Asymptotic Notation:

- Big O when g(n) is worse than f(n), aka worst case
  - f(n) is O(g(n)) if there exist positive constant C and n0 such that 0 <= f(n) <= Cg(n) for all n >= n0
- Little O is the same as Big O but if for all C, so there's no case where g(n) < f(n)
- Big Ω when g(n) is better than f(n), aka best case
  - f(n) is said to be Ω(g(n)) if there exists positive constant C and (n0) such that 0 <= Cg(n) <= f(n) for all n >= n0
- Big Θ when g(n) is equal to f(n)
  - f(n) is said to be Θ(g(n)) if f(n) is O(g(n)) and f(n) is Ω(g(n)
  - 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0

Big O Rules:

- Constants can be ignored, Ex: In $3n^3+30$, ignore 30
- Lower order terms can be dropped^3
  Ex: In $3n^3+n^2+n+3$, only look at 3n
- Smaller exponents are big O of larger exponents, Ex: $n^2 = O(n^3)$
- Any log is Big O of any polynomial
  - Ex: $\log_2 n = O(n^{938490})$
- Any polynomial is Big O of any exponential
  Ex: $n^3 = O(b^{32094890324890})$

Runtime Order:
n, n $\log_2$ n, n^2, n^3.. 1.5^n, 2^n, n

Master Theorem:
T(n) = aT(n/b) + f(n)
n = size of input
a = number of subproblems
n/b = size of each subproblem
f(n) = cost of work done outside of the recursive call

Three Master Thereom Cases:

1. If $f(n) = O(n^{\log\_b(a-e)})$ for some e > 0 ,then
$T(n) = \Omega(n^{\log\_b(a)})$
2. If $f(n) = \Omega(n^{\log\_b(a)})$, then
$T(n) = \Omega(n^{\log\_b(a)} \log n)$
3. If $f(n) = \Theta(n^{\log\_b(a+e)})$ for some e > 0 and af(n/b) <= cf(n) for some c<1, then
$T(n) = \Omega(f(n))$

Divide and Conquer Paradigm: Base Case, Divide Into Smaller Parts, Recurse, Combine Solutions, Prove by Induction

Mergesort: Divide by half until can not be divided, then return by ordering same way
Ex: 38473 splits to 384 73 splits to 38 4 7 3 splits to 3 8 4 7 3, restarts to 38 4 73 then 348 37 then to 33478

Binary Search: when searching for 2, start in the middle and comparing halves until you get there
Ex: searching for 3 in 012345679, compare 3 to 5, 3<4 so search 0123, 3 > 2 so only option is 3

Merging

```
Merge(L,R):
  Let n ← len(L) + len(R)
  Let A be an array of length n
  j ← 1, k ← 1

  For i = 1,…,n:
    If (j > len(L)):              // L is empty
      A[i] ← R[k], k ← k+1
    ElseIf (k > len(R)):          // R is empty
      A[i] ← L[j], j ← j+1
    ElseIf (L[j] <= R[k]):        // L is smallest
      A[i] ← L[j], j ← j+1
    Else:                         // R is smallest
      A[i] ← R[k], k ← k+1

  Return A
```

Mergesort

```
MergeSort(A):
  If (len(A) = 1): Return A     // Base Case

  Let m ← ⌊len(A)/2⌋            // Split
  Let L ← A[1:m], R ← A[m+1:n]

  Let L ← MergeSort(L)          // Recurse
  Let R ← MergeSort(R)

  Let A ← Merge(L,R)            // Merge

  Return A
```

Karatsuba's Algorithm
$x = 10^{n/2}a + b$, $y = 10^{n/2}c + d$ ← Where n = max length of the integers that are being multiplied
Compute (a+b)(c+d) - (bd) - (ac) then compute 10^n * ac * $10^{n/2}$ * above answer + bd
Ex: 5678 x 1234, n=4 because 4 digits so n/2 = 2
a=56 b = 78 c=12 d=34
$x = 10^2(56) + 78$ $y = 10^2(12) + 34$
ac = 672 bd = 2652 (a+b)(c+d) = 6164
6164-2652-672 = 2840, (10^4 * 2652) + (10^2 * 2840) + 2652 = 7006652

