Lecture 2: Recursion

Last updated: Feb 4, 2021

References:

- Algorithms, Jeff Erickson, Chapter 1

Recursion

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

```
peasantmultiply($x, y$):
    if $x = 0$                                    #base case
        return 0
    else
        $x' \leftarrow$ floor($x/2$)
        $y' \leftarrow y + y$
        prod$\leftarrow$peasantmutiply($x', y'$)             #recurse!
        if $x$ is odd
            prod$\leftarrow$prod$+y$
        return prod
```

Example: Tower of Hanoi

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

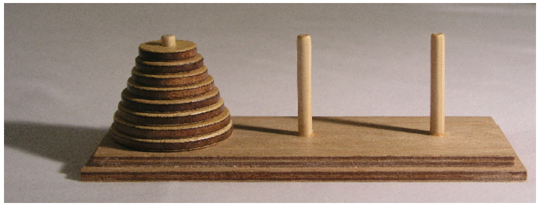


Figure 1. Tower of Hanoi. Image source: Wikipedia

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

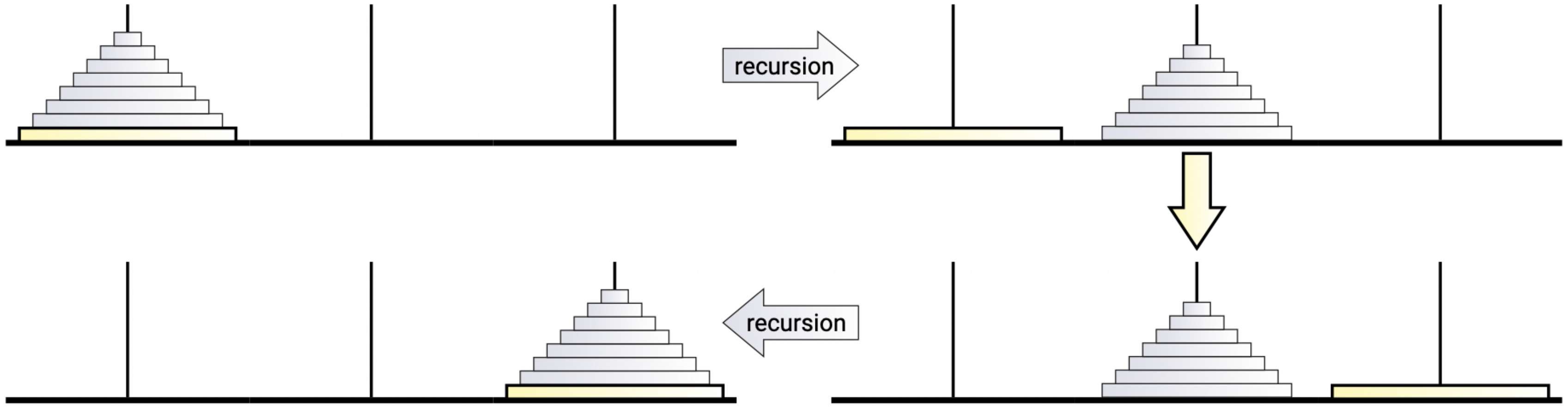
$\boxed{\text{to reduce the problem size, rather than solve it at once!}}$

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:

```
Hanoi(n, src, dst, tmp):
    if n>0
        Hanoi(n-1, src, tmp, dst)                #recurse!
        move disk n from src→dst
        Hanoi(n-1, tmp, dst, src)                #recurse!
    else                                         #base case
        do nothing
```

Do you see how easy to implement, and reason about the correctness & time complexity of recursive algorithm?

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

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

Hanoi Tower: Real Code

OK let's actually implement the code and see how it works.

```c
#include <stdio.h>

void hanoi(int n, int src, int dst, int tmp)
{
    if (n==0) return;
    hanoi(n-1, src, tmp, dst);
    printf("disk␣%d:␣peg␣%d␣->␣%d\n", n, src, dst);
    hanoi(n-1, tmp, dst, src);
}

int main()
{
    hanoi(3,0,2,1);
}
```

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.

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).

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.

```
merge(A[1..m],B[1..n],C[1..(m+n)]):          # merge A,B into C
    if m=0 or n=0, do the obvious thing.              #base case
    if A[m] > B[n]:
        C[m+n]←A[m]
        merge(A[1..m-1],B[1..n], C[1..(m+n-1)]         #recurse
    else
        C[m+n]←B[n]
        merge(A[1..m],B[1..n-1], C[1..(m+n-1)]         #recurse
```

Proof of correctness of merge()

Proof of the merge algorithm:

1. The base cases. It's trivial to see that the base case is correct.
2. Induction. Assuming that, merge() can merge two sorted array into a single sorted array with size up to m+n-1, we show that merge() also works for size m+n.
   i. C[m+n] is larger than all C[1..(m+n-1)]
   ii. C[1..(m+n-1)] will be sorted (induction hypothesis)
   iii. C[1..(m+n)] merges A[1..m] and B[1..n].

