Recursion

204113 Computer & Programming

Dr. Arnon Rungsawang
Dept. of computer engineering
Kasetsart University
https://mike.cpe.ku.ac.th/204113

Massive Information & Knowledge Engineering

Version 2024

Recursive Functions

Recursive Relationships

- Imagine that a CEO of a large company wants to know how many people work for him? One option is to spend a tremendous amount of personal effort counting the number of people on the payroll.
- However, the CEO has other more important things to do, and so implements another, cleverer, option.
  - At the next meeting with his department directors, he asks everyone to tell him at the next meeting how many people work for them.
  - Each director then meets with all their managers, who subsequently meet with their supervisors who perform the same task.
  - The supervisors know how many people work under them and readily report this information back to their managers (plus one to count themselves), who relay the aggregated information to the department directors, who relay the relevant information to the CEO.
- In this way, the CEO accomplishes a difficult task (for himself) by delegating similar, but simpler, tasks to his subordinates.

Recursive Functions

- A recursive function is a function that makes calls to itself.
  - It works like the loops we have seen before, but sometimes the situation is better to use recursion than loops.
- Every recursive function has two components: a base case and a recursive step.
  - The base case is usually the smallest input and has an easily verifiable solution. This is also the mechanism that stops the function from calling itself forever.
  - The recursive step is the set of all cases where a recursive call, or a function call to itself, is made. Note that each call has been done on a smaller side problem.

Factorial

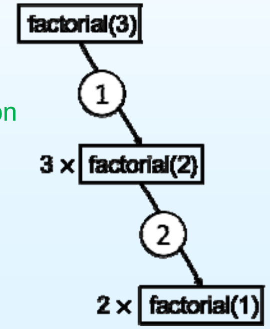
- As an example, we show how recursion can be used to define and compute the factorial of an integer number.
- The factorial of an integer n is $1 \times 2 \times 3 \times \cdots \times (n-1) \times n$.
- The recursive definition can be written as:

$$f(n) = \begin{cases} 1 & if\ n = 1 \\ n \times f(n-1) & \text{oterwise} \end{cases}$$

  - The base case is $n = 1$ which is trivial to compute $f(1) = 1$.
  - In the recursive step, n is multiplied by the result of recursive call to the factorial of $n - 1$.

Factorial (2)

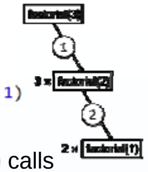
```python
def factorial(n):
  """Computes and returns the factorial of n,
  a positive integer.
  """
  if n == 1: # Base cases!
    return 1
  else: # Recursive step
    return n * factorial(n - 1) # Recursive call
```



- A recursion tree is a diagram of the function calls connected by numbered arrows to depict the order in which the calls were made.

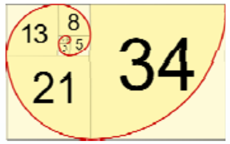
Factorial (3)

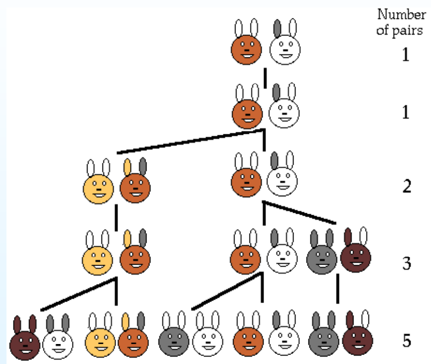
```python
if n == 1: # Base cases!
  return 1
else: # Recursive step
  return n * factorial(n - 1)
```



