of adults

Year 2: 1 pair of adults + 1 pair of children = 2 pairs

Year 3: 2 pairs of adults + 1 pair of children = 3 pairs

Year 4: 3 pairs of adults + 2 pairs of children = 5 pairs

Fibonacci Numbers

This gives the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

We can define the Fibonacci numbers with the following rule

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = 1, \quad F_0 = 0.$$

It turns out that $F_n \sim 2^{0.694n}$.

Simple **recursive** algorithm for computing Fibonacci numbers:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Fibonacci Numbers

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Three questions we always ask about algorithms:

1. Is it correct?
2. How much time does it take?
3. Can we do better?

The function fib is clearly correct. So what is its runtime?

$$T(n) = T(n-1) + T(n-2) + 3 \quad \text{for } n > 1.$$

But this means that $T(n) \geq F_n \sim 2^{0.694n}$. Oh no!

Fibonacci Numbers

How long does it take to compute F_{200}?

$$T(200) \geq F_{200} \geq 2^{138}.$$

The fastest computer in the world, the American machine 'Summit', can perform 122.3 PFLOPS.

That's $\sim 2^{56.8}$ FLOPS.

So F_{200} will take over 2^{81} seconds, which is over 5 million times the age of the universe...

Exponential growth is BAD

Can we do better? Yes, next week.

Complex Arithmetic

When counting operations for Fibonacci, we treated each addition as constant time.

Because I have told you that basic arithmetic operations are all constant time.

This is certainly true for basic data types like ints, floats, or doubles.

After all, how long does it take to add two 32-bit numbers?

That seems like a relatively small number. Computers can do that *constant* amount of work quite quickly.

Complex Arithmetic - Addition

But what if I want to add two very large numbers together as we do when computing large Fibonacci numbers?

Say my two numbers each take up 1 million bits. How long will adding them take?

Say my two numbers each take up n bits. How long will adding them take?

So arithmetic can only be considered constant time for basic data types with fixed sizes.

Aside: Bit Shifting

Before we continue and examine bit-wise multiplication of large numbers...

There is a useful trick in binary to quickly multiply or divide by 2: bit shifting.

What is 10×2?

$$10 = 1010_2, \quad 20 = 10100_2.$$

We just shifted all of the bits to the left and put a 0 on the right.

To divide by 2, we shift the bits to the right. Note that this always rounds down if the number was odd.

Note that shift by a **single bit** is a constant time operation $O(1)$.

Complex Arithmetic - Multiplication

How long does it take to multiply two large numbers that each have n bits?

Using the standard multiplication algorithm, we have to compute:

- The first number times each bit of the second number.
- The sum of those results (after accounting for the shift).

That first step is easy because we are either multiplying by 0 or 1, i.e., each multiplication by a single bit takes a single operation, for a total of n.

For the second step, notice that we are adding n number of length up to $2n$ bits. Adding two at a time, we have $n - 1$ additions that are each $O(n)$, giving a total of $O(n^2)$.

Divide & Conquer

Let's try a new approach to multiplication: we are going to break each number in half!

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R$$

Now when we multiply:

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Note that each addition is $O(n)$, while multiplying by powers of 2 is also $O(n)$ because they are just bit-shifts.

Computing the 4 values $x_L y_L$, $x_L y_R$, $x_R y_L$, and $x_R y_R$ can be done recursively. Note that these numbers are each of size $n/2$.

$$T(n) = 4T(n/2) + O(n)$$

Divide & Conquer

```
def mult(x, y):

[xL, xR] = break(x) # Break x in half in O(1)
[yL, yR] = break(y) # Break y in half in O(1)

LL = mult(xL, yL) # Recursive call of size n/2
LR = mult(xL, yR) # Recursive call of size n/2
RL = mult(xR, yL) # Recursive call of size n/2
RR = mult(xR, yR) # Recursive call of size n/2

sLL = Lshift(LL, n) # (2^n)*LL with O(n) shift
M = LR + RL         # Add the middle terms in O(n)
sM = Lshift(M, n/2) # (2^n/2)*M with O(n) shift

return (sLL + sM + RR) # Add the pieces in O(n)

end mult
```

Divide & Conquer

This code gives us the runtime:

$$T(n) = 4T(n/2) + O(n)$$

For multiplying two numbers of size n, we had 4 recursive calls of size $n/2$, hence the $4T(n/2)$ term.

