

ORDER OF GROWTH

1 < logn < √n < n < nlogn < < < < …. <

from homework

ASYMPTOTIC NOTATIONS:

O – Upper Bound, includes class and everything larger than it f(n) <= something

Ω - Lower Bound, includes class specified and everything smaller something <= f(n)

Θ – Average bound, should only be one, where bigoh and theta overlap. TIGHTEST

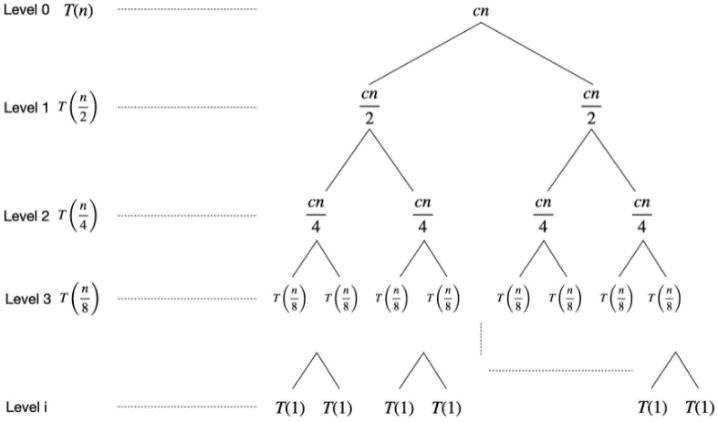
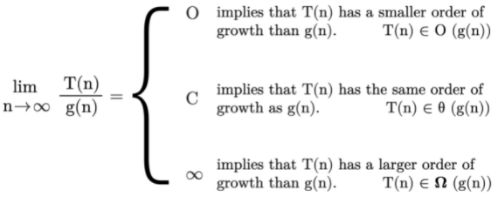
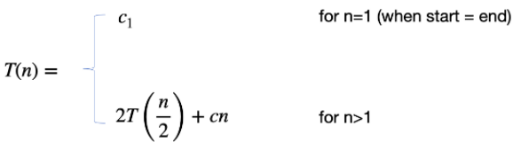
**| = loga - logb | |**

**RECURSION / RECURSIVE RELATIONS**

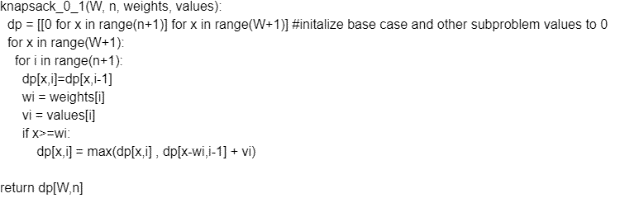
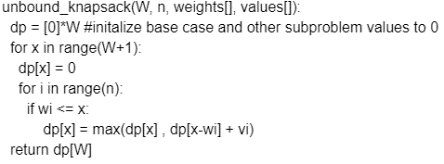
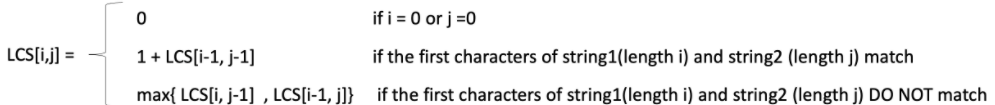
Towers of Hanoi = T(n) = T(n-1) + 1 + T(n-1) – first is time taken to move n-1 disks from first peg to second peg. 1 is time to move last disk to last peg. Last is time taken to move the n-1 disks from the second peg to the third peg.

Substitution: T(n) = T(n-1) + 1 | T(n-1) = T(n-1-1) +1 = T(n-2) + 2 | T(n-2) = T(n-2-1) +1 = T(n-3) + 3 | T(n-k) = T(n-k) + k

At the kth equation we reach the base case. In base case we can find what k is equal to. Once we have k we can substitute it in the base equation.



**DIVIDE AND CONQUOR:** divide a problem into smaller subproblems and then solve them. 3 steps – Divide (break problems into smaller subproblems) Conquer (solve the subproblems) Combine – combine subproblem solutions to get solution to larger problem.

DYNAMIC PROGRAMMING: top-down we start with the bigger problem and go down to the base case. As we go down we store the solution of the subproblems in extra memory to avoid recomputation. Bottom-Up we start with the base case and solve bigger problems as we progress until we reach the actual problem. Top-Down mostly recursive, bottom up mostly iterative. SHOULD HAVE OVERLAPING SUBPROBLEMS AND OPTIMAL **SUBSTRUCTURE**

GREEDY:

activity\_selection(activities, start\_times,end\_times):

  result = []

  blocked\_time = 0

  for i := 1 to len(activities):

    if(start\_times[i] >= blocked\_time):

      result.append(activities[i])

      blocked\_time = end\_times[i]

return result

space and time are O(n)

Backtracking: when we want to generate all possible answers or we want a valid solution, not an optimal one. Find the power set of a given set, find solution to sudoku puzzle, find solution to maze. Power Set Psudo: def powerset\_helper(pointer, choices\_made,input, result):  
    if (pointer < 0)):  
        append choices\_made to results # make a deep copy since we are working with objects  
        return  
    append input[pointer] to choices\_made  
    powerset\_helper(pointer-1, choices\_made, input, result)  
    #backtracking  
    remove last element added to choices\_made  
    powerset\_helper(pointer - 1, choices\_made, input, result)  
def powerset(input):  
    result = []  
    powerset\_helper(len(input)-1, [], input, result)  
    return result

Time & space complexity for both is O(nW)