Recursion

QiShi 11th Algorithm Study Group

Fundamentals

Mathematical Induction vs Recursion

- Mathematical Induction

Prove $\sum_{i=0}^{m} \boldsymbol{i} = \frac{\boldsymbol{m(m+1)}}{\boldsymbol{2}}$ $\boldsymbol{(1)}$

Proof:

When $\mathrm{m} = 0$, $LHS = 0 = \frac{0(0+1)}{2} = \mathrm{RHS}$ Base Case

Assume (1) is correct for $m = n - 1$ Recurrence Relation

Then for $\mathrm{m} = n$, $\mathrm{LHS} = \sum_{i=0}^{n} \boldsymbol{i} = \boldsymbol{n} + \sum_{i=0}^{n-1} \boldsymbol{i} = \boldsymbol{n} + \frac{\boldsymbol{(n-1)(n-1+1)}}{\boldsymbol{2}} = \frac{\boldsymbol{n(n+1)}}{\boldsymbol{2}} = \boldsymbol{RHS}$

- Recursion

```
def n_sum(n):
    if n == 0:
        return 0
    else:
        return n + n_sum(n-1)
```

Recursion is different from **Mathematical Induction** in that you do not need to come up with a general formula. You assume n_sum works for n-1 and let the recursion and computer do their wonder.

Keys to Recursion

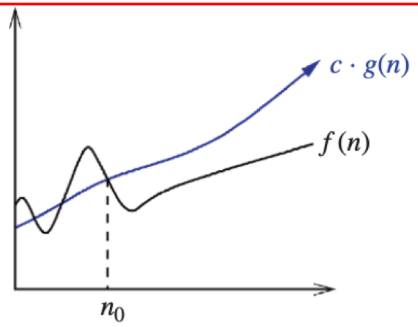
Recursion is an approach to solving problems using a function that **calls itself** as a subroutine.

To make sure the recursion will not cause infinite loop, we need two key ingredients:

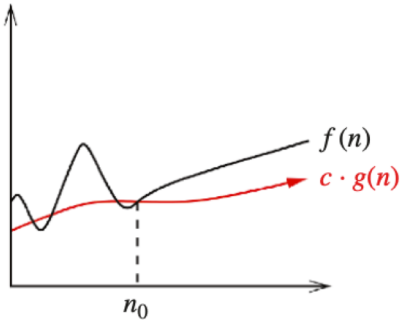
- A simple base case (or cases) — a terminating scenario that does not use recursion to produce an answer.
- A set of rules, also known as recurrence relation that reduces all other cases towards the base case.

How good is the algorithm?

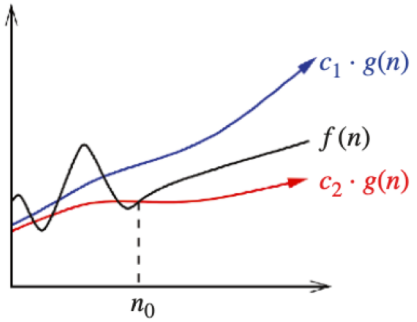
How to Evaluate an Algorithm – Time and Space Complexity



f(n) = O(g(n)) means c · g(n) is an upper bound on f(n). Thus, there exists some constant c such that f (n) ≤ c · g(n) for every large enough n (that is, for all n ≥ n0, for some constant n0).