- When Python executes a function, it creates a workspace for the variables that are created in that function, and whenever a function calls another function, it will wait until that function returns an answer before continuing.
- In programming, that workspace is called stack.
  - A call is made to factorial(3), A new workspace_A is opened to compute factorial(3).
  - Input argument value 3 is compared to 1. Since they are not equal, else statement is executed.
  - 3*factorial(2) must be computed. A new workspace_B is opened to compute factorial(2).
  - Input argument value 2 is compared to 1. Since they are not equal, else statement is executed.
  - 2*factorial(1) must be computed. A new workspace_C is opened to compute factorial(1).
  - Input argument value 1 is compared to 1. Since they are equal, if statement is executed.
  - The return variable is assigned the value 1. factorial(1) terminates with output 1.
  - In workspace_B, 2*factorial(1) can be resolved to $2 \times 1=2$. Output is assigned the value 2. factorial(2) terminates with output 2.
  - In workspace_A, 3*factorial(2) can be resolved to $3 \times 2=6$. Output is assigned the value 6. factorial(3) terminates with output 6.

Fibonacci's Rabbits

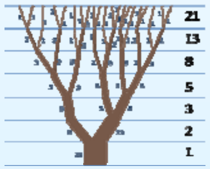




- The original problem that Fibonacci investigated (in the year 1202) was about how fast rabbits could breed in ideal circumstances.
- At the end of the first month, they mate, but there is still one only 1 pair.
- At the end of the second month the female produces a new pair, so now there are 2 pairs of rabbits in the field.
- At the end of the third month, the original female produces a second pair, making 3 pairs in all in the field.
- At the end of the fourth month, the original female has produced yet another new pair, the female born two months ago produces her first pair also, making 5 pairs.

Fibonacci (2)

- **Fibonacci numbers** were originally developed to model the idealized population growth of rabbits. Since then, they have been found to be significant in any **naturally occurring phenomena**.
- The Fibonacci numbers can be generated using the following **recursive formula**.
  - Note that the recursive step contains two recursive calls and that there are also two base cases (i.e., two cases that cause the recursion to stop).
- The recursive definition can be written as:



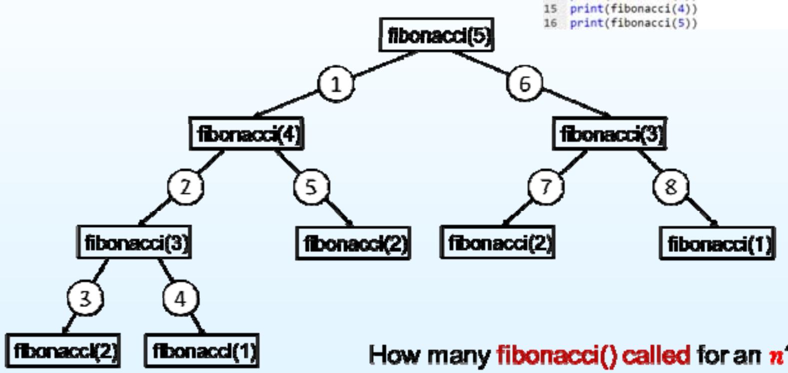
$$F(n) = \begin{cases} 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Fibonacci (3)

```python
def fibonacci(n):
  """Computes and returns the Fibonacci of n,
  a postive integer.
  """
  if n == 1: # first base case
    return 1
  elif n == 2: # second base case
    return 1
  else: # Recursive step
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(1))
print(fibonacci(2))
print(fibonacci(3))
print(fibonacci(4))
print(fibonacci(5))
```


Fibonacci (4)


```python
def fibonacci(n):
  """Computes and returns the Fibonacci of n,
  a postive integer.
  """
  if n == 1: # first base case
    return 1
  elif n == 2: # second base case
    return 1
  else: # Recursive step
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(1))
print(fibonacci(2))
print(fibonacci(3))
print(fibonacci(4))
print(fibonacci(5))
```

How many **fibonacci() called** for an n?

