Lecture 2: Recursion

Last updated: Sept 4, 2025

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 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 subproblem; rather, consider it somebody else's problem and it's solved.

Stop thinking about the solution of the subproblem!

Example: Peasant Multiply

The peasant multiple (recursive version)

```
// x, y are given by positive integers;
// returns x*y
fun peasant_recursive(x,y) {
    if (x==0) return 0;
    else if (x%2==0) return peasant_recursive(x/2, y+y);
    else return y + peasant_recursive(floor(x/2), y+y);
}

print peasant_recursive(123, 456); // expect 56088
```

Proof of correctness: [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.

Time complexity:

Let problem size be the number of bits in x: $n = \log_2 x$ and T(n) denotes work. Let m be the maximum bits of original x and y.

Then we have recurrence:

T(n) = T(n-1) + m

This easily solves to $T(n) = O(n\,m)$. The original problem time complexity would be $T(m) = O(m\,m) = O(m^2)$.

Bisection: Search in Sorted Array

In a sorted array A[0:n] of numbers, and a target number t, find t in the array if it exists, or correctly report t does not exist in A[0:n]. Design a fast algorithm.

Thinking: Try divide and conquer. Maybe we divide the array into two halves; then t can be in either half. We can recursively search for each half (two potential subproblems! Are they the same problem, just smaller?).

Since A[0:n] is sorted, we don't have to search in both halves; one would suffice. Which half? Well we look at A[n/2] and compare to t. If $t < A[n/2]$ then t is **only** possible in A[0:n/2]. If $t > A[n/2]$ then t is **only** possible in A[n/2:n]. By only a few comparison we reduce search space into half. Now our fast divide and conquer recursive solution is in sight.

Description: (omitting textual descrpition; same as thinking)

```
// input: sorted array A of size n, t is target number
// output: returns the index of t if t exists in A[i:j]; otherwise retu
fun bisection(A, t, i, j) {
    if (i >= j) return false; // base case: empty array cannot have tar
    if (i+1 == j) {
        if (A[i] == t) return i;
        else           return false;
    }
    var m = floor((i+j) / 2);
    if (A[m] == t)       return m;
    else if (A[m] < t)   return bisection(A, t, m, j);
    else                 return bisection(A, t, i, m);
}
// sample call
// A = [1, 3, 4, 6, 7];
// find 6 in the array:
// bisection(A, 6, 0, len(A)); // expects 3
```

correctness proof: (full version; later we can drop some of the induction setup for brevity).

Goal: this function/algorithm bisection(A, t, i, j) correctly finds the index of t in sorted A[i:j], or report false if t is not in A[i:j].

Use mathematical induction on the problem size n = j-i.
Base cases: for n = 0 or 1, the bisection algorithm obviously works (see bisection algorithm base cases).
Induction hypothesis: for any j-i < n, bisection(A, t, i, j) correctly finds the index of t or report false.
Now we prove bisection(A, t, i, j) also works for n = j-i, and n >= 2.

Take the middle index of A[i:j]: m=(i+j)/2.
Case 1: If A[m] == t then we found it and return m.
Case 2: A[m] < t, because A is sorted, t cannot be in the first half A[i:m]; according to induction hypotheis, bisection(A, t, m, j) correctly tells us whether t is in second half A[m:j] which tells us

whether t is in A[i:j]
case 3: A[m] > t, likewise t cannot be in the second half A[m:j];
according to induction hypotheis, bisection(A, t, i, m) correctly tells us whether t is in the first half A[i:m]

There are no other cases. QED.

