Lecture 2: Recursion

Last updated: Aug 24, 2024

References:

- Algorithms, Jeff Erickson, Chapter 1

The most important technique used in designing algorithms:

Reduction

Reducing a problem X to another problem Y means that using an algorithm for Y as a blackbox or subroutine for problem X.

It's important to note that the correctness of algorithm for problem X cannot depend on *how* the algorithm for problem Y works.

Example: the peasant multiplication algorithm reduces the multiplication problem to three simpler problems: addition, halving, and parity checking (which we know how to solve).

Recursion

Recursion is a particularly powerful kind of reduction:

- If the given instance of the problem can be solved directly (e.g. it's really small), solve it directly;
- Otherwise, reduce it to one or more **simpler instances of the same problem**.

The most important thing to note about recursion is that, we can solve the simpler instances, by calling the algorithm itself!

The trick is to not go into the details of the solution of the sub-problem; rather, consider it somebody else's problem and it's solved.

Stop thinking about the solution of the subproblem!

Example: Peasant Multiply


```
def PeasantMultiply(x,y): # x,y>=0, returns x*y
    if x == 0:
        return 0
    t = PeasantMultiply(x//2,y+y)
    if x%2 == 0:
        return t
    if x%2 == 1:
        return y+t
```

Proof: [prove the claim in the function comment].

By (math) induction on the input x (its integer value, or rather its number of digits/bits). The claim is obviously correct for x==0 (or that can be represented by 1 bit).

Let x has n bits. (**induction hypothesis)** Assume the claim works for all x that has less than n bits. Now let's prove that the claim is correct for any x,y with x having n bits. In the function we are invoking `t=PeasantMultiply(x//2,y+y)`. By the induction hypothesis (because x//2 will have less than n bits), $t = \lfloor x/2 \rfloor (2y)$.

It's clear that

$$xy = \begin{cases} \lfloor x/2 \rfloor (2y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor (2y) + y & \text{if } x \text{ is odd} \end{cases}$$

The x odd case is because $\lfloor x/2 \rfloor (2y) = (x/2 - 0.5)(2y) = xy - y$.

By induction, the claim holds for all positive integer pairs of x,y.

Example: Tower of Hanoi

- Objective: move 64 disks from 1st peg to 3rd peg
- Rule: bigger disk must always be below smaller ones

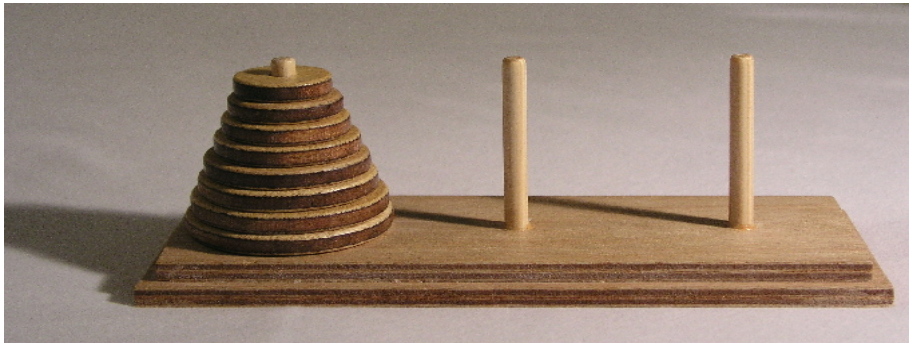


Figure 1. Tower of Hanoi. Image source: Wikipedia

First step: generalize the problem size from 64 to n!

More general problem might be easier to solve than an instance!

Example: Tower of Hanoi, #2

Rephrased problem:

Move n disks from one peg to another, using a 3rd peg as occasional placeholder, without placing bigger disk on top of smaller ones.