Fibonacci – Iterative implementation

- There is an iterative method of computing n^{th} Fibonacci numbers can be generated using the following **recursive formula**.

```python
import numpy as np

def iter_fib(n):
  fib = np.ones(n)

  for i in range(2, n):
    fib[i] = fib[i - 1] + fib[i - 2]

  return fib

def iter_fib2(n):
  fib = [1 for i in range(n)]
  for i in range(2, n):
    fib[i] = fib[i - 1] + fib[i - 2]
  return fib

def fibonacci(n):
  """Computes and returns the Fibonacci of n,
  a postive integer.
  """
```

Fibonacci – Iterative implementation (2)

```python
  if n == 1: # first base case
    return 1
  elif n == 2: # second base case
    return 1
  else: # Recursive step
    return fibonacci(n-1) + fibonacci(n-2)

def timeit(fn, n):
  import time
  # record start time
  start = time.time()
  #-----------------------------------
  fn(n)
  #-----------------------------------
  # record end time
  end = time.time()
  # print the difference between start
  # and end time in milli. secs
  print(f"The time of execution of above program is :",\
    f"{(end-start) * 10**3:.2f}", "ms")

timeit(fibonacci, 25)
```


Fibonacci – Computational time

```
[26]
timeit(fibonacci, 25)
timeit(iter_fib, 25)
timeit(iter_fib2, 25)

The time of execution of above program is : 34.38 ms
The time of execution of above program is : 0.10 ms
The time of execution of above program is : 0.01 ms

[22]
%timeit iter_fib(25)

13.1 µs ± 4.06 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)

[23]
%timeit iter_fib2(25)

4.68 µs ± 128 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

[24]
%timeit fibonacci(25)

22.3 ms ± 1.03 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

TIP! Try to write functions iteratively whenever it is convenient to do so.

Your functions will run faster.

Remark on Recursive Method

- In general, iterative functions are faster than recursive functions that perform the same task. So why use recursive functions at all?
- There are some solution methods that have a naturally recursive structure. In these cases, it is usually very hard to write a counterpart using loops.
- The primary value of writing recursive functions is that they can usually be written much more compactly than iterative functions.
- But the cost of the improved compactness is the added running time.

Remark on Recursive Method (2)

```
factorial(5000)

---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-29-4d2572cc43ba> in <cell line: 1>()
----> 1 factorial(5000)

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

<ipython-input-28-06798059af1f> in factorial(n)
      6     return 1
      7   else: # Recursive step
----> 8     return n * factorial(n - 1) # Recursive call

RecursionError: maximum recursion depth exceeded in comparison
```

```
[31]
import sys
sys.setrecursionlimit(10**5)
sys.set_int_max_str_digits(10000000)

factorial(5000)

