$\log n = O(n^a)$ $\sum_{i=1}^{n}(i+1) = \Theta(n^2) = \frac{(n+1)(n+2)}{2}$

Divide + Conquer Strategy: 1. Find simple base case. 2. Find way to reduce problem to that base case.

$\log_b x = y \leftrightarrow b^y = x$
$\log_b 1 = 0$
$\log_b b^n = n$
$b^{\log_b n} = n$
$\log_b(m \cdot n) = \log_b m + \log_b n$
$\log_b(\frac{m}{n}) = \log_b m - \log_b n$
$\log_b x^n = n \cdot \log_b x$

Insertion Sort

$O(n^2)$ $T(n) = T(n-1) + O(n)$

```
for j=2 to A.length
  key=A[j]
  i=j-1
  while i>0 and A[i]>key
    A[i+1]=A[i]
    i=i-1
  A[i+1]=key
```

Array is unsorted on the left and sorted on the right. Each item is shifted into correct position in sorted portion. Finishes when last item shifted.

Merge Sort $\Theta(n \log n)$

$T(n) = 2T(\frac{n}{2}) + O(n)$

```
Sort(A, p, r)
  if (p<r)
    q=(p+r)/2
    Sort(A,p,q)
    Sort(A, q+1, r)
    merge(A, p, q, r)
merge(A, p, q, r)
  B[p...r]
  i=k=p
  j=q+1
  while (i<=q and j<=r)
    if (A[i]<=A[j])
      B[k++]=A[i++]
    else B[k++]=A[j++]
  while (i<=q)
    B[k++]=A[i++]
  while (j<=r)
    B[k++]=A[j++]
  for i=p to r
    A[i]=B[i]
```

Recursively breaks an array in half until each sub-array is one element. Then merges the resulting sub-arrays by always adding the smaller item to a temporary array, then copying over.

Binary Search

$O(\log n)$ $T(n) = T(\frac{n}{2}) + O(1)$

```
if n==null or val==n.key
  return n
if val < n.key
  return func(n.left, v)
else
  return func(n.right, v)
```

Begins at root of tree and traces a path down. If key at current node equals value sought, return. Same if node is null. Otherwise make recursive call to L/R

Knapsack $O(n \cdot W)$

```
int knapsack(n, W, wt[], val[])
for i=0 to n
  for w=0 to W
    if i==0 or w==0, K[i][w]=0
    else if wt[i-1]<=w
      K[i][w]=max(
        val[i-1]+K[i-1][w-wt[i-1]],
        K[i-1][w])
    else, K[i][w]=K[i-1][w]
return K[n][W]
```

Iterates over all items and weights. At each turn, decides whether or not to take item. If item will fit, take max of leaving item out (taking square above) or taking item and using any remaining space (if 2 lb. left over, go up a row and use value at 2 lb.). Answer is in K[n][W] (last square in table).

Coin Changing $O(d \cdot A)$

Very similar to knapsack above, main difference is that for each cell we are calculating min like so:

$T[r][c] = \min(T[r-1][c], T[r][c-V_r]+1)$

answer in end; row above (not taking coin); taking coin; difference between current column and value of the r^{th} coin.

Pipe Cutting $O(i \cdot T)$

T = Total pipe length; i = Number of individual pipe lengths

Basically same as coin changing and knapsack. Each cell in table is calculated like so:

$T[r][c] = \max(T[r-1][c], T[r][c-l_r]+V_r)$

l_r = length of pipe at row; V_r = value of pipe at row

Longest Common Subsequence $\Theta(m \cdot n)$

```
m=X.length
n=Y.length
new tables b[1..m,1..n], c[0..n,0..m]
for i=1 to m
  for j=1 to n
    if x_i == x_j
      c[i,j]=c[i-1,j-1]+1
      b[i,j]="↖"
    else if c[i-1,j] >= c[i,j-1]
      c[i,j]=c[i-1,j]
      b[i,j]="↑"
    else
      c[i,j]=c[i,j-1]
      b[i,j]="←"
return c,b
```

all initialized to 0 (values, length)

