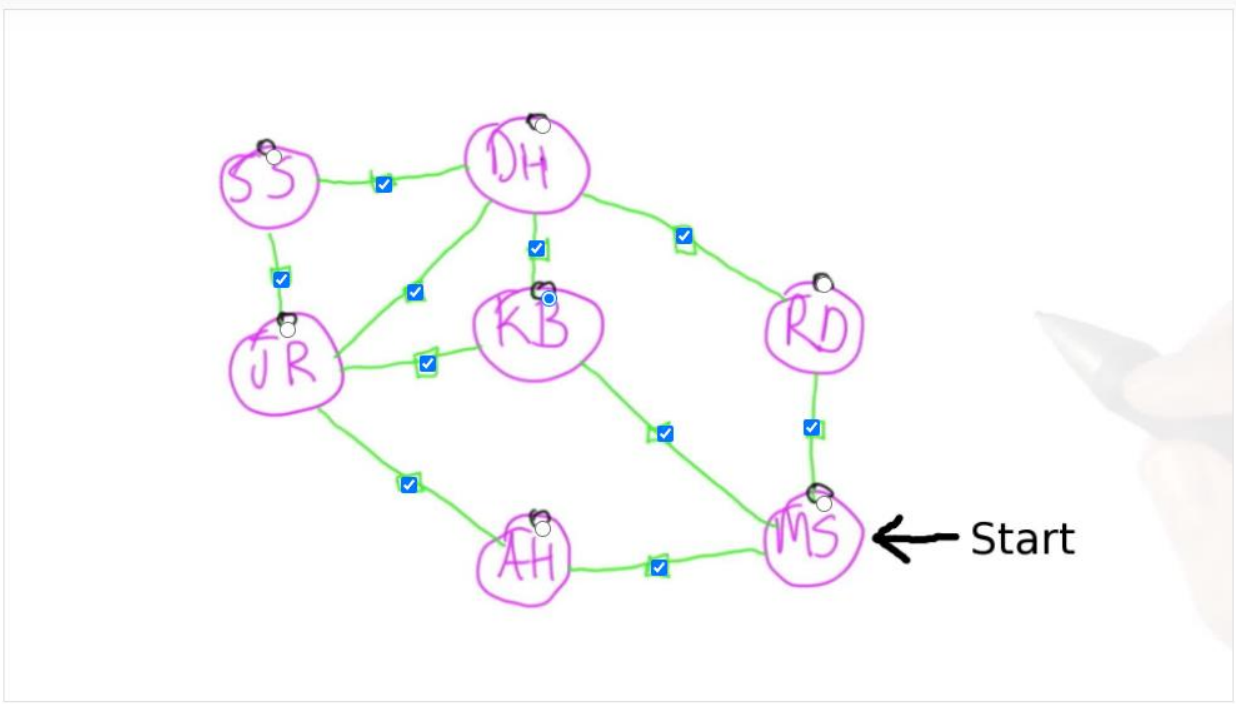
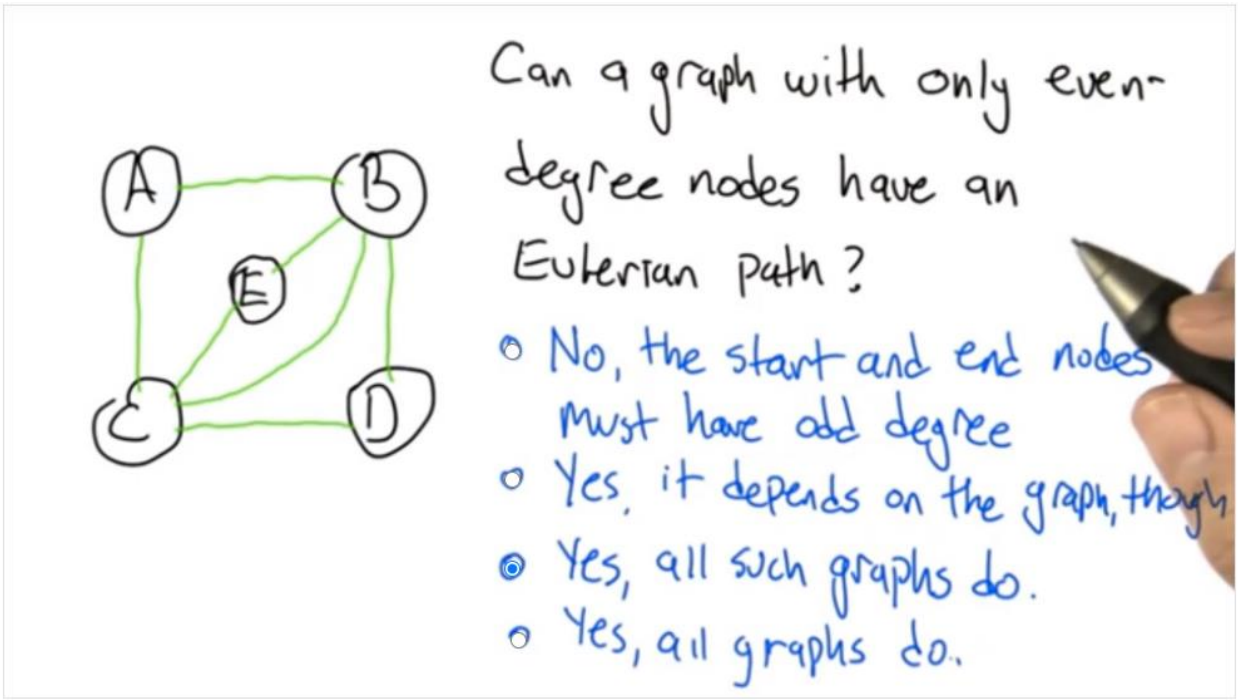
Fellhipe Gutierrez