The secret to solve this problem is

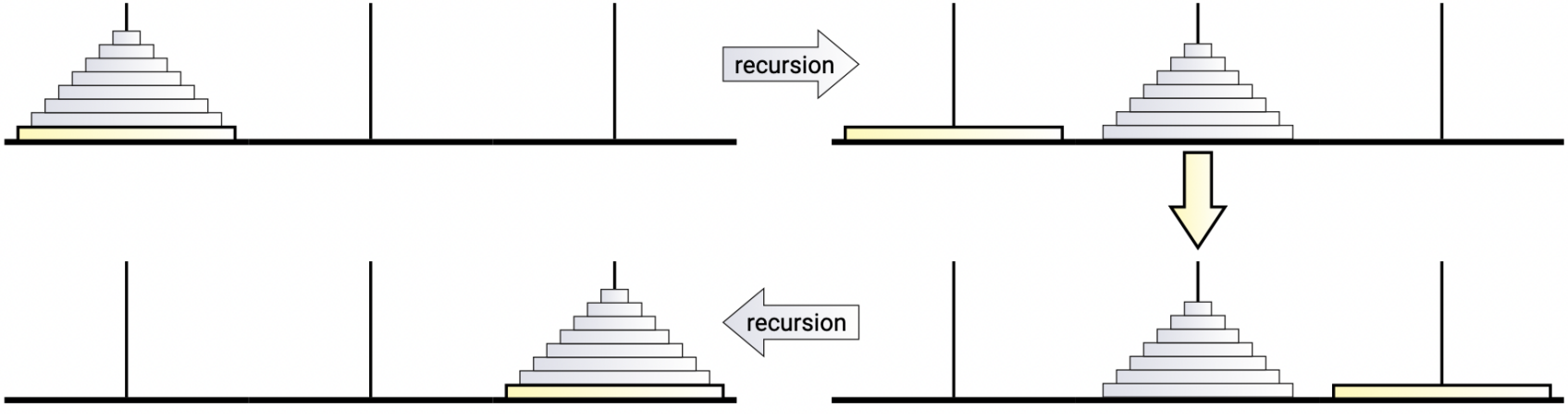
to reduce the problem size, rather than solve it at once!

OK, so now instead of moving a whole n disks, how about we just move one (say, the biggest one, because everything can be put on top the biggest one).

Example: Tower of Hanoi, #3

Recursive solution steps: (suppose we can solve size $n-1$ problem)

- Move the top $n-1$ disks to another peg (recurse)
- move the biggest disk to 3rd peg
- move the $n-1$ disks to the 3rd peg (recurse)



Example: Tower of Hanoi, #4

Now, if only we know how to solve the size $n-1$ problem...

STOP! Don't go into the subproblem solution.

It's somebody else's problem (maybe yours, but not now).

The only missing part is the base case, which is trivial: when there is only one disk, we surely know how to move 1 disk from one peg to another, without violating the rule...

In fact, we can reduce the base case even further: moving 0 disk. We do nothing, which is the correct thing to do.

Oftentimes, using a ridiculously simple base case leads to the most simple & elegant algorithm, with least edges cases to handle.

Example: Tower of Hanoi, #5

The formal algorithm:

```
# move n disks from src peg to dst peg using tmp peg as temporary
holder. no violation of rule that smaller disk must always be on top
of bigger disk. Print out the movement of each disk.
def hanoi(n, src, dst, tmp):
    if n == 0: # base case
        return
    if n > 0: # recursive case
        hanoi(n-1, src, tmp, dst)
        print(f"Move␣disk␣{n}␣from␣peg␣{src}␣to␣peg␣{dst}")
        hanoi(n-1, tmp, dst, src)
```

Proof of correctness of the claim [what exactly is the claim?] by induction. When n=0 the function correctly does nothing.

Suppose the claim is true for all k disks with $k < n$ (induction hypothesis). We now prove the claim is also true for n disks. There are three steps in the function. First step is to move (the first) $n - 1$ disks from `src` peg to `tmp` peg using `dst` peg as temporary holder. This move can be done because 1) induction hypothesis 2) biggest disk (disk n) is not moved and it's always at the bottom, no rule violation. Second step: move disk n to `dst` peg (legal move?) Third step: similar to first step; all legal.

Therefore by mathematical induction, the recursive function correctly prints out movements that move n disks with all legal moves.

Commentary: Is the design of recursive algorithm that different from the induction proof of it?

