ECS 20 – Fall 2021 – P. Rogaway Recursion

Thinking recursively

Closely tied to induction (at least it feels close to me!) is the ability to think recursively.

The key to thinking recursively is this: don't let your mind "descend" into the recursion; instead, think of smaller solutions as being solved as if by magic.

Example 1: Counting possible tic-tac-toe games

Let's count N_0 = total number of possible tic-tac-toe games, where X moves first. I'm going to assume I have a computer to help me out with the work.

It's good to first get a ballpark figure and use this to ascertain if our recursive decomposition is going to give a number, in a reasonable amount of time, when programmed up. For this we would note that $N < 9! =$ **362880** $\approx 2^{18.5}$. If you think back to the comments I made on what is practical, this is on the side of *easy*. Way less than 2^{30}.

Regard a tic-tac-toes position as encoded by a string w from $\{X,O,-\}^9$. That is, we regard a board as a 9-character string. Not all of the 3^9 possible strings can arise. But it's still a good way to represent a board.

Let $S(x)$ be the **set** of all next-possible-positions. Ex:

S(--- -X- ---) = {O-- -X- ---, -O- -X- ---, --O -X- ---, --- -OX- ---,
--- -XO ---, --- -X- O--, --- -X- -O-, --- -X- --O}.

On the other hand, S(OO- XXX ---) = $\emptyset$, as X has already won in this game. Note that either $S(w) = \emptyset$ or $|S(w)|$ is the number of dashes in the string w.

Let $N(w)$ = number of games that can continue from board w.
So we want to know $N_0 = N$(--- --- ---).

It is easy to compute N recursively. It's

$$N(w) = \begin{cases} 1 \text{ if } S(w) = \emptyset \\ \displaystyle\text{Sum}_{y \in S(w)} \; N(y) \end{cases}$$

For example, *N*(--- --- ---)=*N*(X-- --- ---)+*N*(-X- --- ---)+ ... +*N*(--- --- --X).

We can code it up!! The answer is N_0= **255,168** possible games. Took me 22 lines of code:

```
# Counts the number of possible tic-tac-toe games
# Uses recursively defined N(w) = number of continuations of board w.
# The board is represented as a 10-element list of 'X', 'O', '-' chars, element 0 unused.

X,O,EMPTY,UNUSED= 'X','O','-','.'
Wins = [[1,2,3],[4,5,6],[7,8,9], [1,4,7],[2,5,8],[3,6,9], [1,5,9],[3,5,7]]

def win(w,P):              # return true if player P has won. P==X or P==O
    for [a,b,c] in Wins:
        if w[a] == w[b] == w[c] == P: return True
    return False

def whose_move(w):         # Return X or O depending on whose move it is
    if w.count(EMPTY)%2: return X
    return O

def game_over(w):          # True after a win or nowhere left to go
    return win(w,X) or win(w,O) or w.count(EMPTY)==0

def N(w):                  # Calculate number of games that elaborate board
    if game_over(w): return 1
    sum = 0
    for i in range(1,10):
        if w[i]!=EMPTY: continue
        y = w[:]
        y[i] = whose_move(w)
        sum += N(y)
    return sum

w = [UNUSED] + [EMPTY]*9   ###  MAIN PROGRAM  ###
print('Number of possible games is', N(w))
```

Example 2: The Towers of Hanoi

```
          -|-              |               |
         --|--             |               |
        ---|---            |               |
       ----|----           |               |
===========|===============|===============|==========
           A               B               C
```

n rings of increasing diameter are placed on peg A. The pegs must be moved from A to C in a way that respects the following rules:

- Only the topmost ring on a peg can be moved.
- A bigger ring cannot be placed atop a smaller one.

Problem: Find a function that describes the least number of moves needed to solve the problem when you have *n* rings.

How many moves do you think it takes to move the four rings? One student guessed, knew, or figured out the correct answer if 15. Where does this come from?!

We want a formula that specifies a number of moves that is both:

- **Sufficient**: there **is** a solution to the game using this number of moves.
- **Necessary**: no solution can use **fewer** moves than this.