Time complexity:

For problem size n (n=j-i), say the time complexity of bisection() is T(n).
Because bisection does constant amount of work, in addition may go to exactly one recursive call
on half size problem (either m-i, or j-m), we derive recurrence:

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

This solves to $T(n) = O(\log n)$

Example: Tower of Hanoi

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

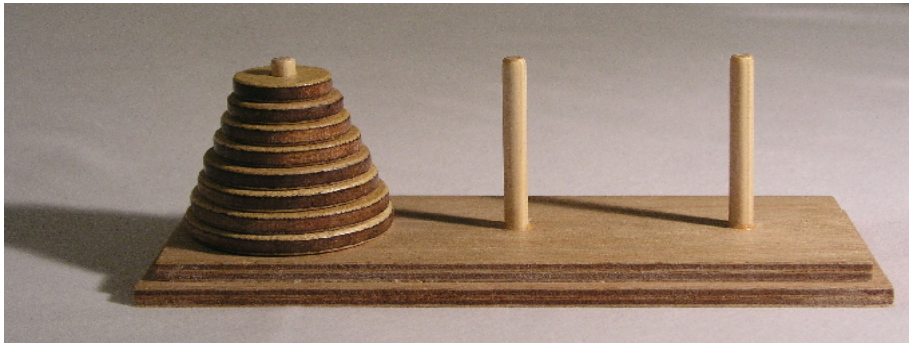


Figure 1. Tower of Hanoi. Image source: Wikipedia

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

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

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.

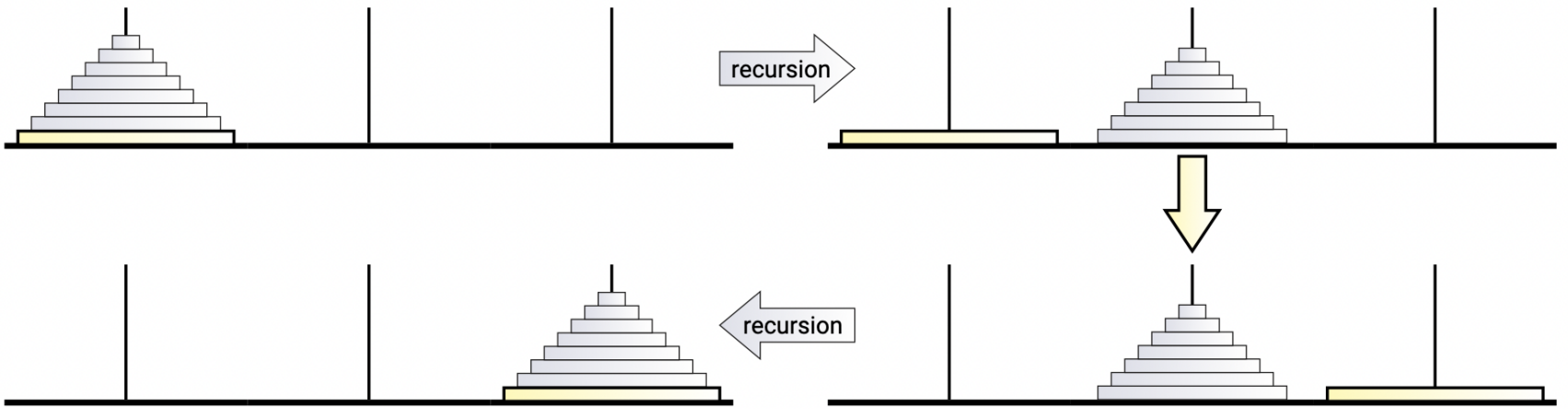
The secret to solve this problem is

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).

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)



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.

The formal algorithm:

```
// prints moves that move top n disks from src peg to
// dst peg; with tmp peg as temporary holder peg
fun hanoi(n, src, dst, 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);
}

hanoi(3, 0, 2, 1);
```

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

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

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

QuickSort

Let's review quicksort (Hoare version) algorithm. It's recursive:

1. Partition the array into two subarrays with a pivot; left subarray is less than pivot and right subarray is larger than pivot

2. Recursively qsort the two subarrays

3. done

```
// sort subarray A[lo:hi] in ascending order defined by less function
fun qsort(A, lo, hi, less) {
    if (hi-lo <= 1) return; // do nothing
    var p = partition(A, lo, hi, less);
    qsort(A, lo, p, less);
    qsort(A, p+1, hi, less);
}
```

Proof is as every other recursive algorihtm case: mathematically induction on the size of the subarray. Assuming the said algorithm works for all instances of subarray size (hi-lo) $<$ n. Prove that it also works for any instance of subarray of size $=$ n. (details omitted)□

Now the `partition()` needs to shuffle the array a bit, moving all elements less than pivot to the left, larger elements to the right. And

also return the pivot position.