Time complexity

$$T(n) = 2T(n-1) + 1, T(0) = 0 \quad \Rightarrow \quad T(n) = 2^n - 1$$

$$T(64) \approx 18.5 \times 10^{18}$$

Hanoi Tower: Real Code

To solve the original "move 64 disks from 1st peg to 3rd peg", call

```
#def hanoi(n, src, dst, tmp):
hanoi(64,0,2,1) # takes long long long time
```

Mergesort

Let's review mergesort algorithm. It's recursive:

1. Divide the array into two subarrays of equal size
2. Recursively mergesort the two subarrays
3. Merge the two sorted subarrays.

```
def mergesort(A): # return the sorted A
    if len(A)==0:
        return []
    n = len(A)
    X = mergesort(A[:n])
    Y = mergesort(A[n:])
    B = [0]*n
    merge(X,Y,B)
    return B
```

The first step is simple. The second step is just two recursive calls. The third step is non-trivial.

The correctness of the mergesort algorithm depends on the correct merge (3rd step).

Bisection: Square Root

Problem: Given a positive real number $x > 1$, compute its square root $\sqrt{x}$ within error bound of 0.001. (i.e. your computed result $|\text{sqrt}(x) - \sqrt{x}| < 0.001$). The elementary instruction is double precision floating point IEEE754 (almost certainly your everyday computer/phone).

Put in other words, we seek a floating point number a such that $f(a) = a^2 - x = 0$, namely the root of the function $f(a)$.

We know that $f(a)$ is monotonic increasing function.

We know $f(0) = -x < 0$, and $f(x) = x^2 - x > 0$. Therefore the root of $f(a)$ must be in the interval $(0, x)$, but which one number inside the interval is the root?

Think divide and conquer–can we reduce the "search space"?

The merge() algorithm

The merge algorithm takes two **sorted** arrays, and merge them into a single **sorted** array. We can recurse! Remember in designing recursion, we try to reduce problem, rather than solving it directly.

In merge, we consider the last element of result C.

```python
# given two sorted array A,B, merge them into a single sorted array C
def merge(A, B, C):
    m = len(A)
    n = len(B)

    # Base case: If either A or B is empty, do the obvious thing
    if m == 0:
        C[:n] = B
        return
    elif n == 0:
        C[:m] = A
        return

    # Recursive case: where would last element of C (i.e. C[m+n-1])
    # come from? Either last element of A (A[m-1]) or of B (B[n-1]).
    if A[m-1] > B[n-1]:
        C[m+n-1] = A[m-1]
        merge(A[:m-1], B, C[:m+n-1])
    else:
        C[m+n-1] = B[n-1]
        merge(A, B[:n-1], C[:m+n-1])
```

How to formally prove that the `merge(A,B,C)` algorithm is correct?

Note that the **algorithm looks like a induction proof**!

Base case in merge(A,B,C) => base cases in induction proof

Recursive calls `merge(A[:m-1], B, C[:m+n-1])`, and `merge(A, B[:n-1], C[:m+n-1])` => invoking **induction hypothesis** (assuming for all A,B,C with `size(C) < m+n` the `merge(A,B,C)` works).

The main argument in induction proof is in the branch: set `C[m+n-1]` to either `A[m-1]` or `B[n-1]` depending which one is bigger: if `C[:m+n-1]` is sorted, then `C[:m+n]` must be sorted as well.

Recursive algorithm is no different than an induction proof!

Writing the Proof of Recursive Algorithm

Note that we have seen several example proofs of recursive algorithm/function.

The proof is mathematical induction with almost the same boilerplate: setting up induction hypothesis, argue that recursive step is correct by invoking the induction hypothesis, argue base case work, and that's about it.

Now we are familiar with it, the proof of recursive algorithm/function can be done without the boilerplate, by arguing only the essential part–why the recursive case is correct? Put in other words, how to solve the big problem with the help of **solutions** of the smaller problem?

In this course when you have a recursive algorithm as your solution, your proof can consists of the Pythonic pseudocode, the # com-

ment on the claim of the algorithm/function (what exactly does the function accomplish?), and the argument of solving current problem with help of solutions of small problems. No need to spell out the whole induction proof.

Isn't that nice? Recursive algorithms are very easy to prove! (And design and proof are the same thing).

Mergesort() time complexity

The time complexity of merge() is

$$S(m+n) = 1 + S(m+n-1), \quad S(0) = 0 \tag{1}$$

We solve it: $S(n) = n$ (how? The best approach is to take a guess, and prove it by induction!)