Aside from the recursive calls, there were:

- Two $O(1)$ constant time operations that broke the two input numbers into their respective halves.
- Two $O(n)$ bit shifts.
- Three $O(n)$ additions.

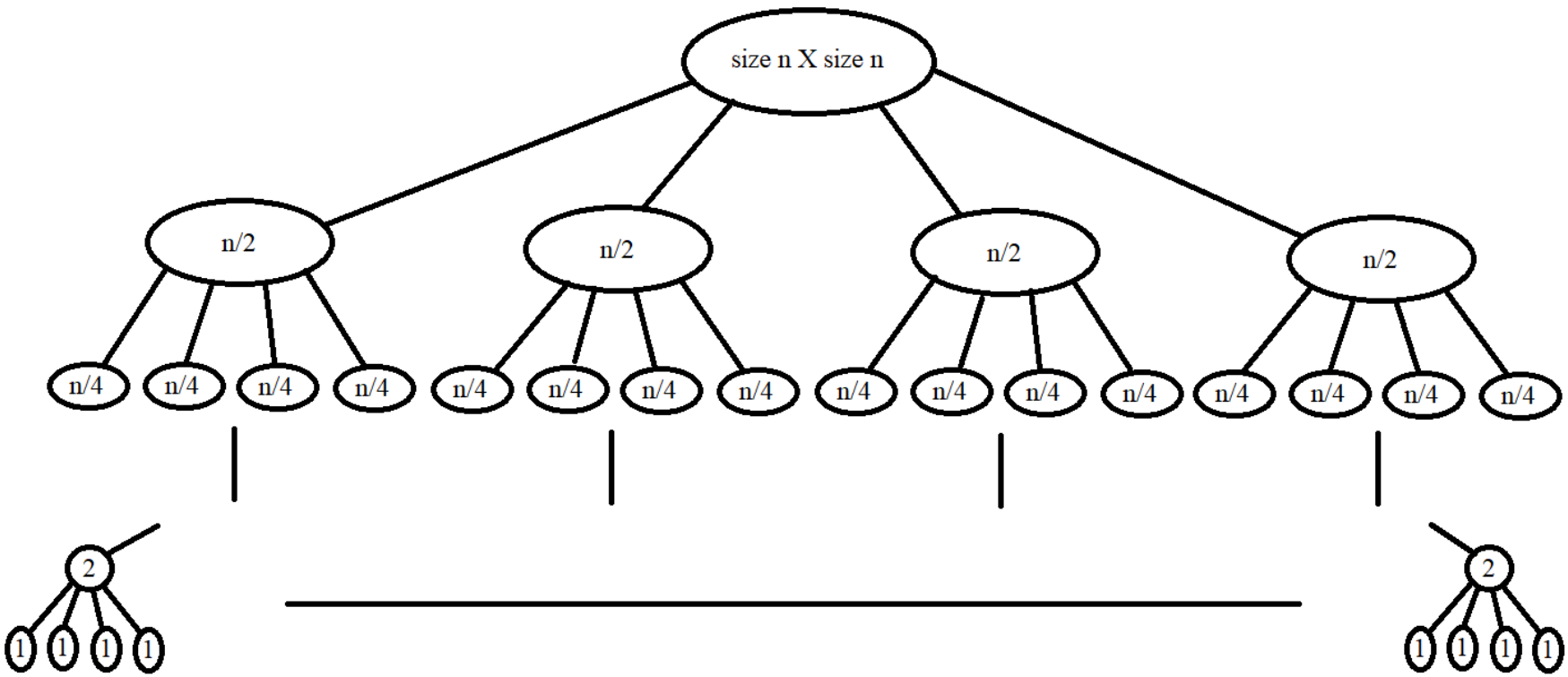
So, aside from the recursive calls, our algorithm required $O(n)$ total work to combine the results of the recursive calls into the return value.

Work Tree Method

$$T(n) = 4T(n/2) + O(n)$$

How to count the total time?

We can think of recursive calls as a tree:



Work Tree Method

We can create a table with info about this tree:

node size	# of nodes	work/node	total work
$s = n$	1	$c \cdot s$	$c \cdot n$
$s = n/2$	4	$c \cdot s$	$4 \cdot c \cdot \frac{n}{2}$
$s = n/4 = \frac{n}{2^2}$	$16 = 4^2$	$c \cdot s$	$4^2 \cdot c \cdot \frac{n}{2^2}$
$\dots$	$\dots$	$\dots$	$\dots$

Work Tree Method

Where did the c come from?