This particular partition uses a variant of Hoare's proposal. There is a nice animation on the wikipedia page: https://en.wikipedia.org/wiki/Quicksort

```
// return pivot index p; partiion the array such that
// A[lo:p] is less than A[p];
// A[p+1:hi] is greator than A[p];
fun partition(A, lo, hi, less) {
    assert(lo < hi);
    var pivot = A[lo]; // use first elem as pivot
    var i = lo+1;
    var j = hi-1;
    while (true) {
        while (i < hi and less(A[i], pivot)) i = i + 1;
        while (j > lo and less(pivot, A[j])) j = j - 1;
        if (i >= j) {
            swap(A, lo, j);
            return j;
        }
        swap(A, i, j);
    }
}
```

```
var A = [3, 4, 2, 6, 5];
print partition(A, 0, 5, fun(a,b){return a<b;}); // expects 1
print A; // expects [2, 3, 4, 6, 5,];

var A = [3, 4, 2, 6, 5];
qsort(A, 0, len(A), fun(a,b) {return a<b;} );
print A; // expects [2,3,4,5,6];
```

Time complexity:

From `qsort()` body we can see that it calls itself 2x, and non-recursive work is in `partition` which is $O(n)$. Let $T(n)$ denote the work that qsort() does on size n array. Then we have recurrence:

$$T(n) = T(n_1) + T(n_2) + n$$

Where n_1, n_2 are the subproblem sizes determined by the `partition` function. We have $n_1 + n_2 = n$, but there is no guaranttee that n is divided between n_1, n_2.

What's the worst case? What's the best case?

Worst case: $T(n) = T(1) + T(n-1) + n \Rightarrow T(n) = O(n^2)$

Best case: $T(n) = T(n/2) + T(n/2) + n \Rightarrow T(n) = O(n \log n)$

It all depends on `partition()`! If partition always uses median as pivot then we have best case. But that could be expensive.

How about we have a linear time, relatively cheap partition that does not guaranttee even division $n_1 = n_2$, but rather it guaranttees that

$$n/3 \leqslant n_1, n_2 \leqslant 2n/3$$

I.e. almost even, the bigger part is no bigger than 2x the smaller part? Then we have worst case time complexity

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

How do you solve this? Does master theorem cover this?

(Spoiler alert: at the end of this slides we'll know how to solve it; and it solves to $T(n) = O(n \log n)$ still, the same as perfect partition!)

Further more what if a weaker partition:

$$T(n) = T(n/10) + T(9n/10) + n$$

?

Further considerations.

We can apply the same idea of `qsort` to the problem of **selection**, namely given a (unsorted) array, find the k-th smallest element. Trivial solution would be to sort in $O(n \log n)$ and pick the k-th. We want faster selection algorithm in $O(n)$.

How about we do similar trick to `qsort`: partition then recurse?

```
// select the k-th smallest elements from array A[lo:hi]
fun qsel(A, lo, hi, k) {
    if (lo + 1 == hi) return A[lo];
    var p = partition(A, lo, hi, less);

    if      (k < (p-lo)) return qsel(A, lo, p, k);
    else if (k > (p-lo)) return qsel(A, p, hi, k-(p-lo));
    else                 return A[p];
}
```

It's recursive alright, and it looks almost like `qsort`. Its correctness is not in doubt (proof omitted here).

What about its time complexity? Like qsort, depends on the pivot. The closer pivot is being the median, the better. Worst case for `qsel` is $T(n) = T(n-1) + n \quad \Rightarrow T(n) = O(n^2)$.

With a good pivot, we can reduce it to **linear!** In fact here is one, call MoMSelect[1], (MoM is median of medians).