The time complexity of mergesort() is

$$T(n) = \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)}_{\text{recursion on two subarrays}} + \underbrace{n}_{\text{merge()}} \tag{2}$$

How to solve? Again, we can take a guess of $T(n) = n \log n$ and prove it by induction. But how do we guess? By looking at recursion tree. (later)

Design Pattern

Divide and Conquer:

1. **Divide** the problem into several *indepenent smaller* instances of the *same problem*.
2. **Delegate** each smaller instances to the recursion fairy (the blackbox solver, subroutine)
3. **Combine** the solutions for the smaller instances into the solution for the given instance.

If the problem is of sufficient trivial size, we directly solve by brute force, in constant time.

Proof of the correctness of D&C recursion is based on induction.

Analysis of the complexity is based on recurrence equation.

Recursion Tree: Solving Reccurence

How to solve recurrence equation like this:

$$T(n) = r\,T(n/c) + f(n) \tag{3}$$

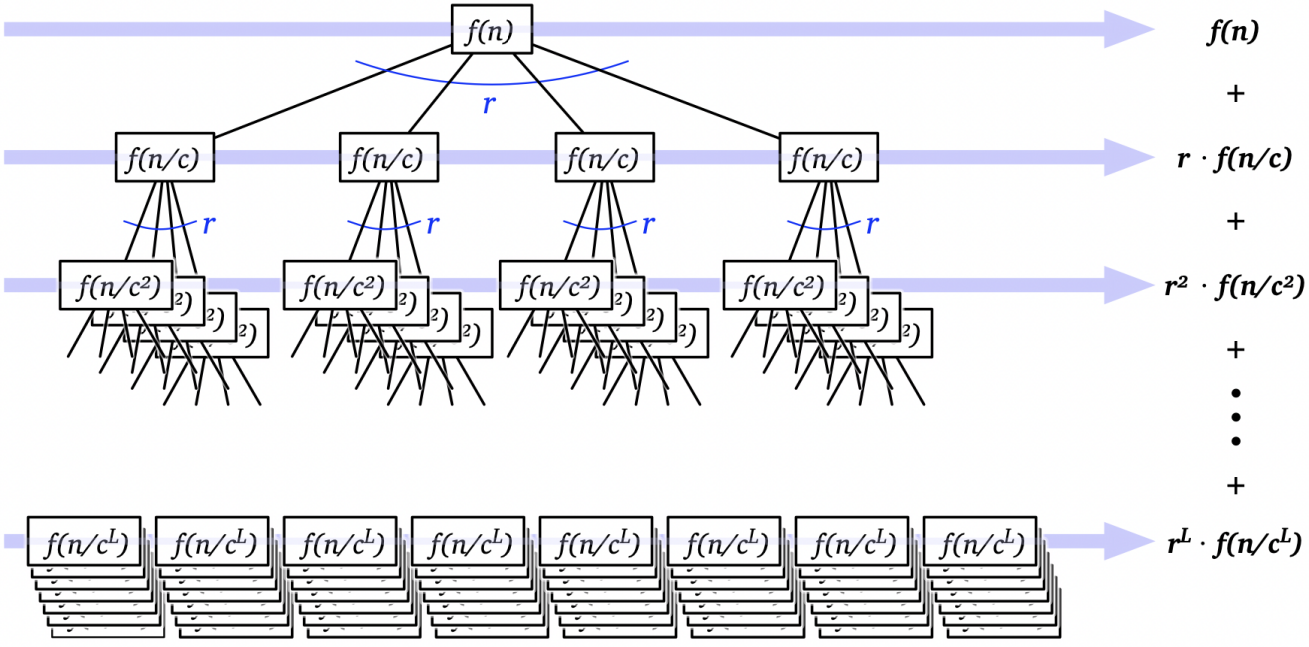


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

Recursion Tree

Now the cost of the whole tree is the summation of all the levels:

$$T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$$

$L = \log_c n$ is the number of levels of the tree. We can assume $T(1) = 1$. How many leaves in the tree? $r^L = r^{\log_c n} = n^{\log_c r}$.

Three cases of level-by-level series ($\sum$).