Populates 2D array with greater and greater values. If chars in sub-seq. match, adds one to up left value, otherwise takes greater of left and up. Bottom right

```
printLCS(b, X, x.length, Y.length)
  if(i==0 or j==0) return
  if(b[i,j]=="↖")
    printLCS(b,x,i-1,j-1)
    print X_i
  else if b[i,j]=="↑"
    printLCS(b,x,i-1,j)
  else if b[i,j]=="←"
    printLCS(b,x,i,j-1)
```

answer. PrintLCS uses "↖" to print chars.

Longest Palindromic Sub.

```
if input[i]==input[j]
  T[i][j]=T[i+1][j-1]+2
else
  T[i][j]=max(T[i+1][j], T[i][j-1])
```

$O(n^2)$ Computes and saves longest from i to j and saves in T[i][j]. Final answer at T[0][input.length-1].

Constant:	$O(1)$
Logarithmic:	$O(\log n)$
Linear:	$O(n)$
Quadratic:	$O(n^2)$
Cubic:	$O(n^3)$
Polynomial:	$O(n^k)$ $k>0$
Exponential:	$O(k^n)$ $k>1$
Factorial:	$O(n!)$

Slower

Recurrence Methods

$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n>1 \end{cases}$

merge sort for example; base case; # of sub problems; sub problem size; work dividing and conquering outside recursion

$T(n) = 1 \cdot T(\frac{n}{2}) + \Theta(1)$ binary search example

Master Method: $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ $a \geq 1$, $b \geq 1$

- $n^{\log_b a} > f(n) \rightarrow T(n) = \Theta(n^{\log_b a})$ leaves dominate
- " $= f(n) \rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$ equal
- " $< f(n) \rightarrow T(n) = \Theta(f(n))$ function dominates

Reg. condition: $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$, $c < 1$

Muster Method: $T(n) = a \cdot T(n-b) + O(n^d)$, $b > 0$, $d \geq 0$

- $a < 1 \rightarrow T(n) = \Theta(n^d)$
- " $= 1 \rightarrow T(n) = \Theta(n^{d+1})$
- " $> 1 \rightarrow T(n) = \Theta(n^d \cdot a^{\frac{n}{b}})$

Asymptotic Properties:

Reflexivity: $f(n) = \Theta(f(n))$

Symmetry: $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

Transpose Symmetry: $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

Transitivity: If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n))$

Hotel Stopping $\Theta(n^2)$

```
S[i]= min. total penalty for stop @ j
S[0]=0
for i=1 to n
  S[i]=∞
  for j=0 to i
    S[i]=min(S[i],
      S[j]+(200-(a_i-a_j))^2)
return S[n]
```

Two loops both iterating over same array of best values, updating with best value on each pass. (n = number of hotels; current best for i; possible improvement by stopping at j)

Canoe Rental

n = # of trade posts $\Theta(n^2)$

```
OPT[1]=0, P[1]=0
for i=2 to n
  min ← R[1,i]
  P[i] ← 1
  for k=2 to i-1
    if OPT[k]+R[k,i] < min
      min=OPT[k]+R[k,i]
      P[i]=k
  OPT[i]=min
return P[]
```

P[] contains final list of stops. Assumes that best option at each point is to not stop. Then updates, checking and updating with nested for loop.

Longest Increasing Subsequence

$O(n^2)$ $T(n) = T(n-1) + O(n)$

```
if arr[j] < arr[i]
  res[i]=max(res[i], res[j]+1)
```

Nested loops on array, always updating with max possible value.

Common Recurrences

$2T(n-1)+1 = T(2^n)$
$T(n-1)+1 = T(n)$
$T(n-1)+n = \Theta(n^2)$
$T(\frac{n}{2})+c = \Theta(\log n)$
$T(\frac{n}{2})+n = \Theta(n)$
$2T(\frac{n}{2})+1 = \Theta(n)$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \rightarrow f(n) \in O(g(n))$, $c > 0 \rightarrow f(n) \in \Theta(g(n))$, $\infty \rightarrow f(n) \in \Omega(g(n))$

Greedy Clue:

Uniform or unit-length amounts (each job takes 1 minute).