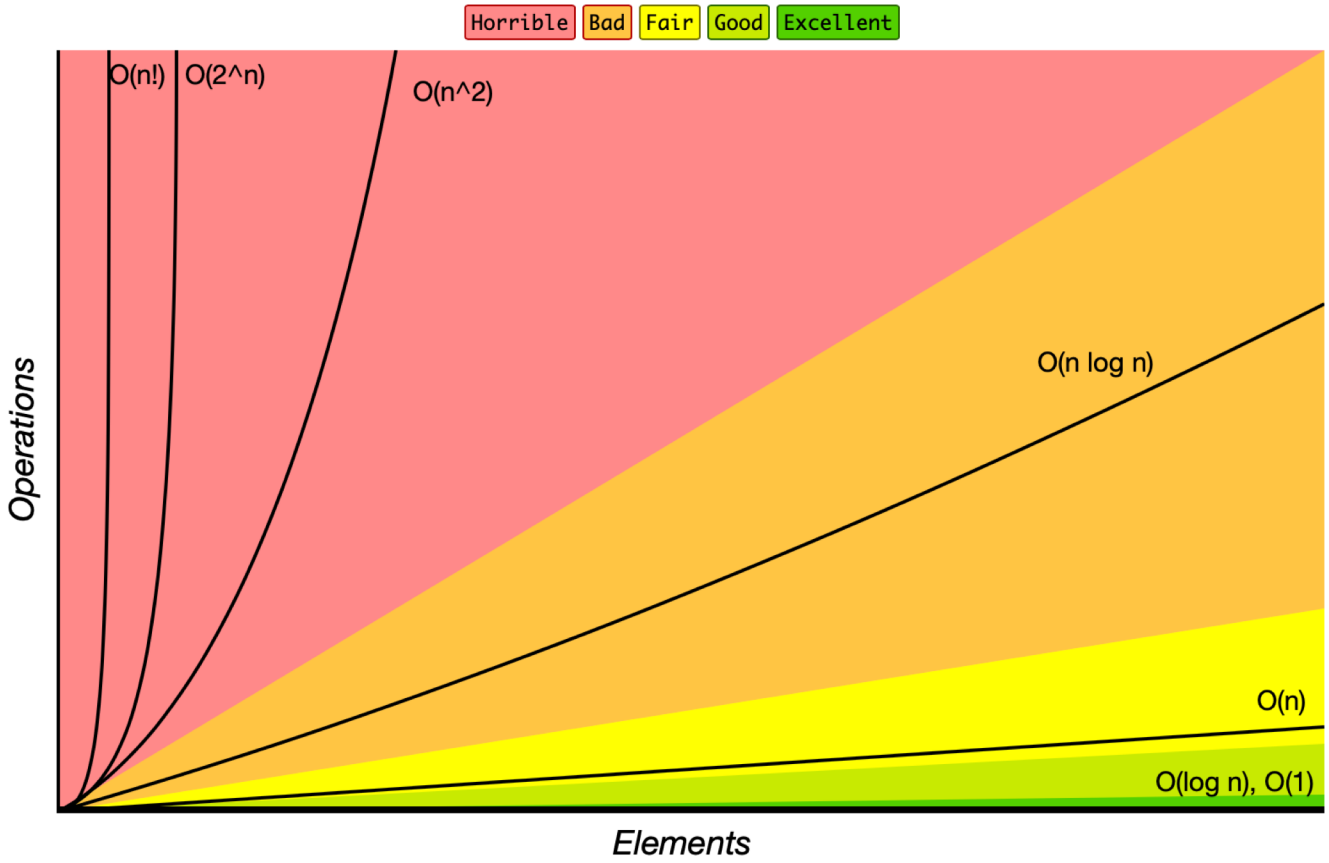


f(n) = Ω(g(n)) means c·g(n) is a lower bound on f(n). Thus, there exists some constant c such that f(n) ≥ c · g(n) for all n ≥ n0.



f(n) = Θ(g(n)) means c1 · g(n) is an upper bound on f(n) and c2 · g(n) is a lower bound on f(n), for all n ≥ n0. Thus, there exist constants c1 and c2 such that f(n) ≤ c1 ·g(n) and f(n) ≥ c2 ·g(n) for all n ≥ n0. This means that g(n) provides a nice, tight bound on f(n).

Big O Complexity Chart



https://www.bigocheatsheet.com

Put Time Complexity into Perspective

Suppose each operation costs 1 nanosecond

n	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

$O(2^n)$ and $O(n!)$ algorithms are almost surely unusable when n is large!

Recursion Space Complexity and Stack Overflow

Time and Space Complexity of n_sum

```python
def n_sum(n):
    if n == 0:
        return 0
    else:
        return n + n_sum(n-1)
```

Time Complexity is clearly **O(n)**, but how about Space Complexity?