1. Decreasing: if the series decays exponentially, then $T(n) = \Theta(f(n))$
2. Equal: we have $T(n) = O(L\, f(n)) = \Theta(f(n) \log n)$

3. Increasing: if the series grows exponentially, then $T(n) = \Theta(n^{\log_c r})$

This level-by-level analysis works not only for the regular recurrence form:

$$T(n) = rT(n/c) + f(n)$$

It also works (with some adaptions) for irregular ones such as:

$$T(n) = T(n/a) + T(n/b) + f(n)$$

We can expand the recursion tree and observe the level-by-level series:

- **Decreasing exponentially**: cost dominated by first level; $T(n) = \Theta(f(n))$
- **Equal**: $T(n) = \Theta(f(n) \log n)$

- **Increasing exponentially**: (different from regular case!) We know that $T(n) = n^{\alpha}$, we need to determine α. How? Substitute it back to recurrence and solve for α.

Example: Recurrence

$$T(n) = T(3n/4) + T(2n/3) + n^2 \tag{4}$$

The level-by-level series are:

$n^2, 145n^2/144, (145n)^2/144^2, \ldots$

This is an exponentially increasing (geometric) series, with ratio 145/144. The third case (**increasing**) applies. Suppose: $T(n) = n^{\alpha}$ Substitute it back to the recurrence (4) gives us equation:

$$n^{\alpha} = (3n/4)^{\alpha} + (2n/3)^{\alpha} + n^2$$

We must have $\alpha > 2$ (why? look at the series). Dividing both sides by n^α and taking $n \to \infty$, we have equation:

$$1 = (3/4)^\alpha + (2/3)^\alpha$$

This solves to $\alpha \approx 2.0203$, which is a root to the previous equation.

(You can solve it via wolframalpha: `http://bit.ly/39P3D7i`)

So the solution is:

$$T(n) = \Theta(n^{2.0203})$$

Reference on solving recurrences: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf

The Master Theorem

The recurrence $T(n) = aT(n/b) + f(n)$ can be solved as follows.

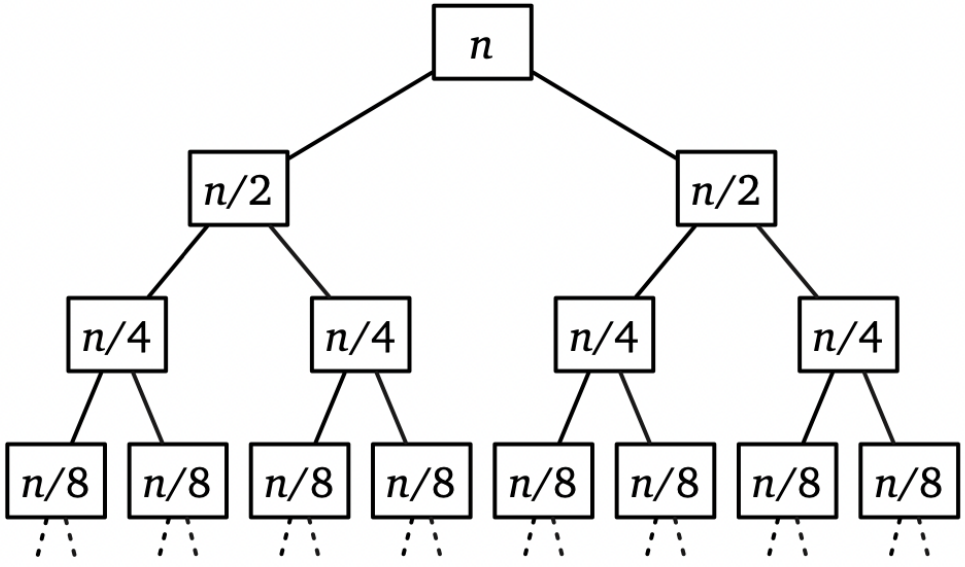
- If $af(n/b) = \kappa f(n)$ for some constant $\kappa < 1$, then $T(n) = \Theta(f(n))$.
- If $af(n/b) = Kf(n)$ for some constant $K > 1$, then $T(n) = \Theta(n^{\log_b a})$.
- If $af(n/b) = f(n)$, then $T(n) = \Theta(f(n) \log_b n)$.
- If None of these three cases apply, you are on your own

Proof. Recursion tree? □

Examples:

- Mergesort: $T(n) = 2T(n/2) + n$
- Randomized selection: $T(n) = T(3n/4) + n$
- SplitMultiplication: $T(n) = 4T(n/2) + n$
- FastMultiplication: $T(n) = 3T(n/2) + n$
- Randomized quicksort: $T(n) = T(3n/4) + T(n/4) + n$
- Deterministic selection: $T(n) = T(n/5) + T(7n/10) + n$

Mergesort recursion tree



In mergesort(), $f(n) = n$, $c = r = 2$, the series is equal in every level, so the second case:

$$T(n) = O(f(n) \log n) = O(n \log n)$$

Exercises.