By proof of induction, we show that the merge() algorithm works for any input of any size. Similarly, we can prove the correctness of mergesort() with induction on the problem size n.

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:

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

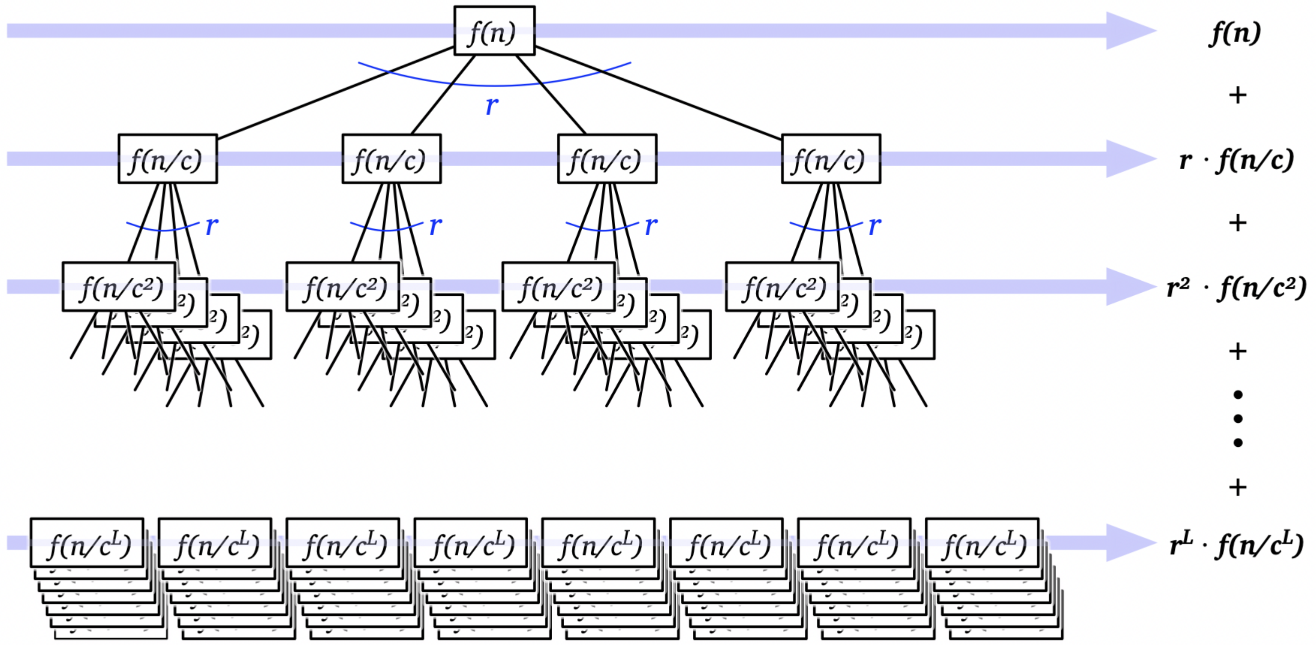


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

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:

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

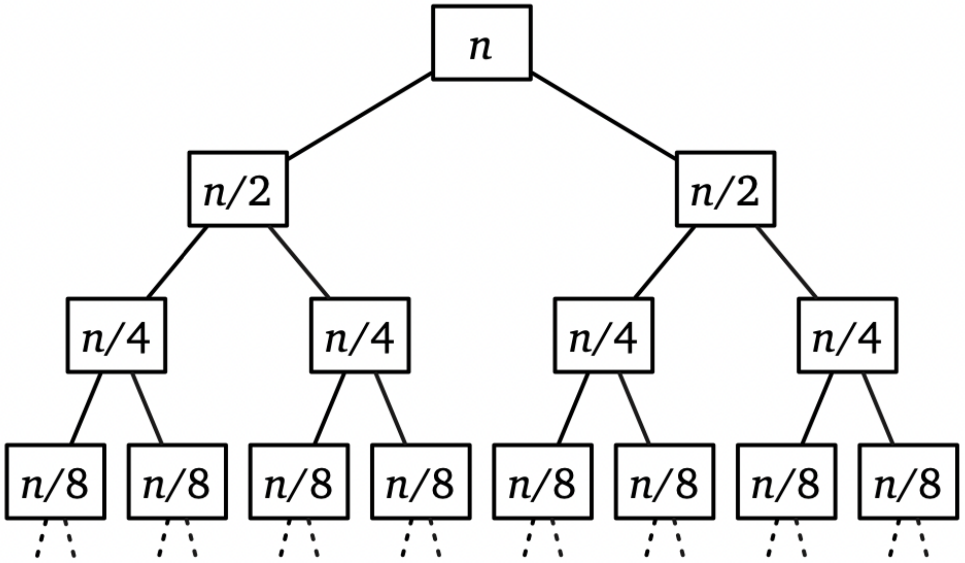
This solves to $\alpha \approx 2.0203$, which is a root to the previous equation.

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

So the solution is:

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

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.