Sufficiency, and the value of generalizing: First, let's define what we're interested in. We could write: let

Let T_n = The minimum number of moves needed to move the *n* rings from peg A to peg C (obeying the rules of the game).

But it's useful to be more general:

Let T_n = The minimum number of moves needed to move *n* rings from some one specified peg to some other specified peg (obeying the rules of the game).

```
// Transfer n pegs from A to C using B as intermediate
algorithm TH (n, A,B, C) // Alternative convention A,B,C ?
if n = 1 then Move(A,C)
TH(n-1, A,B, C)
Move(A,C)
TH(n-1, B,C, A)
```

Think recursively. Assume a "black box" algorithm can move the first *n* - 1 rings from any peg to any other peg. Solving the problem this way requires that the first *n* - 1 rings be moved, then the largest ring be moved once, then the smaller rings be moved on to the largest ring. This number of moves can be represented by:

$$T_n \leq T_{n-1} + 1 + T_{n-1} = 2T_{n-1} + 1$$

Necessity. Now we have to reason about *any* algorithm that solves the puzzle. Any solution must move the largest ring to the final peg for the very last time. That takes one move. But before that happened, we had to get the n-1 rings that were formerly on top of the start peg and move them off to a free peg. That takes at least T_{n-1} moves. After we got the biggest ring to its destination peg, we had to move the n-1 smaller rings from the free peg where they were at to the final peg. That takes at least T_{n-1} moves. So, all in all, any solution needs to spend at least

$$T_n \geq 1 + T_{n-1} + T_{n-1} = 2T_{n-1} + 1$$

moves.

Solution: $T_n = 2^n - 1$

To get this: first, make a table of values from the recursion relations. From the table one can make a natural guess. Then we can prove that by induction. Go through all steps.

Example 3: Karatsuba multiplication (1960/1962)

Suppose we want to multiply two decimal numbers (binary numbers would work the same way). We write one number as $x = x_1 \,||\, x_0$ and the other was $y = y_1 \,||\, y_0$, each half having m digits (let's not worry about what to do if m is odd; no significant complications are added). So

$x = x_1\, 10^m + x_0$
$y = y_1\, 10^m + y_0$,

The product is then
$xy = (x_1 10^m + x_0)(y_1 10^m + y_0)$
$\quad = z_2 10^{2m} + z_1 10^m + z_0$
where
$z_2 = x_1 y_1$
$z_1 = x_1 y_0 + x_0 y_1$
$z_0 = x_0 y_0$.

These formulas require **four multiplications**. Thus one way to multiply decomposes our size-n problem into four problems of size $n/2$, plus some added overhead that is $O(n)$.

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

We will see shortly that decomposition does no better, asymptotically, than grade-school multiplication.

Karatsuba observed that xy can be computed in only **three multiplications** of m-digit values: With z_0 and z_2 as before we can calculate z_1 by way of
$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$ $= (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0 = x_1y_0 + x_0y_1$

Example Let's compute

```
     98  76
  *  56  78
  ---------
       5928
     7644
     4256  These two sum to 11900, which we can also get as
   5488     11900 = (98+76)(56+78) – 5928 - 5488
  ---------       =        174*134 – 5928 - 5488
   56075928       =          23316 – 5928 - 5488
                  = 11900
```

Now let's see if this trick help:

First, the 4-multiply method:

$T(n) = 4T(n/2) + n$ (when $n > 1$; $T(n) =$ const when $n = 1$)
$= 4(4T(n/4) + n/2) + n$
$= 4^2\ T(n/4) + 2n + n$
$= 4^3\ T(n/8) + n(1 + 2 + 4)$
$= 4^4\ T(n/2^4) + n(1 + 2 + 2^2 + 2^3)$
$= \ldots$
$= 4^k + n(2^k - 1)$
$\in \theta\ (n^2) + \theta\ (n^2)$
$\in \theta\ (n^2)$

Now, the 3-multiply method:

