LECTURE 1 (August 27th)

RECURSION

As a warmup, we will talk about recursion, which is one of the most important design tools for algorithms.

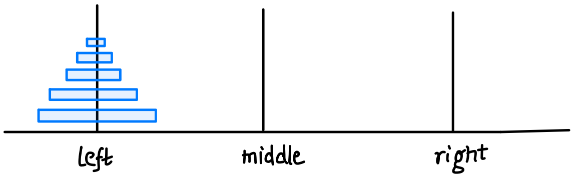
The basic idea is that you want to solve an instance of a problem and the way you solve it is by not solving it but by making a little bit of progress until we have one or more smaller instances of the problem, which you solve by delegating to the "recursion fairy", or just brute-force it if is constant-sized.

As a canonical example, consider the Tower of Hanoi problem.

Tower of Hanoi

This is a physical puzzle designed in late 1800s by the French mathematician Edward Lucas

There are three pegs and on one of them, there is a stack of circular discs of different sizes stacked up so that the sizes increase from top to bottom



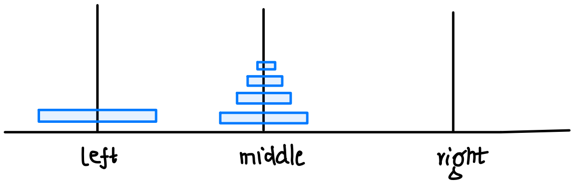
Goal : Move all discs from left to the right peg by following these rules :

1. we can only move one disc at a time
2. a disc can only be on top of a larger disc

How do we do this ?

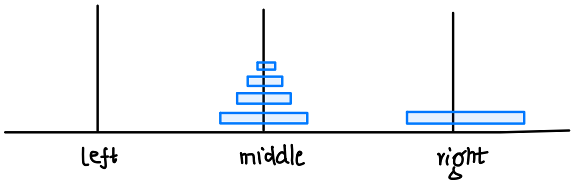
Here is how we are going to think about it

We want to move the bottom most disc to a different peg. In order to do this, we have to get the other discs out of the way. This is the same problem with one less disc.

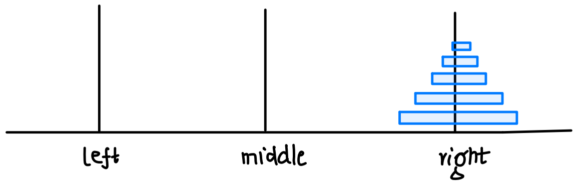


STEP 1 Recursion Fairy solves it

We can assume that the recursion fairy solves the smaller problem and our only job now is move the largest disc to the right peg



STEP 2 Our job



STEP 3 Recursion Fairy solves it

The algorithm formally to move n discs from source (src) to destination (dst) peg using a temporary (tmp) peg is as follows.

```
Hanoi( n, src, dst, tmp)
If n > 0 :
    Hanoi ( n-1 , src, tmp, dst)
    Move disc n from src to dst
    Hanoi ( n-1 , tmp, src, dst)
```

What about the base case ?
There has to be a largest disc, so n has to be a positive integer, otherwise it does not make sense to call Hanoi (n-1,.....)

This means that if the condition $n > 0$ is violated, we have to solve the only remaining case of $n = 0$ differently. But in this case, there are no discs, so there is nothing to do and the algorithm above will work.

Advice : Believe in the recursion fairy and check assumptions about boundary cases

We also have to analyze the algorithm :

1. Prove that it is correct – this is easy to do here, so this is for you to think about
2. Running Time, i.e., the number of moves

Let $T(n)$ = number of moves for n discs

For our algorithm, $T(n) = \begin{cases} 0 & \text{if } n = 0 \\ T(n-1) + 1 + T(n-1) & \text{otherwise} \end{cases}$

This recurrence relation is not useful to compare running time of algorithms, so we want to get a closed form big-O expression for $T(n)$ by opening up the recurrence

Let's review how to solve recurrences. We will see another way later, but the most obvious way is to guess and check.