Time and Space Complexity of n_sum

```python
def n_sum(n):
    if n == 0:
        return 0
    else:
        return n + n_sum(n-1)
```

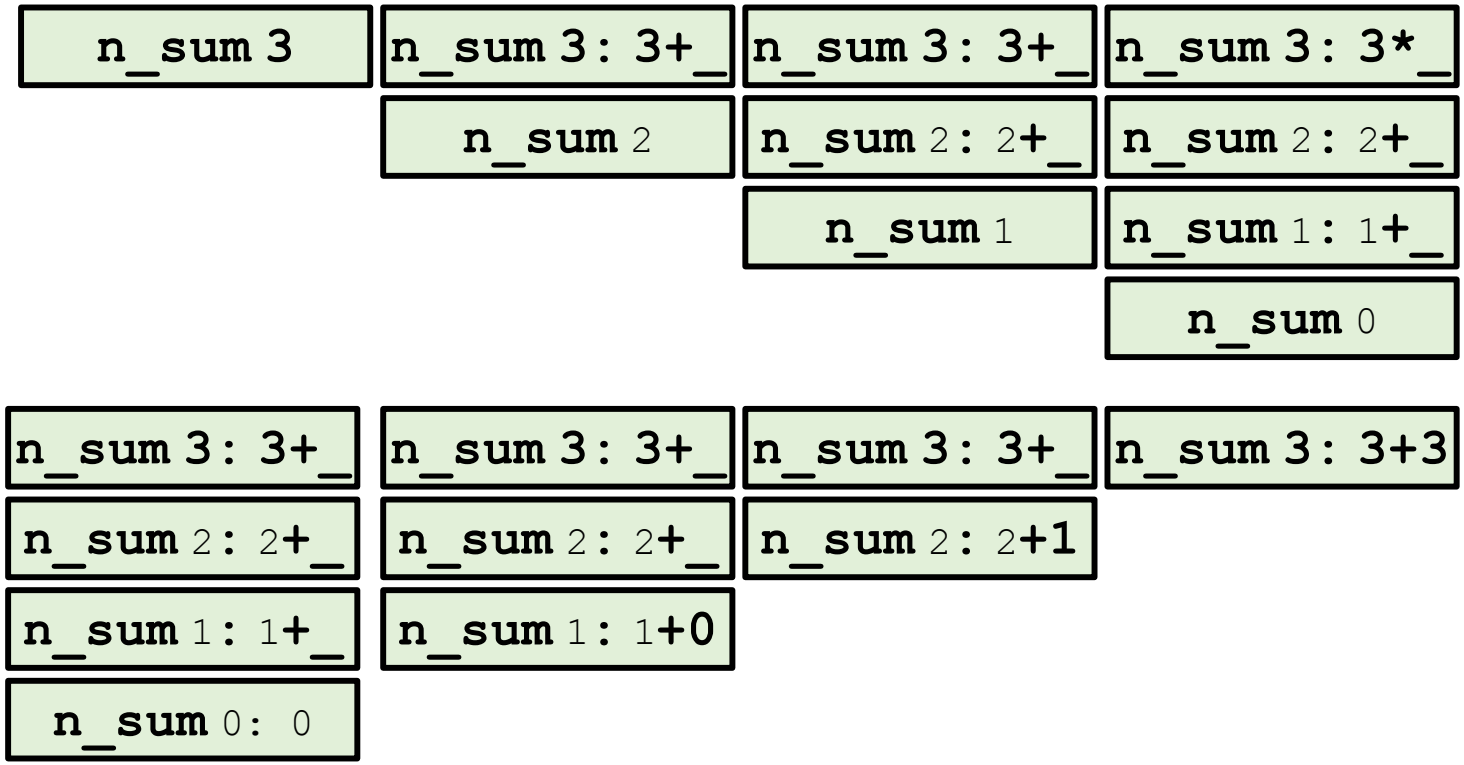
Time Complexity is clearly **O(n)**, but how about Space Complexity? In order to understand its space complexity, we need to know how function call works in the memory.