422857792660554352220106420023358440539078667462664674884978240218135805
```

- When we are using recursive call as showing previously, we need to make sure that it can reach the base case, otherwise, it results to infinite recursion.
- In Python, when we execute a recursive function on a large output that can not reach the base case, we will encounter a `maximum recursion depth exceeded error`.
- We can handle the recursion limit using the `sys` module in Python and set a higher limit.

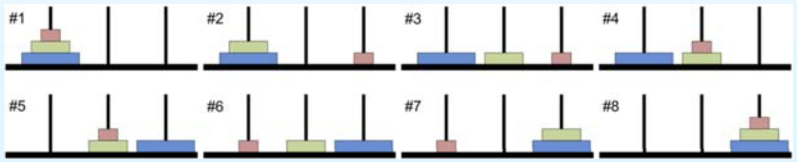
Divide and Conquer

Divide and Conquer

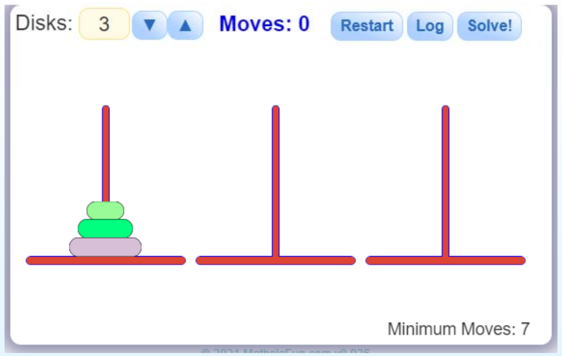
- Divide and conquer is a useful strategy for solving difficult problems.
- Using divide and conquer, difficult problems are solved from solutions to many similar easy problems.
  - In this way, difficult problems are broken up, so they are more manageable.

Towers of Hanoi

- The Towers of Hanoi problem consists of three vertical rods, or towers, and N disks of different sizes, each with a hole in the center so that the rod can slide through it.
- The disks are originally stacked on one of the towers in order of descending size (i.e., the largest disc is on the bottom).
- The goal of the problem is to move all the disks to a different rod while complying with the following three rules:
  - Only one disk can be moved at a time.
  - Only the disk at the top of a stack may be moved.
  - A disk may not be placed on top of a smaller disk.



Towers of Hanoi (2)



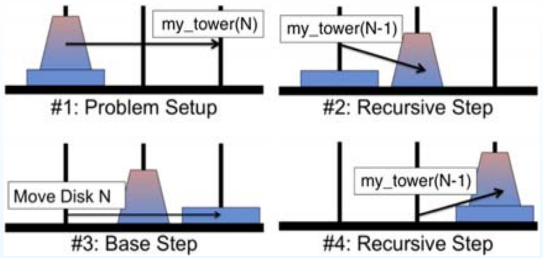
Credit: https://www.mathsisfun.com/

Towers of Hanoi (3)

- A legend goes that a group of Indian monks are in a monastery working to complete a Towers of Hanoi problem with 64 disks. And when they complete the problem, the world will end.
- Fortunately, the number of moves required is $2^{64} - 1$, so even if they could move one disk per millisecond, it would take over 584 million years for them to finish.
- The key to the Towers of Hanoi problem is breaking it down into smaller, easier-to-manage problems that we will refer to as subproblems.
  - For this problem, it is relatively easy to see that moving a disk is easy (which has only three rules) but moving a tower is difficult (we cannot immediately see how to do it).
- *Therefore, we will assign moving a stack of size N to several subproblems of moving a stack of size $N - 1$.*

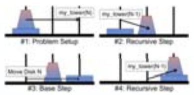
Towers of Hanoi (4)

- Consider a stack of `N` disks that we wish to move from Tower 1 to Tower 3 and let `my_tower(N)` move a stack of size `N` to the desired tower (i.e., display the moves).
- How to write `my_tower` may not immediately be clear. However, if we think about the problem in terms of subproblems, we can see that we need to move the top `N-1` disks to the middle tower, then the bottom disk to the right tower, and then the `N-1` disks on the middle tower to the right tower.



- `my_tower` can display the instruction to move disk `N`, and then make recursive calls to `my_tower(N-1)` to handle moving the smaller towers.
- The calls to `my_tower(N-1)` make recursive calls to `my_tower(N-2)` and so on.
- A breakdown of the three steps is depicted in the figure.

Towers of Hanoi (5)



```python
def my_towers(N, from_tower, to_tower, alt_tower, DEBUG=False):
    """
    Displays the moves required to move a tower of size N from the
    'from_tower' to the 'to_tower'.

    'from_tower', 'to_tower' and 'alt_tower' are uniquely either
    1, 2, or 3 referring to tower 1, tower 2, and tower 3.
    """

    if DEBUG:
        print(f'my_towers({N},{from_tower},{to_tower},{alt_tower})')

    if N != 0:
        # recursive call that moves N-1 stack from starting tower
        # to alternate tower
        my_towers(N-1, from_tower, alt_tower, to_tower, DEBUG)

        # display to screen movement of bottom disk from starting
        # tower to final tower
        print("Move disk %d from tower %d to tower %d."\
                %(N, from_tower, to_tower))

        # recursive call that moves N-1 stack from alternate tower
        # to final tower
        my_towers(N-1, alt_tower, to_tower, from_tower, DEBUG)