$T(n) = 3\,T(n/2) + n$

$= 3\,(3T(n/4) + n/2) + n$

$= 3^2\,T(n/4) + (3n/2 + n)$

$= 3^2(3T(n/8) + n/4) + 3n/2 + n$

$= 3^3\,T(n/8) + 3^2 n/2^2 + 3n/2 + n$

$= 3^3\,T(n/8) + n(1 + 3/2 + (3/2)^2))$

$= 3^4\,T(n/16) + n\,(1 + (3/2) + (3/2)\text{^}2 + (3/2)\text{^}3)$

$= \ldots$

$= 3^k\,T(n/2^k) + n\,(1 + (3/2) + (3/2)^2 + (3/2)^3 + \ldots + (3/2)^{k-1})$

At this point it would be good to know what is

$S = 1 + p + p^2 + \ldots + p^{k-1} + p^k$

$S\,p = p + p^2 + \ldots + p^k + p^{k+1}$

$S\,p + 1 = 1 + p + p^2 + \ldots + p^k + p^{k+1}$

$S\,p + 1 = S + p^{k+1}$

$S(p - 1) = p^{k+1} - 1$

$S = (p^{k+1} - 1) \;/\; (p - 1)$

It is worth remembering this result (or be able to re-derive it if you need it).

$1 + p + p^2 + \ldots + p^{k-1} = (p^k - 1) \;/\; (p - 1)$

So, with $p = 3/2$, we have

$1 + (3/2) + (3/2)^2 + (3/2)^3 + \ldots + (3/2)^{k-1} = 2\,(3/2 - 1)^k$

$3^k\,T(n/2^k) + n\,(1 + (3/2) + (3/2)^2 + (3/2)^3 + \ldots + (3/2)^{k-1})$

$= 3^k\,T(n/2^k) \quad + \quad 2n\,((3/2)^k - 1)$

Now we want $k = \lg n$, so

$= 3^{\lg n} \quad + 2n\,(3/2)^{\lg n}$

$= (2^{\lg 3})^{\lg n} + 2n\;3^{\lg n} / 2^{\lg n}$

$= n^{\lg 3} + 2\,n^{\lg 3}$

$\in \theta(n^{\lg 3})$

$\subseteq O(n^{1.585})$

Best-known algorithm for this problem: we can actually multiply two *n*-digit numbers in time O (*n* log *n*). This is due to Harvey and van der Hoeven (2019). It follows a steady improvement in running times that begin with Karatsubu, continues with a famous result of Schönhage–Strassen (1971) that takes O(n log n log log n).

The O(n log n) multiplication algorithm is https://tinyurl.com/3zhk24xe

Example 4: Binary Search

No doubt many of you have encountered this algorithm before. The algorithm is meant to determine if an element x is in a n-element sorted listed of ordered elements. We first compare x with the middle element p (when n is even you'll need to be just to the left or just to the right of the non-existent middle, but let's ignore that detail.) If $x=p$ you answer Yes, that it's in the list. If $x<p$ then you need to recurse on the left half L. Otherwise you need to recurse on the right half R.

Write out the algorithm …

Now to analyze its running time:

$T(n) = T(n/2) + 1$

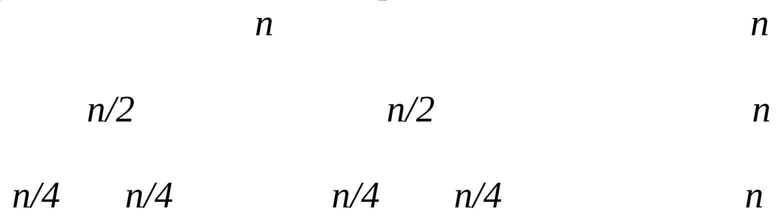
We can use repeated substitution, as before, to get that $T(n) \in \theta$ (lg n).

Example 5: Mergesort

```
**algorithm** Sort($A$)
$n \leftarrow |A|$
**if** $n = 1$ **then return** A
Left $\leftarrow$ Sort ($A[1 .. \lfloor n/2 \rfloor]$)
Right $\leftarrow$ Sort ($A[\lfloor n/2 \rfloor + 1 .. n$)
**return** Merge(Left, Right)
```

Analysis: Let $T(n)$ be the **worst-case** number of comparisons to sort n items using the algorithm above. Then $T(n) = 2\ T(n/2) + n\text{-}1$. Show how to bound it by replacing the n-1 with n and then using repeated substitution, as before, to get $T(n) = n \lg n$.