1. $T(n) = 2T(n/3) + 1$
2. $T(n) = T(n/3) + T(2n/3) + 1$
3. $T(n) = 3T(n-1) + 2$

Fast Multiplication

We have seen two algorithms for multiplying two n-digits number in $O(n^2)$ time: grade-school lattice algorithm, and Egyption peasant algorithm.

Now let's think of a divide and conquer way of solving multiplication.

Can we get a bit more efficient algorithm by splitting the digit arrays into half, and exploit the following identity:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

```python
# given two n digits integer x, y, return x*y
def split_multiply(x, y, n):
    # Base case: if n is 1, return the product of x and y
    if n == 1:
        return x * y # single digit multiplication

    # Calculate m as the ceiling of n/2
    m = ceil(n / 2)

    # Split x into two parts: a and b
    a = x // (10 ** m)
    b = x % (10 ** m)

    # Split y into two parts: c and d
    c = y // (10 ** m)
    d = y % (10 ** m)

    # Recursively calculate e, f, g, and h
    e = split_multiply(a, c, m)
    f = split_multiply(b, d, m)
    g = split_multiply(b, c, m)
    h = split_multiply(a, d, m)

    # Combine the results and return the final product
    return (10 ** (2 * m)) * e + (10 ** m) * (g + h) + f
```

SplitMultiply analysis

Correctness is easy to show using induction. The run time is

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

This results in an increasing geometric series, which implies:

$$T(n) = O(n^{\log_2 4}) = O(n^2)$$

This did not improve on the efficiency of previous two algorithms. The culprit is the 4 subproblems (multiplication). If we can reduce...

$$ac + bd - (a-b)(c-d) = bc + ad$$

that to 3?

FastMultiply


```
def fast_multiply(x, y, n):
    # Base case: if n is 1, return the product of x and y
    if n == 1:
        return x * y

    # Calculate m as the ceiling of n/2
    m = ceil(n / 2)

    # Split x into two parts: a and b
    a = x // (10 ** m)
    b = x % (10 ** m)

    # Split y into two parts: c and d
    c = y // (10 ** m)
    d = y % (10 ** m)

    # Recursively calculate e, f, and g
    e = fast_multiply(a, c, m)
    f = fast_multiply(b, d, m)
    g = fast_multiply(a - b, c - d, m)

    # Combine the results and return the final product
    return (10 ** (2 * m)) * e + (10 ** m) * (e + f - g) + f
```

OK, now the time complexity is

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

which is still increasing series, and gives us: $T(n) = O(n^{\log_2 3}) \approx = O(n^{1.58496})$, a significant improvement!

More Hanoi Tower Variants

Variant of Hanoi Tower: move n disks from peg 0 to peg 2, with the restriction that you cannot move disk between peg 1 and 2; every move must come from or to peg 0.