STACK MEMORY	HEAP MEMORY
small size	large size
fast access	slow access
stores function calls and local variables	stores objects
no fragmentation	may become fragmented

Time and Space Complexity of n_sum

```python
def n_sum(n):
    if n == 0:
        return 0
    else:
        return n + n_sum(n-1)
```

Time Complexity is clearly **O(n)**, but how about Space Complexity?
Space Complexity: **O(n)**



Time and Space Complexity of n_sum

You can try to see it yourself

```python
import inspect

def print_frames(frame_list):
    module_frame_index = [i for i, f in enumerate(frame_list) if f.function == '<module>'][0]
    for i in range(module_frame_index):
        d = frame_list[i][0].f_locals
        local_vars = {x: d[x] for x in d}
        print("  [Frame {} '{}': {}]".format(module_frame_index - i, frame_list[i].function, local_vars))
    print("  [Frame '<module>']\n")

def n_sum(n):
    if n == 0:
        print("n_sum({}) called:".format(n))
        print_frames(inspect.stack())
        print("n_sum({}) returned {}".format(n, 0))
        return 0
    else:
        print("n_sum({}) called:".format(n))
        print_frames(inspect.stack())
        result = n + n_sum(n-1)
        print_frames(inspect.stack())
        print("n_sum({}) returned {}".format(n, result))
        return result

n_sum(2)
```

```
n_sum(2) called:
  [Frame 1 'n_sum': {'n': 2}]
  [Frame '<module>']

n_sum(1) called:
  [Frame 2 'n_sum': {'n': 1}]
  [Frame 1 'n_sum': {'n': 2}]
  [Frame '<module>']

n_sum(0) called:
  [Frame 3 'n_sum': {'n': 0}]
  [Frame 2 'n_sum': {'n': 1}]
  [Frame 1 'n_sum': {'n': 2}]
  [Frame '<module>']

n_sum(0) returned 0
  [Frame 2 'n_sum': {'n': 1, 'result': 1}]
  [Frame 1 'n_sum': {'n': 2}]
  [Frame '<module>']

n_sum(1) returned 1
  [Frame 1 'n_sum': {'n': 2, 'result': 3}]
  [Frame '<module>']

n_sum(2) returned 3
```

Stack Overflow

If the recursion is too deep, it will cause stack overflow.
Or in case of Python, it will cause, maximum recursion depth exceeded

```
n_sum(3000)
```

```
RecursionError                            Traceback (most recent call last)
<ipython-input-18-bf104aed8105> in <module>
----> 1 n_sum(3000)

<ipython-input-16-0e7a5dce633a> in n_sum(n)
      3         return 0
      4     else:
----> 5         return n + n_sum(n-1)

... last 1 frames repeated, from the frame below ...

<ipython-input-16-0e7a5dce633a> in n_sum(n)
      3         return 0
      4     else:
----> 5         return n + n_sum(n-1)

RecursionError: maximum recursion depth exceeded in comparison
```

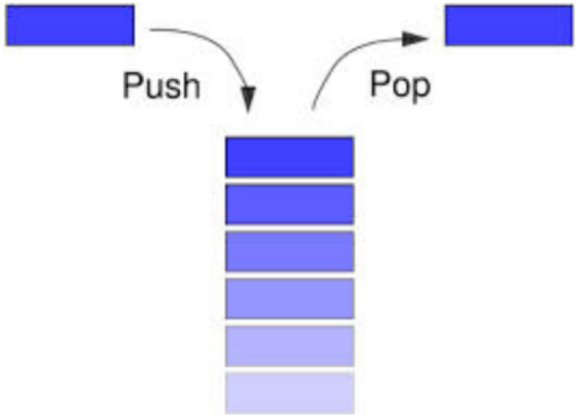
```
import sys
sys.getrecursionlimit()
```

```
3000
```