# my_towers(3, 1, 3, 2, True)
my_towers(3, 1, 3, 2)
```

- By using Divide and Conquer, we have solved the Towers of Hanoi problem by making recursive calls to slightly smaller Towers of Hanoi problems that, in turn, make recursive calls to yet smaller Towers of Hanoi problems.
- Together, the solutions form the solution to the whole problem. The actual work done by a single function call is actually quite small: two recursive calls and moving one disk.
- In other words, a function call does very little work (moving a disk), and then passes the rest of the work onto other calls, a skill you will probably find very useful throughout your engineering career.

Quicksort

- A list of numbers, A, is *sorted* if the elements are arranged in ascending or descending order.
- Although there are many ways of sorting a list, quicksort is a divide-and-conquer approach that is a very fast algorithm for sorting using a single processor (there are faster algorithms for multiple processors).
- The quicksort algorithm starts with the observation that sorting a list is hard, but comparison is easy. So instead of sorting a list, we separate the list by comparing to a pivot.
- At each recursive call to quicksort, the input list is divided into three parts:
  - elements that are smaller than the pivot,
  - elements that are equal to the pivot,
  - and elements that are larger than the pivot.
- Then a recursive call to quicksort is made on the two subproblems: the list of elements smaller than the pivot and the list of elements larger than the pivot.
- Eventually the subproblems are small enough (i.e., list size of length 1 or 0) that sorting the list is trivial.

Quicksort (2)

```python
def my_quicksort(lst, DEBUG=False):
  if DEBUG:
    print(f"DEBUG: myquick_sort({lst})")
  if len(lst) <= 1:
    # list of length 1 is easiest to sort
    # because it is already sorted
    sorted_list = lst
  else:
    # select pivot as the first element of the list
    pivot = lst[0]
    # initialize lists for bigger and smaller elements
    # as well those equal to the pivot
    bigger = []
    smaller = []
    same = []
    # loop through list and put elements into proper array
    for item in lst:
      if item > pivot:
        bigger.append(item)
      elif item < pivot:
        smaller.append(item)
      else:
        same.append(item)
    sorted_list = my_quicksort(smaller,DEBUG) + same\
                  + my_quicksort(bigger,DEBUG)
  return sorted_list

res = my_quicksort([2, 1, 3, 5, 6, 3, 8, 10], True)
# my_quicksort([2, 1, 3, 5, 6, 3, 8, 10])
print(res)
```

- Similar to Towers of Hanoi, we have broken up the problem of sorting (hard) into many comparisons (easy).

```python
[10] my_quicksort([2, 1, 3, 5, 6, 3, 8, 10], True)
     # my_quicksort([2, 1, 3, 5, 6, 3, 8, 10])
```

```
DEBUG: myquick_sort([2, 1, 3, 5, 6, 3, 8, 10])
DEBUG: myquick_sort([1])
DEBUG: myquick_sort([3, 5, 6, 3, 8, 10])
DEBUG: myquick_sort([])
DEBUG: myquick_sort([5, 6, 8, 10])
DEBUG: myquick_sort([])
DEBUG: myquick_sort([6, 8, 10])
DEBUG: myquick_sort([])
DEBUG: myquick_sort([8, 10])
DEBUG: myquick_sort([])
DEBUG: myquick_sort([10])
[1, 2, 3, 3, 5, 6, 8, 10]
```

```python
import random
#Generate 50 random numbers between 10 and 90
randomlist1 = random.sample(range(1, 90000), 500)
randomlist2 = randomlist1.copy()
%timeit my_quicksort(randomlist1)
%timeit sorted(randomlist2)
```

```
473 µs ± 58.8 µs per loop (mean ± std. dev. of 7 runs, 1000 lo
26 µs ± 6.84 µs per loop (mean ± std. dev. of 7 runs, 10000 lo
```


Sample Problem Solving

my_sum

- Write a function `my_sum(lst)` where `lst` is a list, and the output is the sum of all the elements of `lst`. You can use recursion or iteration to solve the problem, but do not use Python's function sum.