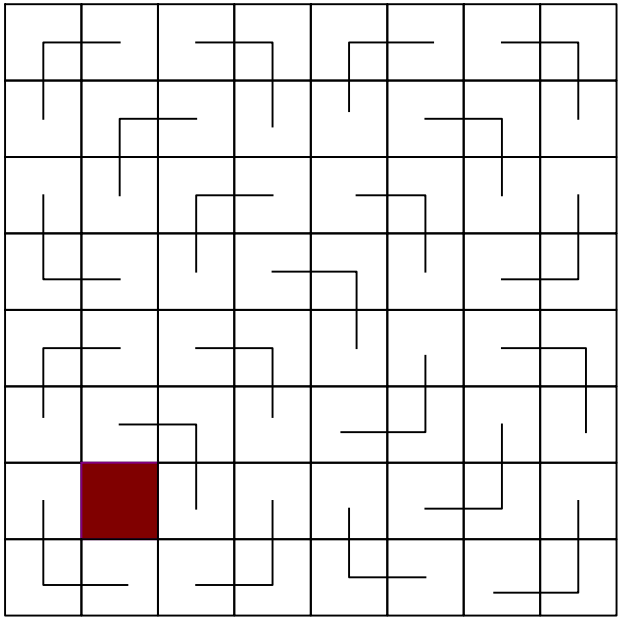
```
Hanoi1(n,src,dst,tmp):                          #peg 0->1, 0->2
    Hanoi1(n-1,src,tmp,dst)
    move disk n to dst
    Hanoi2(n-1,tmp,src,tmp)
    Hanoi1(n-1,src,dst,tmp)

Hanoi2(n,src,dst,tmp):                          #peg 1->0, 2->0
    Hanoi2(n-1,src,dst,tmp)
    Hanoi1(n-1,dst,tmp,src)
    move disk n to dst
    Hanoi2(n-1,tmp,dst,src)
```

Recurrence? Time complexity?

Example: Tiling Problem

Ex 26. Suppose you are given a $2^n \times 2^n$ checkerboard with one (arbitrarily chosen) square removed. Describe and analyze an algorithm to compute a tiling of the board by without gaps or overlaps by L-shaped tiles, each composed of 3 squares. Your input is the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time. [Hint: First prove that such a tiling always exists.]



Example: Rotated Sorted Array

33. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that some prefix $A[1..k]$ is sorted in increasing order, the corresponding suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$.

    For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	1	3	4	5	7	8

(a) Describe and analyze an algorithm to compute the unknown integer k.

(b) Describe and analyze an algorithm to determine if the given array contains a given number x.

Example: Bisection Search

Problem: Given a **sorted** array of n integers $L[0:n]$ and a target number x, find x in the array if exsits and return its index, or -1 if x does not exist in the array.

```
# find x in a sorted array slice L[s:e]
# return i if L[i]==x and s <= i < e; otherwise return -1
def bisection_find(L,s,e,x):
    if e-s == 0:
        return -1
    if e-s == 1:
        return s if L[s] == x else -1
    m = (s+e)//2 # divide interval [s,e) into two halves
    if x < L[m]:
        return bisection_find(L,s,m,x)
    if x > L[m]:
        return bisection_find(L,m,e,x)
    if x == L[m]:
        return m
```

Exercise: Finding Changepoint

Your VC backed bank (fintech) provides an API as the only way to query your account balance. On some day you deposit money. You do not withdraw money. The API is `getBalance(day)` where day is an integer (0,1,2,...) indicating the days since you opened the account. `getBalance(day)` returns balance at the end of that day. You forgot which days you deposited money and you want to find out via querying the API. Do so with as few API calls as possible:

1) assume you only deposited once during days [a,b]

2) assume you deposited twice during days [a,b].

3) assume you deposited k times during days [a,b].

1) do you find it somewhat similar to bisection search?

```
# find the one changepoint between days [a,b]
def find_one_cp(a,b):
    bal_a = getBalance(a)
    bal_b = getBalance(b)
    if b-a == 1:
        assert bal_b > bal_a
        return b
    m = (a+b)//2 # mid point of a,b
    bal_m = getBalance(m)
    if bal_a < bal_m:
        return find_one_cp(a,m)
    if bal_m < bal_b:
        return find_one_cp(m,b)
```

Example: Symbolic Differentiation

Assuming a function of single variable x. Assume the function consists of exponentials, inverse, sin and cos, log, and +,-,*,/

Derivative rules: (f, g are functions of x, greek letters are constants)

- Constant rule: if f is constant e.g. $f(x) = \alpha$, then $f'(x) = 0$
- Sum rule: $(\alpha f + \beta g)' = \alpha f' + \beta g'$
- Product rule: $(fg)' = f'g + fg'$
- Quotient rule: $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$
- Chain rule: if $f(x) = h(g(x))$, then $f'(x) = h'(g(x))\, g'(x)$

- Special function derivatives: $\sin'(x) = \cos(x)$, $\cos'(x) = \sin(x)$, $\ln'(x) = 1/x$, $(x^\alpha)' = \alpha x^{\alpha-1}$, $(e^x)' = e^x$

Using these five rules plus derivatives of common functions, it's possible to compute the derivatives of an arbitrary function of x that consists of the arithmetics and composite function.