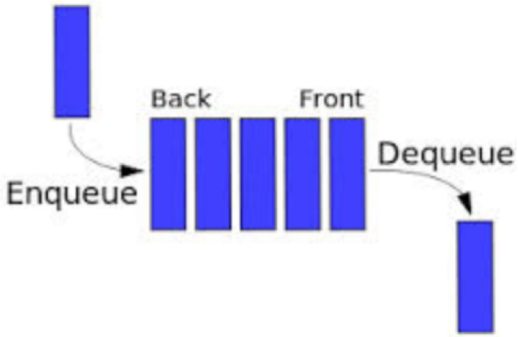
Stack overflow can sometimes be avoided by constructing **tail recursion** if the programming language is optimized for tail recursion (Python is not, see Guido van Rossum's posts *Tail Recursion Elimination* and *Final Words on Tail Calls*).
If the programming language is not optimized for tail recursion, one can usually convert tail recursion into a loop.

Unfold Recursion

If you are asked to **remove recursion** in your program, consider mimicking call stack with the **stack data structure**. If the function call order does not matter, you can change stack to queque as well. (Link to other Algorithms: both DFS and BFS can traverse a tree but in different order. One uses stack data structure and the other one uses queue structure.)



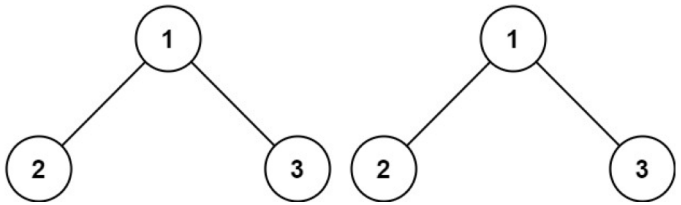
Stack (FILO)



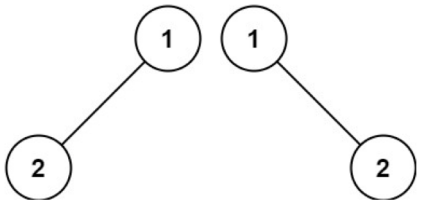
Queue (FIFO)

Unfold Recursion Example: Same Tree

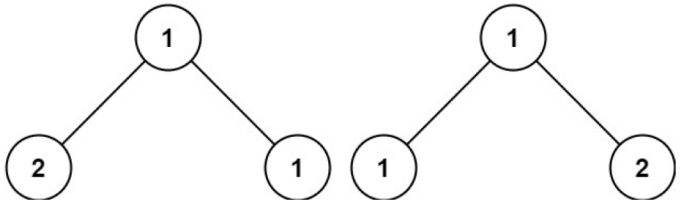
Given the roots of two binary trees **p** and **q**, write a function to check if they are the same or not



True



False



False

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

Same Tree – Two Solutions

Recursion

```python
def is_same_tree(p, q):
    # p and q are both None
    if not p and not q:
        return True
    # one of p and q is None
    if not q or not p:
        return False
    if p.val != q.val:
        return False
    return is_same_tree(p.right, q.right) and \
           is_same_tree(p.left, q.left)
```

Time Complexity: $O(n)$
Space Complexity: $O(n)$ (it should be the depth of the tree but the worse case is n)

Unfold Recursion using Loop and Queue

```python
from collections import deque

def is_same_tree(p, q):

    def check(p, q):
        # if both are None
        if not p and not q:
            return True
        # one of p and q is None
        if not q or not p:
            return False
        if p.val != q.val:
            return False
        return True

    deq = deque([(p, q),])
    while deq:
        p, q = deq.popleft()
        if not check(p, q):
            return False
        if p:
            deq.append((p.left, q.left))
            deq.append((p.right, q.right))
    return True
```