```python
def my_sum(lst, DEBUG=False):
  global count
  if DEBUG:
    print(f'{count}: Called my_sum({lst})')
    count += 1
  if len(lst)==0:
    return 0
  return lst[0]+my_sum(lst[1:],DEBUG)

count = 1
lst = [i for i in range(11) if i%2==0]
print(f'sum({lst})={my_sum(lst,True)}')
```

```
1: Called my_sum([0, 2, 4, 6, 8, 10])
2: Called my_sum([2, 4, 6, 8, 10])
3: Called my_sum([4, 6, 8, 10])
4: Called my_sum([6, 8, 10])
5: Called my_sum([8, 10])
6: Called my_sum([10])
7: Called my_sum([])
sum([0, 2, 4, 6, 8, 10])=30
```


Decimal ⇨ Binary

```python
def dec2bin(n):
    n = int(n)
    if n==0:
        return ''
    return str(dec2bin(n//2))+str(n%2)

def bin2dec(b):
    if len(b)==1:
        return int(b)
    return int(b[-1:]) + 2 * bin2dec(b[:-1])

n = 571
print(f'dec2bin({n})={dec2bin(n)}')
b = '1000111011'
print(f'bin2dec({b})={bin2dec(b)}')
```

```
dec2bin(571)=1000111011
bin2dec(1000111011)=571
```

my_GCD

- The greatest common divisor of two integers `a` and `b` is the largest integer that divides both numbers without remainder, and the function to compute it is denoted by `gcd(a,b)`.
- The `gcd` function can be written recursively. If `b` equals 0, then `a` is the greatest common divisor. Otherwise, `gcd(a,b) = gcd(b,a%b)` where `a%b` is the remainder of `a` divided by `b`. Assume that `a` and `b` are integers.
- Write a recursive function `my_gcd(a,b)` that computes the greatest common divisor of `a` and `b`. Assume that `a` and `b` are integers.

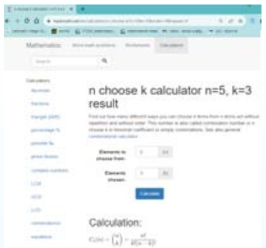
```python
def my_gcd(a, b, DEBUG=False):
    global count
    if DEBUG:
        print(f'{count}: Called my_gcd({a},{b})')
        count += 1
    if b==0:
        return a
    else:
        return my_gcd(b, a%b, DEBUG)

count = 1
a,b = 66,121
print(f'gcd({a},{b})={my_gcd(a,b,True)}')
```

```
1: Called my_gcd(66,121)
2: Called my_gcd(121,66)
3: Called my_gcd(66,55)
4: Called my_gcd(55,11)
5: Called my_gcd(11,0)
gcd(66,121)=11
```


n choose k

- A function, `C(n,k)`, which computes the number of different ways of uniquely choosing `k` objects from `n` without repetition, is commonly used in many statistics applications.
  - For example, how many three-flavored ice cream sundaes are there if there are 10 ice-cream flavors?
- To solve this problem, we would have to compute `C(10,3)`, the number of ways of choosing three unique ice-cream flavors from 10. The function `C` is commonly called `n choose k`. We may assume that `n` and `k` are integers.
  - If `n=k`, then clearly `C(n,k)=1` because there is only way to choose `n` objects from `n` objects.
  - If `k=1`, then `C(n,k)=n` because choosing each of the `n` objects is a way of choosing one object from `n`.
  - For all other cases, `C(n,k)=C(n-1,k)+C(n-1,k-1)`. Can you see, why?
- Write a function `n_choose_k(n,k)` that computes the number of times `k` objects can be uniquely chosen from `n` objects without repetition.

n choose k (2)