An alternative is to think of a "recursion tree". Each level of the tree records the size that the recursive calls are invoked on. Compute a partial sum for each level of the tree that captures all additive overhead. Sum up all those partial sums to get the total. In this example:



The tree will have lg n levels, each full, so the sum of the numbers on the right will be n lg n, and the total time for our algorithm will be $T(n) = n \lg n$.

Example 6: Partition numbers

How many ways $p(n)$ can we write n as the sum of positive numbers? Order of addends doesn't matter.

1 = 1	So P(1)=1
2 = 2, 1+1	So P(2)=2
3 = 3, 2+1, 1+1	So P(3)=3
4 = 4, 3+1, 2+2, 2+1+1, 1+1+1+1	So P(4)=5
5 = 5,4+1,3+2,3+1+1,2+2+1,2+1+1+1,1+1+1+1+1	So P(5)=7

$p(n, m)$ = number of ways to partition n where the maximum number used among the addends is m

$$p(n) = p(n, 1) + p(n, 2) + \dots + p(n, n)$$

$$p(n, m) = p(n-m, m) + p(n-m, m-1) + \dots + p(n-m, 1)$$

$p(n,1) = 1$ $\quad$ $p(n, n)=1$

$p(n, m) = 0$ if $n<m$

$p(n, m) = 1$ if $n=m$

	m=1	2	3	4	5	6	7	8	9	p(n)
n=1	1	0	0	0	0	0	0	0	0	1
n=2	1	1	0	0	0	0	0	0	0	2
n=3	1	1	1	0	0	0	0	0	0	3
n=4	1	2	1	1	0	0	0	0	0	5
n=5	1	2	2	1	1	0	0	0	0	7
n=6	1	3	3	2	1	1	0	0	0	11
n=7	1	3	4	3	2	1	1	0	0	15
n=8	1	4	5	5	3	2	1	1	0	22
n=9	1	4	7	6	5	3	2	1	1	30

More practice with recursion trees

Let's try the recurrence relation

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

This would correspond to a recursive decomposition of a problem that we can solve by breaking the problem into a half-size chunk and a third-size chunk and combining those results, and the original problem instance, in O(n^2) time, to solve our initial problem.

Well, we could use repeated-substitution, but I think that a recursion tree will be more easier. The "top" of the tree would look like this:

$$n \qquad\qquad n^2$$

$$n/2 \qquad\qquad n/3 \qquad\qquad n^2\,(1/2+1/3) = n^2(5/6)$$

$$n/4 \quad n/6 \qquad\qquad n/6 \quad n/9 \qquad n^2\,(1/4+1/3+1/9) = n^2(25/36)$$

….

We want to add up all the red numbers on the right. But it's not so simple what happens to them. Long root-to-leaf in the tree will have different lengths than short root-to-leaf paths—the tree is not "full". And while its seems clear that the sum of the constants multiplying n^2 will converge to some constant, it's might not clear what constant it converges to. (Although that is not hard. For $0<p<1$, the infinite series $1 + p + p^2 + \dots$ is easily show to be $1/(1-p)$, where $p=5/6$, so even if the tree were infinite the sum of the red values would only be $6n^2$. There will be $\theta\,(n)$ leaves on our tree, so their contribution to the sum will be smaller than the (red) additive overhead. We conclude that $T(n)$ is O(n^2), and since it is clearly $\Omega(n^2)$ too (we get that from even the first recursive call), we can conclude that $T(n) \in \theta\,(n^2)$,

Addendum: Some material not covered

Example 7: Cake cutting

See http://www.cs.berkeley.edu/~daw/teaching/cs70-s08/notes/n8.pdf for a nice writeup

1. If $n = 2$, use the cut-and-choose protocol. Otherwise:
2. The first $n-1$ participants divide the cake by recursively invoking this procedure.
3. For i = 1,2,...,$n-1$, do:
   a) Participant *i* divides her share into *n* pieces she considers of equal worth (by her measure).
   b) Participant *n* collects whichever of those *n* pieces he considers to be worth most (by his measure).