Greedy Scheduling w/ Penalties $O(n^2)$

Sort by pen., decreasing. Schedule as late as possible before deadline. If no space available before deadline, schedule first available from end of array. NOT dominated by sorting. $O(n^2)$ to find a spot

Dynamic Programming

- At each step, choice is made based on solutions of sub-problems.
- Sub-problems are solved first.
- Bottom-up approach.
- Slower and more complex.

Greedy Algorithms

- At each step, choice is made based on what currently looks best. Locally optimal (greedy)
- Greedy choice made first.
- Top-down approach.
- Faster, simpler, may not give correct answer.

D.P. Properties

Optimal sub-structure: The solution to the problem contains solutions to sub-problems.

Over-lapping: Revisits same problems repeatedly: Fibonacci, factorial, etc.

Greedy Properties

Optimal Substructure: The same as for D.P.

Greedy Choice: Making greedy choice at every step still results in optimal solution. Earlier choices never need to be reconsidered.

3-SAT → 4-SAT → Independent Set → Vertex Cover → Long-Path
3-SAT → Clique
3-SAT → Dir-Ham-Cycle → Ham-Cycle → Ham-Path
Ham-Cycle → TSP
3-SAT → 3-Color → 4-Color → Course Time
→ Knapsack → IP-Decision
→ Set Partition

Huffman Coding $O(n \cdot \log n)$

1. Rank letters by frequency.
2. Form min heap from letters with internal nodes being sums of children.
3. To encode, decode, traverse tree. Left is 0, right is 1. Stop at a letter.

A 5
B 2
R 2
A
C 1
A
D 1
A
B
R
A

A: 0
B: 100
C: 1010
D: 1011
R: 11

Kruskal's MST $O(E \cdot \log E)$

for each vertex v in G
  make empty set out of v
sort edges of G ascending
for each edge u to v
  if u and v in different sets
    add (u,v) to T
    join u and v into set
return T

Sort all edges in ascending order by weight. Pick the smallest edge. If forms cycle with already chosen edges, discard. Else, include. Repeat until there are V-1 edges in spanning tree.

Dijkstra's Shortest Path $O(n^2)$

for each vertex v in G
  dist[v] = ∞
  prev[v] = ?
dist[src] = 0
Q = all v in G
while Q !empty
  u = v in Q w/ smallest dist[]
  remove u from Q
  for each neighbor v of u
    alt = dist[u] + dist_btwn(u,v)
    if alt < dist[v]
      dist[v] = alt
      prev[v] = u
return prev[]

Start dist. to all vert. at ∞. Dist. to start vert is permanent, others are temporary. Set start vertex as active. Calc dist from active vert. to all other accesible verts. by summing dist with weight of edges. If calculated distance is smaller, update. Set vertex with minimal temp distance as active. Mark its dist as permanent. Repeat until no permanent verts have neighbors with temp distance.

BFS $O(V \cdot E)$

unmark all vertices
choose start vertex x, mark
list L = x
tree T = x
while L !empty
  get vertex v from list front
  visit v
  for each unmark neigh. w
    mark w, add to end of list
    add edge vw to T

Starts at vertex and explores the neighbor nodes first, then moves out to next level of neighbors, and so on till all visited.

DFS $O(V \cdot E)$

DFS(G, v)
label v as visited
for all edges from v→w in G.adjacentEdges(v) do:
  if vertex w is not visited
    call DFS(G, w)

Starts at a root v and explores as far as possible before backtracking.

Fractional Knapsack $O(n \cdot \log n)$

sort by density, descending
while i ≤ n and weight < W
  if weight + W[i] ≤ W
    x[i] = 1
  else x[i] = (W - weight)/W[i]
  weight = weight + x[i] · w[i]
  i++

current item; total items; capacity; take all!; percent to take; current weight; weight of individual item

Take as much of highest value item as possible, then as much of next highest, and so on. Array x[i] contains fraction to take of each item i.