```python
def n_choose_k(n, k, DEBUG=False):
    global count
    if DEBUG:
        print(f'{count}: Called n_choose_k({n},{k})')
        count += 1
    if n==k:
        return 1
    if k==1:
        return n
    return n_choose_k(n-1,k,DEBUG)\
        + n_choose_k(n-1,k-1,DEBUG)

count = 1
n,k = 5,3
print(f'n_choose_k({n},{k})={n_choose_k(n,k,True)}')
```

```
1: Called n_choose_k(5,3)
2: Called n_choose_k(4,3)
3: Called n_choose_k(3,3)
4: Called n_choose_k(3,2)
5: Called n_choose_k(2,2)
6: Called n_choose_k(2,1)
7: Called n_choose_k(4,2)
8: Called n_choose_k(3,2)
9: Called n_choose_k(2,2)
10: Called n_choose_k(2,1)
11: Called n_choose_k(3,1)
n_choose_k(5,3)=10
```


All Combinations

```python
def allComb(lst, DEBUG=False):
    global count
    if DEBUG:
        print(f'{count}: Called allComb({lst})')
        count += 1
    if len(lst)==0:
        return [[]]
    # recursive case: find smaller combin
    smallerCmb = allComb(lst[1:],DEBUG)
    # pair lst[0] with each elm in smallerCmb
    allCmb = [[lst[0]] + x for x in smallerCmb]
    # return allCmd + the rest of smallerCmd
    return allCmb + smallerCmb

count = 1
m = [1,5,7]
res = allComb(m, True)
print(f'allComb({m}):\n{res}')
```

```
1: Called allComb([1, 5, 7])
2: Called allComb([5, 7])
3: Called allComb([7])
4: Called allComb([])
allComb([1, 5, 7]):
[[1, 5, 7], [1, 5], [1, 7], [1], [5, 7], [5], [7], []]
```

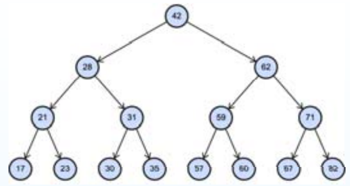
my_change

```python
def my_change(cost=1234, paid=2000, DEBUG=False):
  global count, notes
  if DEBUG:
    print(f'{count}: Called my_change({cost},{paid})')
    count += 1
  ##---
  res = [] # to keep the list of changed note(s)
  if paid < cost:
    e = f"Amount ({paid}) paid is less than ({cost}) cost!!"
    raise Exception(e)
  paid -= cost
  if paid==0: # BASE CASE, no amount to change
    return []
  for b in notes: # RECURSIVE CASE
    if paid >= b:
      paid -= b
      res.append(b)
      if DEBUG:
        print(res)
      res += my_change(0,paid,DEBUG)
      break # out of loop after changing this b note
  return res

count = 1
notes = [1000,500,100,50,20,10,5,2,1]
#notes = [500,100,50,20,10,5,2,1]
res = my_change(1117,4000,True)
#res = my_change(1117,4000)
print(res)
```

- Use recursion to program a function `my_change(cost, paid)` where cost is the cost of the item, paid is the amount paid.
- Output change is a list of bills and coins that should be returned to the seller.

Binary Search Tree

```python
from random import randint

def rand(a=0, b=100):
  return randint(a, b)

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

## insert a node
def insert(node, val):
  # return a new node if tree is empty
  if node is None:
    return Node(val)
  # traverse to the right and insert
  if val < node.val:
    node.left = insert(node.left, val)
  else:
    node.right = insert(node.right, val)
  return node
```

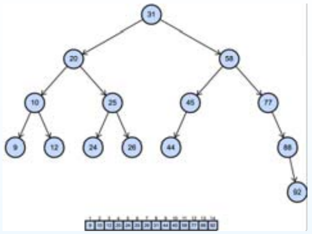


- In a Binary Search Tree (BST). Every circle is called a node, and each node can be connected to 2 other nodes -- one on the left and right.
- That's why they're called Binary Trees, they have 2 child nodes, and it looks like a tree!

31,20,58,10,25,45,77,9,12,24,26,44,88,92

Binary Search Tree (2)