Consider the work done by a node of size s.

There will be 4 recursive calls of size $s/2$, but that work will be counted further down in the table.

Aside from the recursive calls, the algorithm required $O(s)$ work.

But, $O(s) \sim cs$ where c is some positive constant number.

Work Tree Method

We will now assume that n is a power of 2, i.e., $n = 2^k$.

Adding up the total work at each level of recursion gives us:

$$cn + 4c\frac{n}{2} + 4^2 c\frac{n}{2^2} + \cdots + 4^k c\frac{n}{2^k}$$

Which can be rewritten as

$$cn\left(1 + \frac{4}{2} + \frac{4^2}{2^2} + \cdots + \frac{4^k}{2^k}\right) = cn\left(1 + 2 + 2^2 + \cdots + 2^k\right)$$

$$= cn\left(2^{k+1} - 1\right) = cn(2n - 1) \in O(n^2).$$

We haven't done any better than before...

Powers of 2

Why are we allowed to simply assume that n is a power of 2?

How can we be sure that our analysis is still valid when n is not a power of 2?

We make a very important assumption: **the runtime of an algorithm is a monotonically increasing function of the input size** n.

This means that

$$a \leq n \leq b \quad \Rightarrow \quad T(a) \leq T(n) \leq T(b).$$

Now, when we bound both $T(a)$ and $T(b)$ by the function $f(n)$, it must be the case that $T(n)$ is also bounded by the same $f(n)$.

Otherwise, there must have been a point where $T(n)$ was larger than $T(b)$, which contradicts the fact that the runtime is a monotonically increasing function.

Gauss

Can we do better?

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss had an important realization:

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

So we can instead compute

$$xy = 2^n x_L y_L + 2^{n/2}\Big((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R\Big) + x_R y_R$$

This requires 6 addition/subtraction operations for $O(n)$ cost, 2 bit-shift operations for $O(n)$ cost, but **only 3** multiplications of size $n/2$.

$$T(n) = 3T(n/2) + O(n)$$

Gauss

```
def mult2(x, y):

[xL, xR] = break(x) # Break x in half in O(1)
[yL, yR] = break(y) # Break y in half in O(1)

xx = xL + xR # Addition of size n/2, O(n)
yy = yL + yR # Addition of size n/2, O(n)

LL = mult2(xL, yL) # Recursive call of size n/2
MM = mult2(xx, yy) # Recursive call of size n/2
RR = mult2(xR, yR) # Recursive call of size n/2

sLL = Lshift(LL, n) # (2^n)*LL with O(n) shift
M = MM - LL - RR    # Compute middle term in O(n)
sM = Lshift(M, n/2) # (2^n/2)*M with O(n) shift

return (sLL + sM + RR) # Add the pieces in O(n)
```

Gauss

This code gives us the runtime:

$$T(n) = 3T(n/2) + O(n)$$

For multiplying two numbers of size n, we had only 3 recursive calls of size $n/2$, hence the $3T(n/2)$ term.

Aside from the recursive calls, there were:

- Two $O(1)$ constant time operations that broke the two input numbers into their respective halves.
- Two $O(n)$ additions before the recursive call.
- Two $O(n)$ subtractions.
- Two $O(n)$ bit shifts.
- Two $O(n)$ additions.

So, aside from the recursive calls, our algorithm required $O(n)$ total work to combine the results of the recursive calls into the return value.

Work Tree Method

$$T(n) = 3T(n/2) + O(n)$$

We can create a table with info about the work tree:

node size	# of nodes	work/node	total work
$s = n$	1	$c \cdot s$	$c \cdot n$
$s = n/2$	3	$c \cdot s$	$3 \cdot c \cdot \frac{n}{2}$
$s = n/4 = \frac{n}{2^2}$	$9 = 3^2$	$c \cdot s$	$3^2 \cdot c \cdot \frac{n}{2^2}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

Work Tree Method

We will now assume that n is a power of 2, i.e., $n = 2^k$.

Adding up the total work at each level of recursion gives us:

$$cn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^k\right) = cn\frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1}$$

$$= 2cn\left(\frac{3}{2}\left(\frac{3}{2}\right)^k - 1\right).$$

But $k = \log_2 n$, and so

$$= 2cn\left(\frac{3}{2}\left(\frac{3^{\log_2 n}}{n}\right) - 1\right) = 3cn^{\log_2 3} - 2cn$$

$$\in O(n^{\log_2 3}) \sim O(n^{1.59}).$$