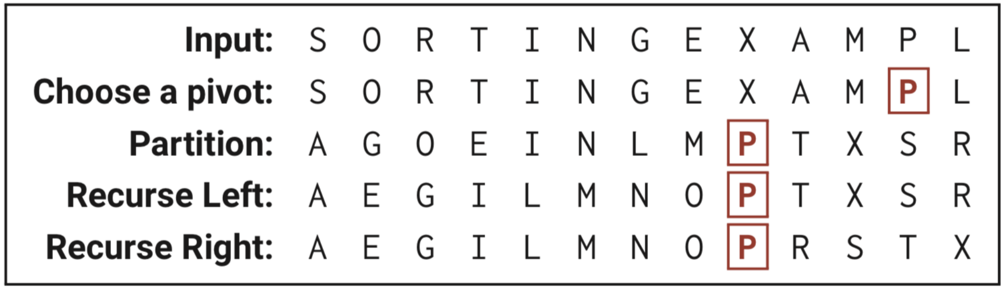
1. $T(n) = 2T(n/3) + 1$
2. $T(n) = T(n/3) + T(2n/3) + 1$
3. $T(n) = 3T(n-1) + 2$

QuickSort

Similar to MergeSort, QuickSort is another recursive sorting algorithm, and it's one of the fastest and most practical.

Different from MergeSort, QuickSort **partitions** the array into a **smaller** (value) array, and a **bigger** (value) array, and a **pivot**.

Once the two sub-arrays are sorted, there is no need to merge.



Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	$\boxed{P}$	L
Partition:	A	G	O	E	I	N	L	M	$\boxed{P}$	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	$\boxed{P}$	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	$\boxed{P}$	R	S	T	X

Figure 1.7. A quicksort example.

Figure 2.

Analysis

Similar to MergeSort, the time complexity of QuickSort is determined by a recurrence equation:

$$T(n) = T(r-1) + T(n-r) + O(n) \tag{5}$$

where r is the rank of pivot, also the size of the smaller array. How does r affect $T(n)$?

If r is very small like 2, or very large?

If r is exactly half of n, $r = \lfloor n/2 \rfloor$?

What if $r = \lfloor n/3 \rfloor$?

What if $r = \lfloor n/10 \rfloor$?

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.

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

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

```
SplitMultiply($x, y, n$):
    if $n = 1$ return $x \cdot y$
    $m \leftarrow \lceil n/2 \rceil$
    $a \leftarrow \lfloor x/10^m \rfloor$;    $b \leftarrow x \bmod 10^m$
    $c \leftarrow \lfloor y/10^m \rfloor$;    $d \leftarrow y \bmod 10^m$
    $e \leftarrow$ SplitMultiply($a, c, m$); $f \leftarrow$ SplitMultiply($b, d, m$);
    $g \leftarrow$ SplitMultiply($b, c, m$); $h \leftarrow$ SplitMultiply($a, d, m$);
    return $10^{2m} e + 10^m (g + h) + f$
```

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

```
FastMultiply($x, y, n$):
    if $n = 1$ return $x \cdot y$
    $m \leftarrow \lceil n/2 \rceil$
    $a \leftarrow \lfloor x/10^m \rfloor$;    $b \leftarrow x \bmod 10^m$
    $c \leftarrow \lfloor y/10^m \rfloor$;    $d \leftarrow y \bmod 10^m$
    $e \leftarrow$ FastMultiply($a, c, m$);
    $f \leftarrow$ FastMultiply($b, d, m$);
    $g \leftarrow$ FastMultiply($a - b, c - d, m$);
    return $10^{2m}e + 10^m(e + f - g) + f$
```

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!

Expontiation

$\boxed{\text{Given a number } a \text{ and positive integer } n\text{, compute } a^n}$

Naive algorithm that just perform $n - 1$ multiplications will cost linear time.

Is there a faster algorithm?

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{otherwise} \end{cases}$$

```
PIṄGALAPOWER($a, n$):
    if $n = 1$
        return $a$
    else
        $x \leftarrow$ PIṄGALAPOWER($a, \lfloor n/2 \rfloor$)
        if $n$ is even
            return $x \cdot x$
        else
            return $x \cdot x \cdot a$
```

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.

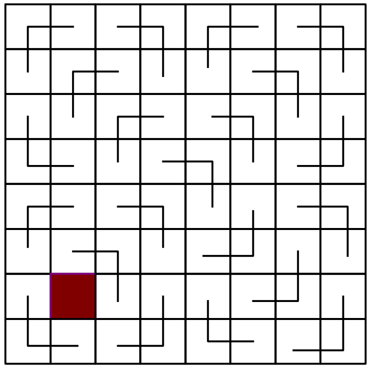
```
Hanoi1(n,src,dst,tmp):                                    #peg 0->1, 0->2
    Hanoi1(n-1,src,tmp,dst)
    move disk n to dst
    Hanoi2(n-1,tmp,src,tmp)
    Hanoi1(n-1,src,dst,tmp)

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.