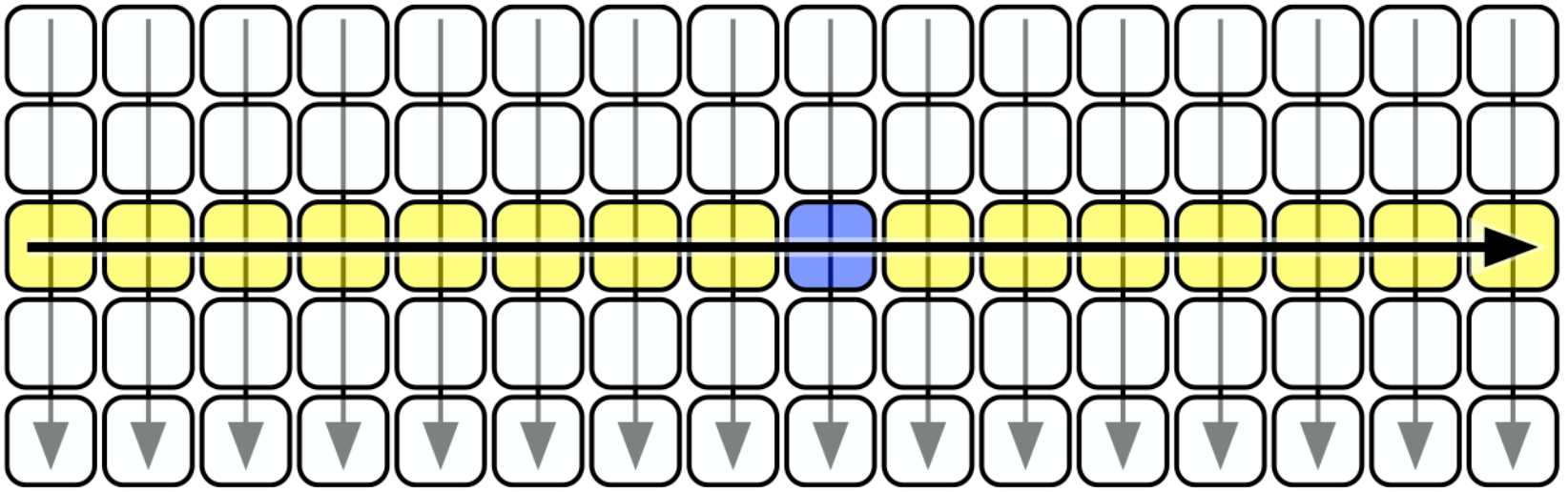
---

1. full rhythm implementation: https://github.com/robbwu/rhythm/blob/main/examples/mom_select.rhy

```
// select the k-th smallest elements from array A[lo:hi]
// returns its index in A.
fun momsel(A, lo, hi, k) {
    if (lo + 25 > hi) {
        // omitted here; use brute force
    }
    var n = hi - lo;
    var m = ceil(n/5);
    var M = [];
    for (var i=0; i<m; i=i+1) push(M, median_of_5(A, i*5));
    var mom_in_M = momsel(M, 0, m, floor(m/2));
    // get mom_in_A from mom_in_M doing index translation
    var p = partition2(A, lo, hi, less, mom_in_A); // partition using given pivot A[mom_in_A]

    if      (k < (p-lo)) return momsel(A, lo, p, k);
    else if (k > (p-lo)) return momsel(A, p, hi, k-(p-lo));
    else                 return A[p];
}
```

Note that we use momsel itself to find a good pivot.



What's the quality of the MoM pivot? In fact at least $3n/10$ elements are smaller than it and at most $7n/10$ elements are bigger than it.

By using this MoMo as pivot our second recursive call has size no larger than $T(7n/10)$. Now the time complexity:

$$T(n) = \underbrace{T(n/5)}_{\text{MoM recursive}} + \underbrace{T(7n/10)}_{\text{one side of partition recursive}} + \Theta(n)$$

This solves to $T(n) = \Theta(n)$!

(How? For now you can just try prove it by showing $T(n) = O(n)$, and $T(n) = \Omega(n)$ by induction).

Later at the end of these slides, we can use the tree recursion trick to derive the solution directly.

