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

$\log_b x = y \Leftrightarrow b^y = x$

$\log_b 1 = 0$

$\log_b b^n = n$

$b^{\log_b n} = n$

$\log_b(m \cdot n) = \log_b m + \log_b n$

$\log_b\left(\frac{m}{n}\right) = \log_b m - \log_b n$

$\log_b x^n = n \cdot \log_b x$

## 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
```

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

## Merge Sort ← Divide and Conquer!

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

```
mergeSort(array A, int p, int r)
  if (p < r)
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
merge(array A, int p, int q, int r)
  array 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

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

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

Begins at the root of the tree and traces a path down. If key at current node equals value being searched for, search is complete. Similarly if node is null. Otherwise makes recursive call to left or right.

## Knapsack

$O(nW)$

```
int knapSack(n, W, wt[], val[])     n ← avail items, W ← avail weight
  for i = 0 to n                   ← items
    for w = 0 to W                 ← weights
      if i == 0 or w == 0, K[i][w] = 0          ← start w/ zeros
      else if wt[i-1] ≤ w                       ← fits
        K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],   ← OPT
                      K[i-1][w])
      else, K[i][w] = K[i-1][w]                 ← doesn't fit
  return K[n][W]
```

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

## Longest Common Subsequence

$O(m \cdot n)$

```
m = X.length      ← values
n = Y.length      ← length
new tables b[1..m, 1..n], c[0..n, 0..m]
all table elements = 0
for i = 1 to m
  for j = 1 to n
    if X_i == Y_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
```

```
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)
```

Populates a 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 cell contains answer. printLCS uses "↖" values to print chars.

## Coin Changing

$O(dA)$

Very similar to knapsack, main difference is that for each cell we are calculating min as follows:

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

answer in max r, c — row above (not taking coin) — (taking coin) — difference betwn. current col. and value of $r^{th}$ coin

## Pipe Cutting

$O(iT)$ T = Total pipe length, i = Number of individual pipe lengths

Basically the same as coin changing and knapsack. Each cell in the table is calculated as follows:

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

length of pipe at row — value of pipe at row

## Longest Increasing Subsequence

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

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

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

## Longest Palindromic Subsequence

$O(n^2)$

```
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])
```

Computes and saves longest from i to j and saves in array T[i][j].

final answer @ T[0][input.length-1]

## Recurrences (Common)

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

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

$T(n-1)+n = \Theta(n^2)$

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

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

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

$\lim_{n\to\infty} \frac{f(n)}{g(n)} =$

$\infty \rightarrow f(n) \in \Omega(g(n))$

$c > 0 \rightarrow f(n) \in \Theta(g(n))$

$0 \rightarrow f(n) \in O(g(n))$

AGATHA RUIZ DE LA PRADA

## Greedy Scheduling w/ Penalties:

$\Theta(n^2)$ ← NOT dominated by sorting

- Sort by penalties, decreasing.
- Schedule as late as possible before deadline. If no space before deadline, schedule at first available from end of array. ← $\Theta(n^2)$ to find a spot

← for optimization →

## Greedy Clue:

Uniform or unit-length amounts (each job is 1 minute). With greedy sort by whatever you'd like to minimize or maximize.

greedy!

## Product Sum Optimization Formula

$$OPT[j] = \begin{cases} 0 \text{ if } j=0 \\ v_1 \text{ if } j=1 \\ \max(OPT[j-1]+v_j,\ OPT[j-2]+v_j \cdot v_{j-1}) \end{cases}$$

j = current term; adding; multiplying

## Dynamic Programming vs. Greedy Algos

| Dynamic Programming | vs. Greedy Algos |
|---|---|
| At each step, choice is made based on solutions of sub-problems. | At each step, make choice that currently looks best. Locally optimal (greedy) choice. |
| Sub-problems are solved first. | Greedy choice is made first. |
| Bottom-up approach. | Top-down approach. |
| Slower, more complex. | Faster, simpler, may not work! |

## D.P. Properties

Optimal substructure: The solution to a problem includes the solutions to sub-problems.

Overlapping Sub-problems: The solution revisits the same problems repeatedly: Fibonacci, factorial, etc.

## Greedy Properties

Optimal Substructure: The solution to a problem includes the solutions to subproblems.

Greedy choice: Making greedy choice at every step still results in optimal solution. You never need to reconsider earlier choices.

## Fractional Knapsack

$O(n \cdot \log n)$

```
sort from highest density to lowest
while i ≤ n and weight < W        ← n = total items, W = capacity
  if weight + w[i] ≤ W
    x[i] = 1  ← take all!
  else         ← percent to take of ith
    x[i] = (W - weight)/w[i]
  weight = weight + x[i] * w[i]    ← w[i] = weight of individual item
  i++                              ← current weight
```

(i = current item)

Takes 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

$O(n)$ assume sorted

check all denominations

```
sort from highest value to lowest
for i = 0 to denominations.length
  while V >= deno[i]
    V = V - deno[i]      ← value to make; current denom
    ans.push(deno[i])    ← answer
```

Very similar to fractional knapsack. Take as many of highest value coins, then next value, etc., until V is no longer greater than smallest coin.

## Greedy Scheduling

$O(n \cdot \log n)$ ← sorting

```
GreedySched(S[1..n], F[1..n])
  Sort F by earliest finish time
    and permute S 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 input so that the event with the earliest finish time is first. Adds this event, then looks for the next event that starts after the previously-added event finishes. ● Let f be the class that finishes first. X is a maxim. 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 set will 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. ● Note that algo doesn't choose only optimal, only an optimal.

## Recurrence Format and Methods

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

Merge Sort →; # of subproblems; subproblem size; work

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

dividing and combining, outside recursion

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

### Muster Method:

$a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$, $c < 1$ ← regularity condition for case 3

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

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

## Huffman Coding

$O(n \cdot \log n)$ ← for sorting

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

Greedy because most frequently used letters are given shortest codes.

## Hotel Stopping Problem

$\Theta(n^2)$

```
S[i] = minimum total penalty for stop @ j
S[0] = 0
for i = 1 to n          ← number of hotels
  S[i] = ∞              ← current best
  for j = 0 to i        ← for i
    S[i] = min(S[i], S[j] + (200 - (a_i - a_j))^2)   ← possible improv. by stopping @ j
return S[n]
```

Two loops both iterating over same array of best values, updating with best val each pass.

## Big-O Classes

(↓ slower)

- 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: $O(n!)$

## Canoe Rental

$\Theta(n^2)$

```
n = # tradeposts
OPT[1] = 0, P[1] = 0
for i = 2 to n
  min ← R[1,i]          ← no stops
  P[i] ← 1
  for k = 2 to i-1 {
    if OPT[k] + R[k,i] ≤ min      ← better?
      min = OPT[k] + R[k,i]      ← update
      P[i] = k }
  OPT[i] = min
return P[]
```

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

## 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 $g(n) = \Omega(f(n))$ if f bounded above by g, g bounded below by f

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

If $a = b$ and $b = c$ then $a = c$