Number of cuts:

$T(n) = T(n-1) + (n-1)^2$

1	2	3	4	5	6
0	1	5	14	30	55

$T(n) = T(n-1) + (n-1)^2$
$= T(n-2) + (n-1)^2 + (n-2)^2$
$= T(n-3) + (n-1)^2 + (n-2)^2 + (n-3)^2$
$= T(n-3) + (n-1)^2 + (n-2)^2 + (n-3)^2$
$= 1 + 2^2 + 3^2 + 4^2 + \dots + (n-1)^2$
approx. Integeral$_1^n$ x^2 approx $n^3/3$

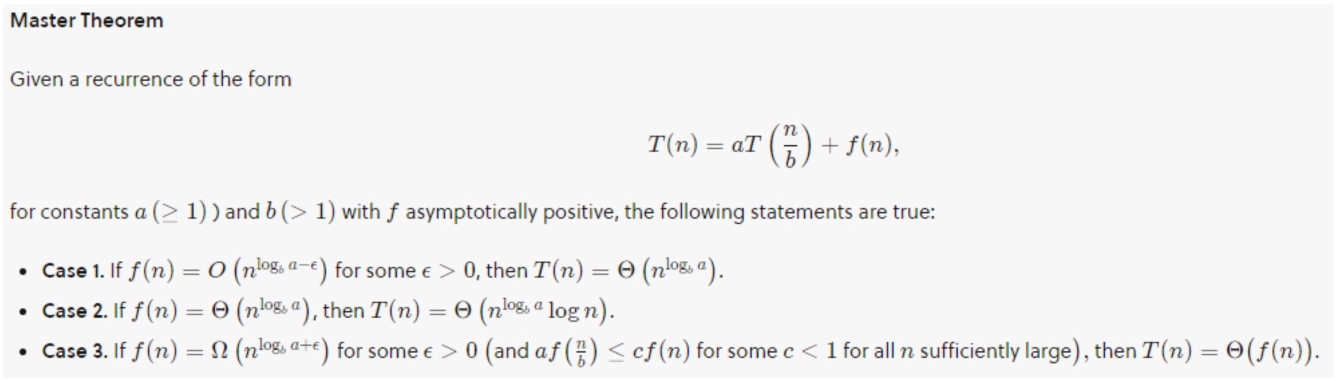
$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Prove by induction.

Not covered in class:

"Master Theorem" for many recurrences

https://brilliant.org/wiki/master-theorem/



Master Theorem

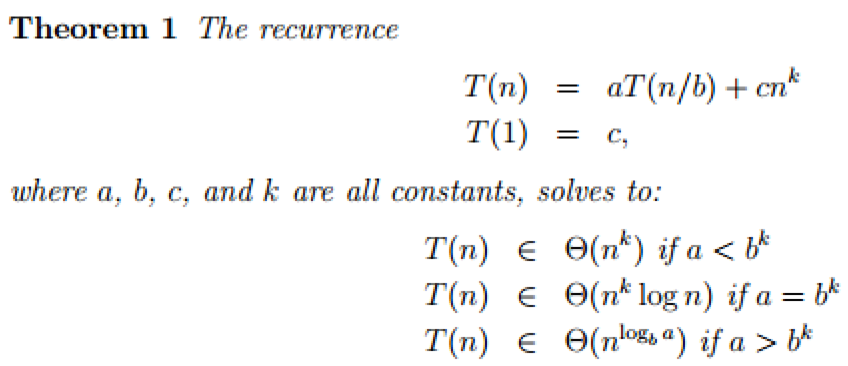
Given a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

for constants $a\,(\geq 1)$) and $b\,(> 1)$ with f asymptotically positive, the following statements are true:

- **Case 1.** If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
- **Case 2.** If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
- **Case 3.** If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$ $\left(\text{and } af\left(\frac{n}{b}\right) \leq cf(n) \text{ for some } c < 1 \text{ for all } n \text{ sufficiently large}\right)$, then $T(n) = \Theta\big(f(n)\big)$.

https://randerson112358.medium.com/master-theorem-909f52d4364



Theorem 1 *The recurrence*

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \text{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}$$