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"?

```
// compute x ~ sqrt(a);
// give a rough interval [t0,t1] that contains sqrt(a);
// where t0*t0 < a,  t1*t1 > a
// stop when found x such that abs( x*x-sqrt(a) ) < eps
fun sqrt_rec(a, t0, t1, eps) {
    assert(a > 0);
    assert(t0 < t1);

    var t = (t0+t1) / 2;
    if (fabs( t*t - a ) < eps) {
        return t; // fonnd close enough solution
    }
    if (t*t < a) return sqrt_rec(a, t, t1, eps);
    if (t*t > a) return sqrt_rec(a, t0, t, eps);
}

// sample call: find sqrt(2):
print sqrt_rec(2, 0, 2, 0.000001); // expects 1.41421
```

This looks like a “continuous” version of besection in sorted array?

Correctness proof: This is different than usual as the problem size (the interval size (t1-t0)) is not an integer so mathematical induction does not work out of box. Instead we can do induction on the depth of recursive calls k. (details omitted; almost the same as bisection in sorted array)

Time complexity: This is also involved; essentially how many "halving" until a close enough solution is found, which depends on the eps is set.

But intuitively, each halving gains us 1-bit of accuracy; so this algorithm is very fast; within the practical floating point representation like float64, roughly at most 64 recursive steps (plus a few that depends on the initial interval t0,t1).

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 Rhythmic 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. **You must state the claim that you are proving clearly! (namely, the purpose of the recursive function)**

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

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.

Time Complexity of Recursive Algos

(Here we briefly talk about time complexity of recursive algorithms, and at the end we'll discuss at length and in more general form.)

What's the time complexity of recursive algo? Look at the (recursive function using divide and conquer); say time complexity of size n problem is T(n), then:

$$T(n) = f(n) + T(n_1) + \ldots + T(n_k)$$

where $n_1, \ldots, n_a$ are the k-subproblem sizes; f(n) is the non-recursive work, also called per-frame work. Often $n_1 = n_2 = \cdots = n_a = n/b$, so it's

$$T(n) = f(n) + a\,T(\tfrac{n}{b})$$

For now we just recall the master theorem:

- 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

Typical example:

(Mergesort) T(n)=2T(n/2)+n $\Rightarrow$ T(n)=O(n log n)

(Bisection) T(n)=T(n/2)+1 $\Rightarrow$ T(n)=O(log n)

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

Another seeming cheating way to solve recurrence is

Guess and verify.

Which is very powerful! Try it out on the previous examples.

Like this one:

Solve $T(n) = T(n/2) + \Theta(n)$

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 (m = n/2), 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
```

Correctness is easy to show using induction and the equation. 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?

```
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
```

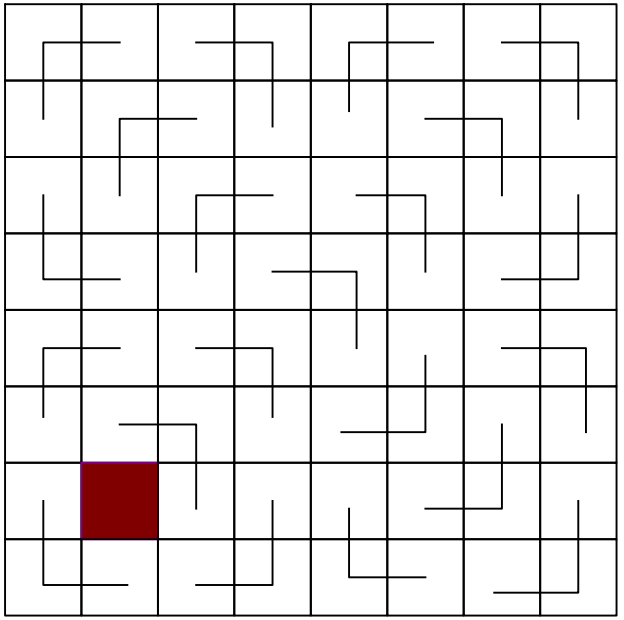
OK, now the time complexity is

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

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

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.]



Working with Trees

Trees are recursive structure and are mostly time most conveniently and efficiently dealt with recursive algorithms.

Take binary search tree. Let's create a binary search tree by adding a list of numbers one by one to the binary search tree:

```
// Binary Search Tree with in-order traversal
// Node represented as a Map: {"value": val, "left": left_node, "right": right_node}
// Note that the accessor syntax root.left is syntax sugar for root["left"].