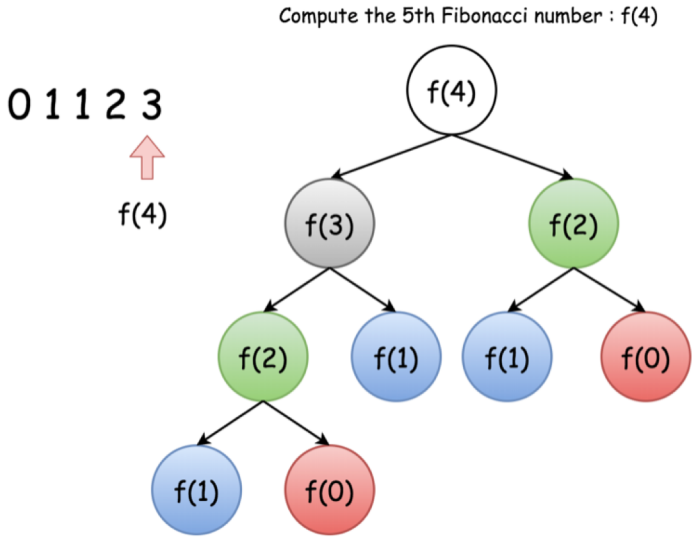
Recursion Time Complexity and Memoization

Tree Recursion

If your function is called more than once in your recursion formula and the size of problems that are called on only reduces at linear rate (in contrast to **Divide and Conquer** that conforms to the **Master Theorem**), you will end up with a tree recursion with time complexity at **exponential rate**.

Example: calculate the nth Fibonacci number

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



- Time complexity is proportional to the number of tree nodes, i.e. $\boldsymbol{O(2^n)}$. The problem is that there are **too many repetitive calculations**.
- Space complexity is proportional to the maximum depth of the tree, i.e. $\boldsymbol{O(n)}$ (Think about the call stack. Very similar to the DFS in recursion form.)
- You will not run into stack overflow problem with fib(100) but it will be almost impossible for you to get the answer at any reasonable time.

Memoization

We can reduce the time complexity to $\boldsymbol{O(n)}$ (from $\boldsymbol{O(2^n)}$) by caching some intermediate results through **Memoization** (this is trading space for time and space is cheap nowadays). Dynamic programming is a more general way of doing this.

```
def memo(f):
        """Return a memoized version of
        single-argument function f."""
        cache = {}
        def memoized(n):
            if n not in cache:
                cache[n] = f(n)
            return cache[n]
        return memoized

@memo
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Summary so far

- When in doubt, write down the **recurrence relationship**.
- Whenever possible, apply **memoization**. (Reduce time complexity)
- When stack overflows, **tail recursion/loop** might come to help. (Reduce space complexity)

More Examples

What Recursion is Suitable for*

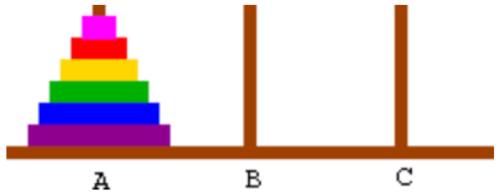
1. Things involving stack data structure: The function call stack naturally provide a way to implement algorithm that requires a stack data structure without explicitly using stack, e.g. DFS.
2. Divide and conquer: A divide-and-conquer algorithm works by recursively breaking the problem down into two or more subproblems of the same or related type, until these subproblems become simple enough to be solved directly. You might or might not need to combine the results of subproblems to form the final solution (e.g. merge sort vs binary search).
3. Backtracking
4. Things involving tree or graph traversing
5. Loops: You can usually replace a loop with recursion. Functional programming languages tend to use recursion instead of loop. Some of them might even not provide loop construct.

* You will see some of these in practice problems at the end

Tower of Hanoi

The objective of the puzzle is to move the entire stack from A to B, obeying the following simple rules:

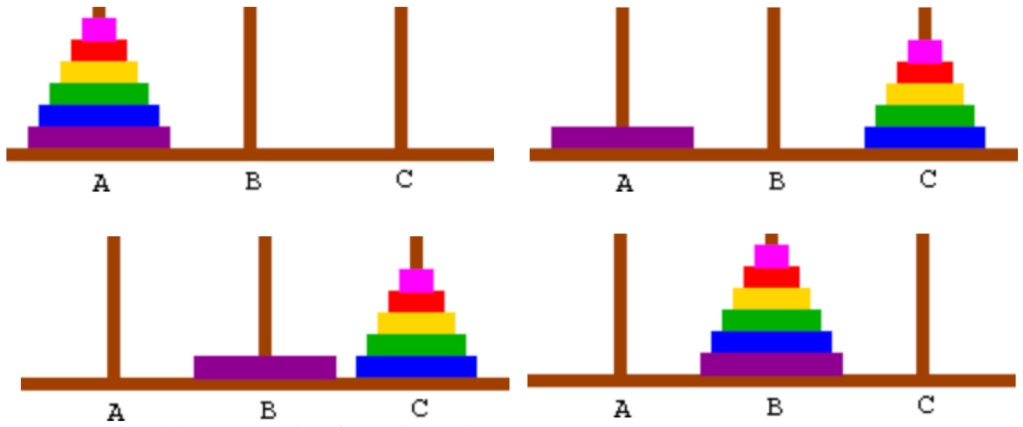
- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.



Tower of Hanoi

The objective of the puzzle is to move the entire stack from A to B, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.



Tower of Hanoi