Karatsuba's Algorithm

```
Karatsuba(u,v,n):
  If (n = 1): Return u·v                           // Base Case   constant

  Let m ← ⌈n/2⌉                                    // Split   linear
  Write u = 10^m · a + b,  v = 10^m · c + d

  Let e ← Karatsuba(a,c,m)      e = ac             // Recurse
      f ← Karatsuba(b,d,m)      f = bd             3·T(n/2)
      g ← Karatsuba(b-a,c-d,m)  g = (b-a)(c-d)

  Return 10^(2m) · e + 10^m · (e + f + g) + f      // Merge   linear
```

$\underbrace{e}_{ac}$, $\underbrace{(e+f+g)}_{= ad+bc}$, $\underbrace{f}_{bd}$

$T(n) = 3\,T(\frac{n}{2}) + C_n$

Median Algorithm: MOMSelect

$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + C_n$

```
MOM(A[1:n]):
  Let m ← ⌈n/5⌉
  For i = 1,…,m:
    Meds[i] = median{A[5i-4],A[5i-3],…,A[5i]}
    Let p ← MOMSelect(Meds[1:m],⌊m/2⌋)
```

```
MOMSelect(A[1:n],k):
  If(n ≤ 25): sort & return A[k]

  Let p = MOM(A)
  Partition around the pivot, let p = A[r]

  If(k = r): return A[r]
  ElseIf(k < r): return MOMSelect(A[1:r-1],k)
  ElseIf(k > r): return MOMSelect(A[r+1:n],k-r)
```

Mean Of Median Selection:
Divide elements into groups of 5, find median in each group, find median of n/5 medians
T(n) <= T(n/5) + T(7n/10) + n
T(n) = Θ(n)

Dynamic Programming Recipe: Identify Set of Subproblems, Make a Recurrence, Find an Algorithm, Reconstruct Solution

Weighted Interval Scheduling:
Given organized lines and weights, pick non-overlapping lines with the highest possible weight

Interval Scheduling: Top Down

Memoization

```
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← v1
FindOPT(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← max{FindOPT(n-1), vn + FindOPT(p(n))}
    return M[n]
```

Filling the Knapsack

```
// All inputs are global vars
// M[0:n,0:T] contains solutions to subproblems
FindSol(M,n,T):
  if (n = 0 or T = 0): return Ø       //base case
  else:
    if (wn > T): return FindSol(M,n-1,T)
    else:
      if (M[n-1,T] > vn + M[n-1,T-wn]):
        return FindSol(M,n-1,T)
      else:
        return {n} + FindSol(M,n-1,T-wn)
```

figure out which of two cases

Knapsack:
Given items with a value and weight and a backpack with a max weight, find items that have the highest value while still being able to fit inside of the backpack

Segmented Least Squares:
Finding segmented line of best fit that changes a little as possible while hitting all the points

SLS: Take II ("Top-Down")

```
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← C, M[2] ← C
FindOPT(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← min_{1≤i≤n} (ε_{i,n} + C + FindOPT(i − 1))
    return M[n]
```

Have to fill $O(n)$ elements

To fill *M[j]* we make $O(n)$ rec. calls

Total # of rec. calls: $O(n^2)$ **Total runtime:** $O(n^2)$

Solving the Recurrence: Bottom-Up

```
// All inputs are global vars
FindOPT(n):
  M[1] ← 1

  for (j = 2,…,n):
    M[j] = 1 + max_{1≤i<j and xi<xj} M[i]     ← O(n)

  return max_{1≤j≤n} M[j]    ← O(n)
```

TOTAL: $O(n^2)$

$O(n)$ loops
each loop → $O(n)$

Longest Increasing Subsequence:
Finding the numbers that increase with every number ex: 3843567 would be 34567

Solving the Recurrence: Bottom-Up

```
// All inputs are global vars
FindOPT(x,y,n,m):
  M[i,0] ← 0,   M[0,j] ← 0

  for (i= 1,…,n):
    for (j = 1,…,m):        ← O(nm) loops
      if (xi = yj):
        M[i,j] ← 1 + M[i-1,j-1]           O(1) time
      else:
        M[i,j] ← max{M[i-1,j],M[i,j-1]}

  return M[n,m]
```

Time: $O(nm)$ Space: $O(nm)$
↳ DP table