// insert a new node of `value` into a binary search tree rooted at `root`
fun insert(root, value) {
    if (root == nil) return {"value": value, "left": nil, "right": nil};
    if (value < root.value)
        root.left = insert(root.left, value);
    else
        root.right = insert(root.right, value);
    return root;
}
```

we can then add numbers to form the binary search tree:

```
var tree = nil;
var values = [5, 3, 7, 1, 9, 4, 6, 2, 8];
for (var i = 0; i < len(values); i = i + 1) {
    tree = insert(tree, values[i]);
}
```

Hmm, it would be great to print out the tree…

```
// print the binary tree rooted at `node`, with `depth` indent
fun print_tree(node, depth) {
    if (node == nil) return;

    print_tree(node.right, depth + 1);  // right subtree (upper part)

    // print current node with proper indentation
    var indent = "";
    for (var i = 0; i < depth; i = i + 1) {
        indent = indent + "    ";  // 4 spaces per level
    }
    printf("%s%d\n", indent, node.value);

    print_tree(node.left, depth + 1);   // left subtree (lower part)
}
```

The resulting tree prints to

```
print "Binary Search Tree (rotated 90° clockwise):";
print_tree(tree, 0);
```

var values = [5, 3, 7, 1, 9, 4, 6, 2, 8];

```
Binary Search Tree (rotated 90° clockwise):
      9
         8
   7
      6
5
      4
   3
         2
      1
```

Now we can do a in-order traversal of the binary search tree to validate that it's in fact a search tree:

```
// in-order printouts of binary tree
fun traverse_inorder(root) {
    if (root == nil) return;
    traverse_inorder(root.left);
    printf("%d ", root.value);
    traverse_inorder(root.right);
}
print "in order traversal of tree";
traverse_inorder(tree);
```

It prints out:

```
in order traversal of tree
1 2 3 4 5 6 7 8 9
```

Hooray!

Pretty much everything you do with binary tree, you do with recursion.

Traversal of binary tree: in-order, pre-order, post-order

```
fun traverse_inorder(root) {
    if (root == nil) return;
    traverse_inorder(root.left);
    printf("%d ", root.value);
    traverse_inorder(root.right);
}