```
def solve_hanoi(n , source, destination, auxiliary):
    if n==1:
        print "Move disk 1 from source",source,"to destination",destination
        return
    solve_hanoi(n-1, source, auxiliary, destination)
    print "Move disk",n,"from source",source,"to destination",destination
    solve_hanoi(n-1, auxiliary, destination, source)

solve_hanoi(6, 'A', 'B', 'C')
```

Time complexity analysis

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

=> $T(n) = 1 + 2 + \cdots + 2^k T(n-k)$ => $T(n) = O(2^n)$

All subsets of size k

From S={1, 2, …, n}, find all subsets of size k

- Brute-force way: find all subsets and check each subset to see whether it is of size k. The time complexity is at least $O(2^n)$ since S has 2^n subsets.
- Recursion: start from 1, for each i
  1. either include i and find subsets of size one smaller in {i+1, …, n}
  2. or do not include i and find subsets of the same size in {i+1, …,n}

  Two criteria to stop
  1. Subsets of size k found
  2. The remaining elements are not enough

```
def combinations(n, k):
    result = []
    def directed_combinations(offset, partial_combination):
        if len(partial_combination) == k:
            result.append(list(partial_combination))
            return

        num_remaining = k - len(partial_combination)

        # if the remaining elements are not enough, stop
        if num_remaining > n-offset+1:
            return
        else:
            directed_combinations(offset+1, partial_combination + [offset])
            directed_combinations(offset+1, partial_combination)

    directed_combinations(1, [])
    return result
```

All subsets of size k

It is tree recursion, **but** we do not grow the full tree because of the stop criteria

1. Subsets of size k found: $\binom{n}{k}$ such cases
2. The remaining elements are not enough: close to $\binom{n}{n-k+1}$ such cases

For each case that stops, we traverse no more than n nodes, hence we have time complexity: $O\left(n\binom{n}{k} + n\binom{n}{n-k+1}\right) = O\left(n\binom{n}{k}\right)$

When $k \ll n$, this is $O(n^{k+1})$ which is much smaller than $O(2^n)$

There is also a bit manipulation solution for people who are interested.

```python
def combinations(n, k):
    result = []
    def directed_combinations(offset, partial_combination):
        if len(partial_combination) == k:
            result.append(list(partial_combination))
            return

        num_remaining = k - len(partial_combination)

        # if the remaining elements are not enough, stop
        if num_remaining > n-offset+1:
            return
        else:
            directed_combinations(offset+1, partial_combination + [offset])
            directed_combinations(offset+1, partial_combination)

    directed_combinations(1, [])
    return result
```

Practice Problems

1. https://leetcode.com/problems/merge-two-sorted-lists/ (This can be helpful for your merge sort algorithm and can also remind you of the linked list data structure.)
2. https://leetcode.com/problems/validate-binary-search-tree/ (This can be helpful for your binary search algorithm and can also remind you of the binary search tree data structure.)
3. https://leetcode.com/problems/partition-to-k-equal-sum-subsets/
4. https://leetcode.com/problems/k-th-symbol-in-grammar/