How to guess ? Use wikipedia which says $T(n) = 2^n - 1$ or write down a few small values and make a guess by looking at the pattern.

n	0	1	2	3	4	5	....
$T(n)$	0	1	3	7	15	31	....

We still need to verify that the guess is correct.
How do we prove this ? Induction

Theorem $T(n) = 2^n - 1$ for all $n \geq 0$

Proof Let n be an arbitrary integer $n \geq 0$.

Induction Hypothesis For all $k < n$, we have $T(k) = 2^k - 1$

Base Case $n = 0$. Then, $T(0) = 0 = 2^0 - 1$

General Case $n > 0$.
$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 \qquad \text{(Induction Hypothesis)} \\ &= 2^n - 2 + 1 \\ &= 2^n - 1 \end{aligned}$$ □

Integer Multiplication

Lattice based multiplication algorithm uses product of single digit integers to compute products of large integer

E.g.

$$\begin{array}{r} 1\;2\;3 \\ \times\; 4\;5\;6 \\ \hline 7\;3\;8 \\ 6\;1\;5 \\ 4\;9\;2 \\ \hline 5\;6\;0\;8\;8 \end{array}$$

We multiply each digit of one number by all of the digits of the other number, write down the partial products and add them up

To multiply two n-digit numbers, we need to write down n^2 digits
So, this takes $O(n^2)$ time – two nested for loops with no recursion

Algorithms for multiplying integers in $O(n^2)$ times have been known for millenia

Kolmogorov in 1953 formulated the "n^2" conjecture:

> any algorithm for multiplying two n-digit integers, needs at least n^2 steps

Kolmogorov organized a seminar to get mathematicians to work on the conjecture
Karatsuba was a graduate student who attended the first seminar and came up with a faster algorithm. The seminar was cancelled afterwards.

Karatsuba's Idea

Idea : Any number x with n digits can be written as a combination of two numbers with $n/2$ digits

$$\underbrace{x}_{n \text{ digits}} = a \cdot 10^{n/2} + b$$

Let $\underbrace{y}_{n \text{ digits}} = c \cdot 10^{n/2} + d$ be another number

Goal: Compute product of x and y

$$x \cdot y = ac \cdot 10^{n} + (a \cdot d + b \cdot c)\, 10^{n/2} + b \cdot d$$

We have reduced multiplication of one instance of two n-digit numbers to computing four products of $n/2$ digit numbers

We can let the recursion fairy compute these products and use the identity above to compute $x \cdot y$

Multiplying by powers of 10 corresponds to adding extra zeros which is easy to do

Addition is also easy to do.

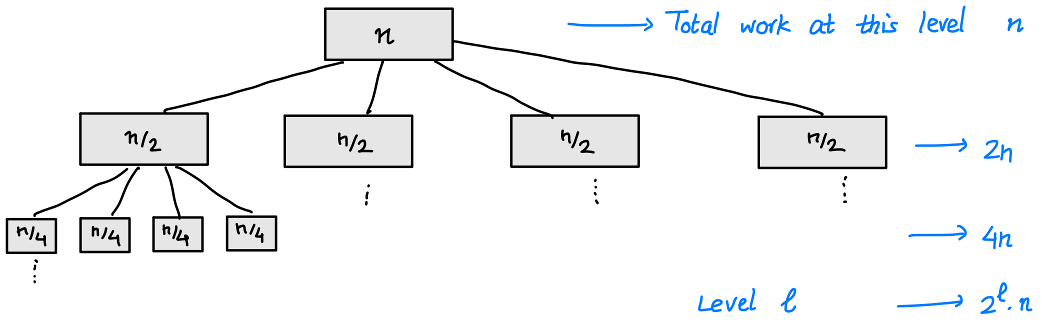
So, we get an algorithm whose running time $T(n)$ satisfies

$$T(n) = 4\,T\left(\frac{n}{2}\right) + O(n)$$

Time to add extra zeros and perform addition

How do we solve this recurrence ? We can guess and check, but to show another method we are going to use a recurrence tree.