fun traverse_preorder(root) {
    if (root == nil) return;
    printf("%d ", root.value);
    traverse_inorder(root.left);
    traverse_inorder(root.right);
}
fun traverse_postorder(root) {
    if (root == nil) return;
    traverse_inorder(root.left);
    traverse_inorder(root.right);
    printf("%d ", root.value);
}
```

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.

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)

Parsing: Recursive Descent

Parsing is the process of turning a sequence of tokens into an abstract syntax tree (AST) of a given (context-free) **grammar**.

The simplest algorithm for parsing is called **recursive descent**; As the name suggest it's a (very) recursive algorithm. Beat uses it to parse a tokenized Rhythm program to an Abstracted Syntax Tree (AST) [2]. It's also used in GCC, Chrome V8, etc.

We won't go into details; but check the nice illustration in Crafting Intepreter[3]. Here we just note that both grammar, program, and AST are recursive structures.

---

2. https://github.com/robbwu/rhythm/blob/main/src/parser.hpp

3. https://craftinginterpreters.com/parsing-expressions.html

Solving Recurrence

How to solve recurrence equation like this:

$$T(n) = r\,T(n/c) + f(n) \tag{1}$$

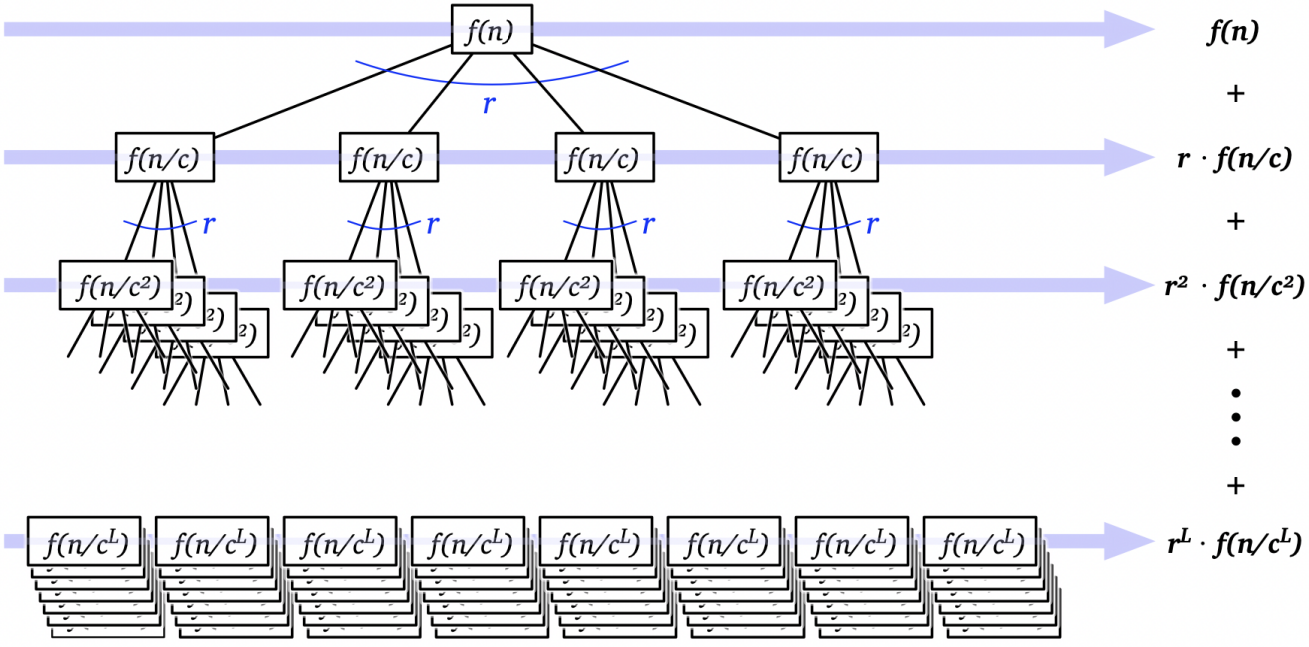
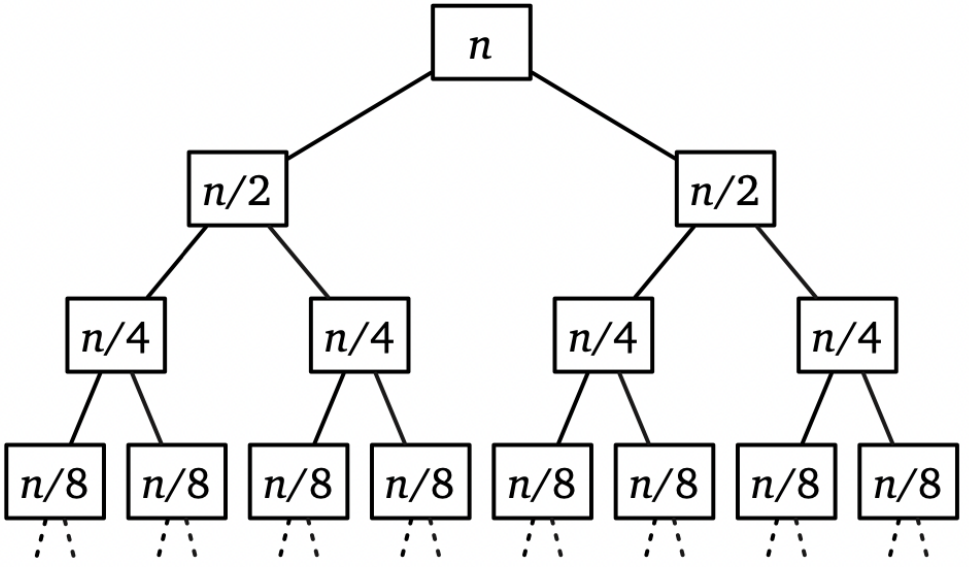


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

Example: mergesort recurrence



In mergesort we have $T(n) = 2\,T(n) + n$

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

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

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:

$$T(n) = r\,T(n/c) + f(n)$$

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

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

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: $T(n) = T(n/5) + T(7n/10) + n$