What is a function that we can differentiate? Like

$$f(x) = \sqrt{x}/\sin(1/x) + \ln(1 + e^x)$$

How do we **define** and **represent** these functions (let's call them **cool** functions)? Basic functions like $\sin(x), \cos(x), \ln(x), \text{const}(x, \alpha), \text{pow}(x, \alpha), \exp(x)$, and arithmetic between them, and also compositions of cool functions are cool.

Note that definition of **cool** functions are recursive. Definition and reprentation is closed related as representation is also recursive.

Here's one way to represent cool functions in Pythonic code:

- All Cool function is represnted as a list.
- The identify function $I(x) = x$ is a cool function, represented as `["I"]` (a list of a single string). It's the only cool function as a list with one element, the function name.
- Basic functions are cool: `["sin", ["I"]]`. (this means $\sin(x) = \sin(I(x))$. Functions except the identity has at least one argument: the parameter of the function.
  - `["const",["I"], a]` represents a constant function $f(x) = a$.
  - `["pow", ["I"], a]` represents a power function $f(x) = x^a$

- Arithmetic operators are represented as basic functions:
  - `[“add”, [“I”], ["const", 5]]` means $f(x) = x + 5$
- Composition of cool functions are also cool:
  - `["ln", ["add", ["pow", ["I"], -1], ["const", 2]]]` means $\ln\left(\frac{1}{x} + 2\right)$

According to this representation, the aforementioned function

$$f(x) = \sqrt{x}\sin(1/x) + \ln(1 + e^x)$$

```
["add",
  ["mul",
    ["pow", ["I"], 0.5],
    ["sin", ["pow", ["I"], -1]]
  ],
  ["ln",
    ["add", ["const", 1], ["exp", ["I"]]]
  ]
]
```

OK, now we can start writing a function to auto-diff a cool function.

```
def diff(F):
    if F[0] == "const":
        return 0
    if F[0] == "I":
        return ["const", 1]

    if F[0] == "add":
        return ["add", diff(F[1]), diff(F[2])]
    if F[0] == "sub":
        return ["sub", diff(F[1]), diff(F[2])]
    if F[0] == "mul":
        return ["add", ["mul", diff(F[1]), F[2]], ["mul", F[1], diff(F[2])]]
    if F[0] == "div":
        return ["div", ["sub", ["mul", diff(F[1]), F[2]], ["mul", F[1], diff(F[2])]],
["pow", F[2], 2]]

    if F[0] == "ln":
        return ["mul", ["pow", F[1], -1], diff(F[1])]
    if F[0] == "exp":
        return ["mul", ["exp", F[1]], diff(F[1])]
    if F[0] == "sin":
        return ["cos", diff(F[1])]
    if F[0] == "cos":
        return ["sin", diff(F[1])]
```

After applying to the above function:

```
diff(["Add",
          ["mul",
           ["pow", ["I"], 0.5],
           ["sin", ["pow", ["I"], -1]]
           ],
          ["ln",
           ["add", ["const", 1], ["exp", ["I"]]]
           ]
          ])
=> ['add', ['add', ['mul', ['mul', ['mul', ['const', 0.5], ['pow',
['I'], -0.5]], ['const', 1]], ['sin', ['pow', ['I'], -1]]], ['mul',
['pow', ['I'], 0.5], ['cos', ['mul', ['mul', ['const', -1], ['pow',
['I'], -2]], ['const', 1]]]]], ['mul', ['pow', ['add', ['const', 1],
['exp', ['I']]], -1], ['add', 0, ['mul', ['exp', ['I']], ['const',
1]]]]]
```

Hmm, it's kind of hard to verify it..I'm being a bit lazy here so how about we also write a function to evaluate a cool function, so that we can evaluate at a certain x and compare to Mathematica:

Exercise1: the output of `diff` can be very verbose, with a lot f(x)*0 kind of terms. Write a function to simplify a cool function.

Exercise2: The list representation of cool function is quite convenient to handle in the program, but no so much to read or write. Write translation routines to convert list formt to regular math formula style like ln(1+1/x) * sin(e^x)