Greedy Coin Change

$O(n \cdot \log n)$ ← for sorting
Sort from highest value → lowest
for i=0 to denom.length
  while V ≥ denom[i]
    V = V - denom[i]
    ans.push(denom[i])

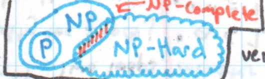
current denom; value to make; answer

Greedy Scheduling $O(n \log n)$

Sort F[] by earliest finish time and permute S[], start times, to match
count = 1, T[count] = 1
for i=2 to n
  if S[i] > F[T[count]]
    count++
    T[count] = i
return T[1...count]

Sorts the input so that the event with earliest finish time is first. Adds this event, then looks for the next event that starts after the previously-added event finishes.

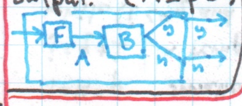
NP-Completeness



If A reduces to B, then A is no harder to solve than B. ($A \leq_p B$). Prereqs:
1. Input for A can be converted to input for B in polynomial time.
2. A given input must have same output for both A and B.

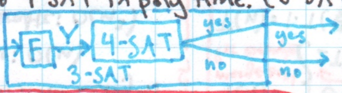
Proving NP-Complete:
1. Prove that problem is reducible to known NP-Complete problem. (NP-Hard)
2. Prove that a given solution can be verified in polynomialtime. (NP-Complete)

Example: A: Given a set of booleans, is at least one true? B: Given a set of integers, is their sum positive? ('A is known NP-Complete). Transform by setting true in A to 1 and false to 0, then check if sum is positive. A and B have same output. ($A \leq_p B$)



Example: Prove 4-SAT is NP-Complete

1. Show that 4-SAT can be verified in polynomial time (which means that it's NP). Set 4-SAT instance and proposed truth assignments can be verified in polynomial time.
2. Show that a known NP-Complete problem can be reduced to 4-SAT in poly time. ($3\text{-SAT} \leq_p 4\text{-SAT}$).



$X = (\bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_3) \wedge (X_1 \vee X_2 \vee X_3)$
$Y = (\bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_3 \vee H) \wedge (X_1 \vee X_2 \vee X_3 \vee H) \wedge (X_1 \vee X_2 \vee X_3 \vee \bar{H}) \wedge (\bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_3 \vee \bar{H})$

Linear Programming:

1. Define decision variables.
2. Write objective function equation (min, max)
3. Write each constraint equation
4. Graph and determine vertices of feasibility.

Macrosoft Example:
X_t = full time employees starting @ shift t
Y_t = part-time employees " "
Objective: min cost of full-time shift $(X_1 + \ldots + X_6)$ + part time $(Y_1 + \ldots + Y_6)$
Constraints: $X_1 + X_6 + \frac{5}{6} Y_1 \geq 15$
etc. enough people on shift
$X_1 + X_6 \geq \frac{2}{3}(X_1 + X_6 + Y_1)$
etc. $\frac{2}{3}$ must be full-time
$X_t \geq 0$, $Y_t \geq 0$ non-negativity

Product Sum Optimization Formula:

BREATHE

$$OPT[j] = \begin{cases} 0 & \text{if } j = 0 \\ v & \text{if } j = 1, \text{ else:} \\ \max(OPT[j-1] + v_j,\ OPT[j-2] + v_j \cdot v_{j-1}) \end{cases}$$

j = current term; adding; multiplying

Greedy Scheduling Proof: Let f be the class that finishes first. X is a maximal, conflict-free schedule that excludes f. Let g be first to finish in X. Since f finishes before g, it cannot conflict with any event in X. We can replace g with f and still be maximal and conflict-free. The best schedule that includes f must contain optimal schedule that doesn't conflict with f, L. Greedy algorithm chooses f, then by inductive hypothesis, computes optimal schedule of classes from L. There can be more than one optimal!