We draw a tree for each recursive call and write how much non-recursive work is performed in that recursive call



Total amount of work $= \sum_{\ell=0}^{L} 2^{\ell} \cdot n = n \left(\sum_{\ell=0}^{L} 2^{\ell} \right)$ where L is the maximum level or maximum depth of recursion

This is a geometric series, so $\sum_{\ell=0}^{L} 2^{\ell} = O(2^L)$

What is L? Each time we recurse, we reduce the problem size by 2. After $L = \log_2 n$ times, we have 1-digit number, so we can multiply without recursion

Thus, $T(n) = O(n \cdot 2^{\log_2 n}) = O(n^2)$

This did not help! An intuitive way of seeing why this is n^2 is to pick one digit in each number, then there must be one bottom most recursive call where these two digits get multiplied. So, $O(n^2)$ is really the best we can hope for this algorithm, since every pair of digits gets multiplied.

Karatsuba's Algorithm was based on one more idea:

Recall $x \cdot y = ac \cdot 10^{n} + (a \cdot d + b \cdot c)\, 10^{n/2} + b \cdot d$

We computed by recursion $a \cdot c$, $a \cdot d$, $b \cdot c$, and $b \cdot d$
But what we want is $a \cdot c$, $a \cdot d + b \cdot c$, and $b \cdot d$

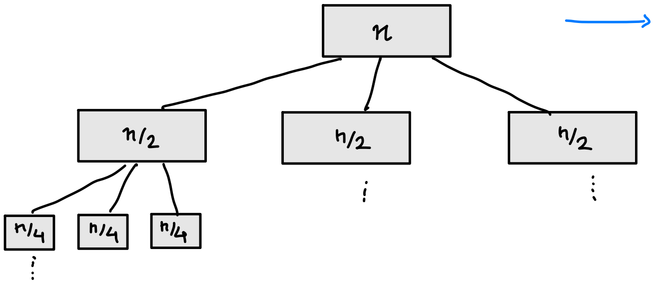
Karatsuba observed that
$$(a-b)(c-d) = ac - ad - bc + bd$$
$$\Rightarrow ad + bc = ac + bd - (a-b)(c-d)$$

Therefore, by computing three products $a \cdot c$, $b \cdot d$ and $(a-b) \cdot (c-d)$ recursively we can compute $x \cdot y$

Recursive calls might need to do some additional work now, e.g., subtraction but that is also easy to do

Overall, our recurrence now becomes $T(n) = 3\,T(n/2) + O(n)$

Redrawing the recursion tree, each node now has only 3 children and number of levels is still $\log_2 n$ since problem size goes down by half each time.



⟶ Total work at this level n

⟶ $\frac{3n}{2}$

⟶ $\frac{9n}{4}$

Level ℓ ⟶ $\left(\frac{3}{2}\right)^{\ell} n$

Thus, total amount of work $= \sum_{l=0}^{\log_2 n} \left(\frac{3}{2}\right)^{l} \cdot n = n \cdot O\left(\left(\frac{3}{2}\right)^{\log_2 n}\right)$

Using the identity $a^{\log_b c} = e^{\ln a \left(\frac{\ln c}{\ln b}\right)} = c^{\log_b a}$

Thus, $T'(n) = O(n \cdot n^{\log_2 (3/2)}) = O(n^{1 \cdot 6 \cdots})$ Significantly subquadratic!

We are not multiplying all pairs of digits anymore

In practice it is also faster than lattice-based multiplication, if $n \geqslant 50$

But one can improve it further by dividing number into more pieces and combining those with fewer steps

Eventually, this leads to $O(n \cdot \log n)$ algorithm from 2019 based on the Fast Fourier Transform which we are going to see next time

Next Lecture Fast Fourier Transform