```python
## inorder traversal
def inorder(node):
  if node is not None:
    # traverse left
    inorder(node.left)
    # traverse root
    print(str(node.val), end=' ')
    # traverse right
    inorder(node.right)

## post-order traversal
def post_order(node):
  if node is not None:
    post_order(node.left)
    post_order(node.right)
    print(str(node.val), end=' ')
```

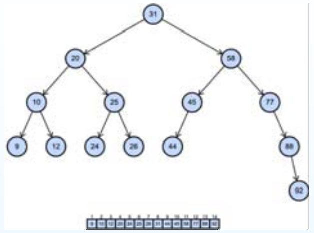


- The left child node is always smaller or equal in value than the parent, and the right is always greater or equal.
- Some implementations allow only the left or right node to be equal to the parent.

REF: 4 Ways To Traverse Binary Trees

Binary Search Tree (3)

```python
## search
def search(root, number):
  if root==None:
    return None
  if root.val==number:
    return root.val
  if number < root.val and root.left is not None:
    return search(root.left, number)
  if root.right is not None:
    return search(root.right, number)
  return None

### main begin here
#lst = [rand() for i in range(10)]
lst = [31,20,58,10,25,45,77,9,12,24,26,44,88,92]
print(lst)

tree = None
for elm in lst:
  tree = insert(tree, elm)

print("Inorder traverse:", end=' ')
inorder(tree)
print()
print(search(tree, 45), search(tree, 32))
```

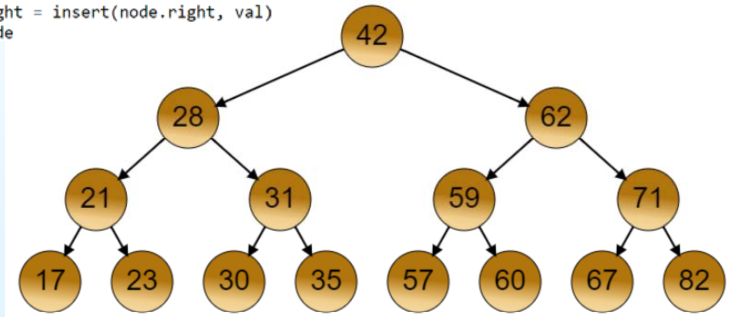


42,28,62,21,31,59,71,17,23,30,35,57,60,67,82

Binary Search Tree (4)

```
## insert a node
def insert(node, val):
  # return a new node if tree is empty
  if node is None:
    return Node(val)
  # traverse to the right and insert
  if val < node.val:
    node.left = insert(node.left, val)
  else:
    node.right = insert(node.right, val)
  return node
```

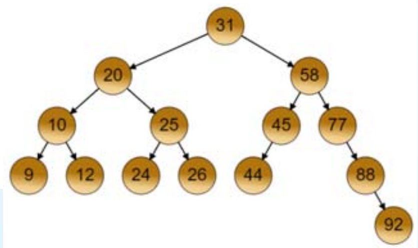
```
lst = [42,28,62,21,31,59,71,17,23,30,35,57,60,67,82]
print(lst)

tree = None
for elm in lst:
  tree = insert(tree, elm)
```



Binary Search Tree (5)

```
def printTree(node, level=0):
    if node != None:
        printTree(node.right, level + 1)
        print(' ' * 4 * level + '-> ' + str(node.val))
        printTree(node.left, level + 1)

### main begin here
#lst = [rand() for i in range(10)]
lst = [31,20,58,10,25,45,77,9,12,24,26,44,88,92]
print(lst)

tree = None
for elm in lst:
  tree = insert(tree, elm)

print("Inorder traverse:", end=' ')
inorder(tree)
print()
print(search(tree, 45), search(tree, 32))
print()
printTree(tree)
```

```
                -> 92
            -> 88
        -> 77
    -> 58
        -> 45
            -> 44
-> 31
            -> 26
        -> 25
            -> 24
    -> 20
            -> 12
        -> 10
            -> 9
```



To be continue..

つづく