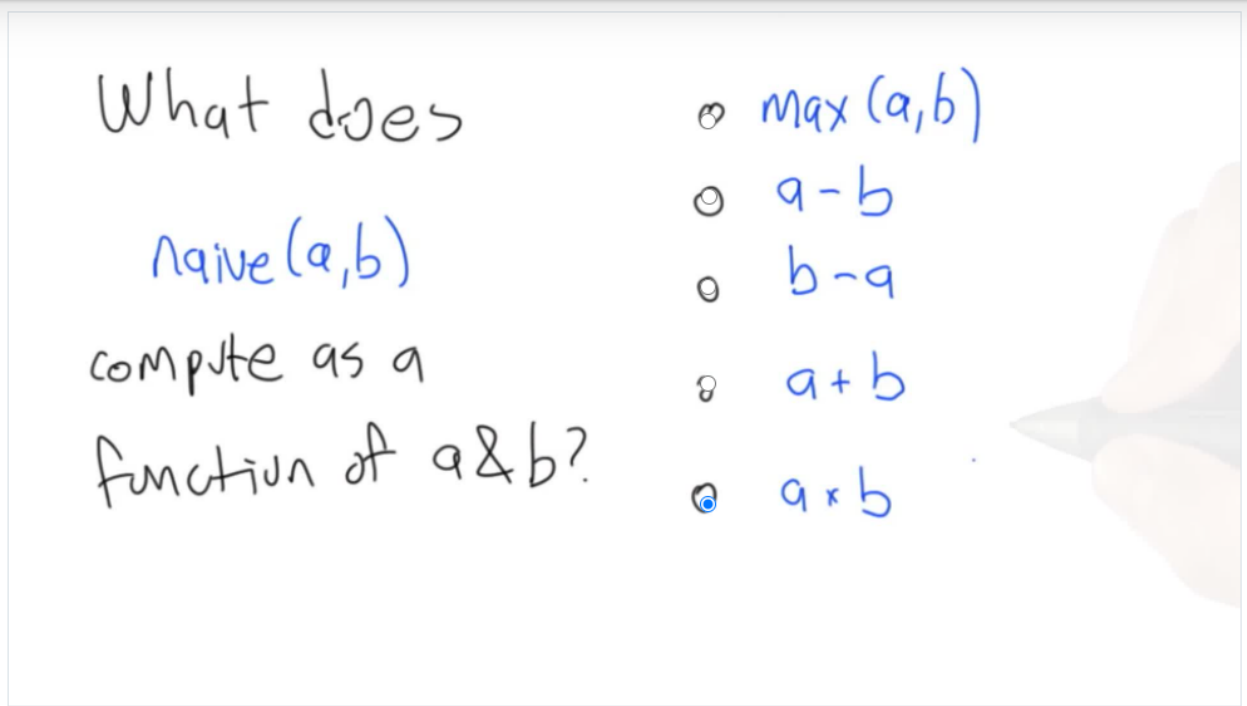
Magic Trick



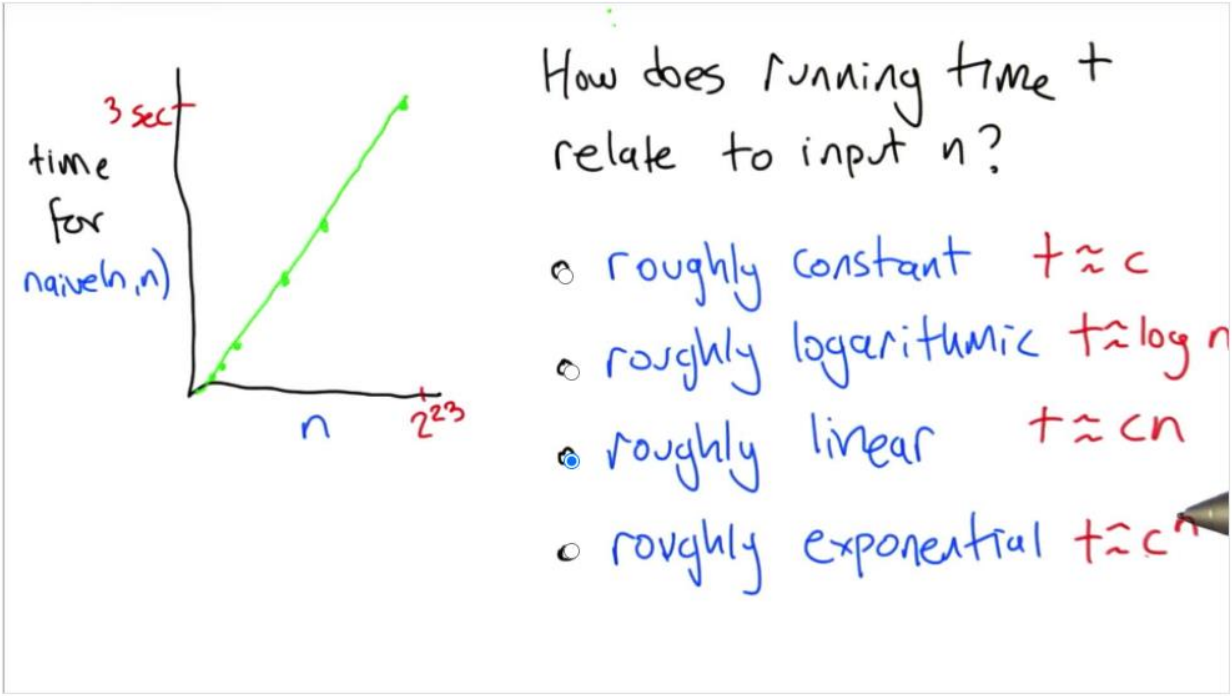
Eulerian Path



Case Study

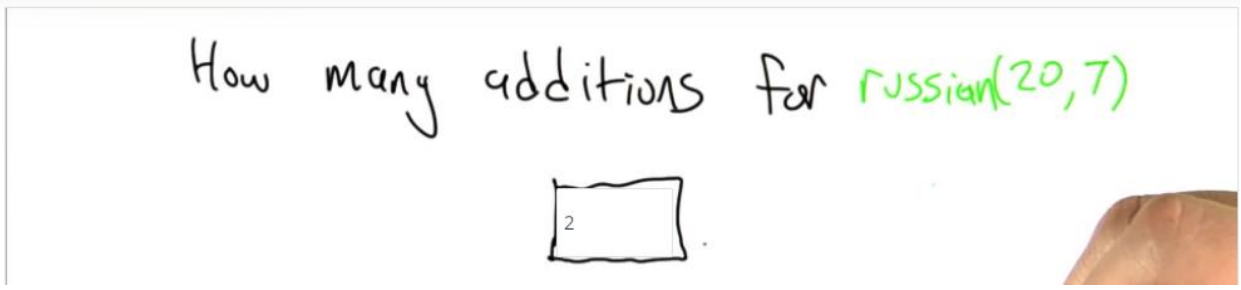


Running Time



Russian Peasant Algorithm

How Many Additions



Measuring Time

```
# How many units of time will this python code take to run?

s = 0
for i in range(10):
    s = s + i
print s
```

```
12
```

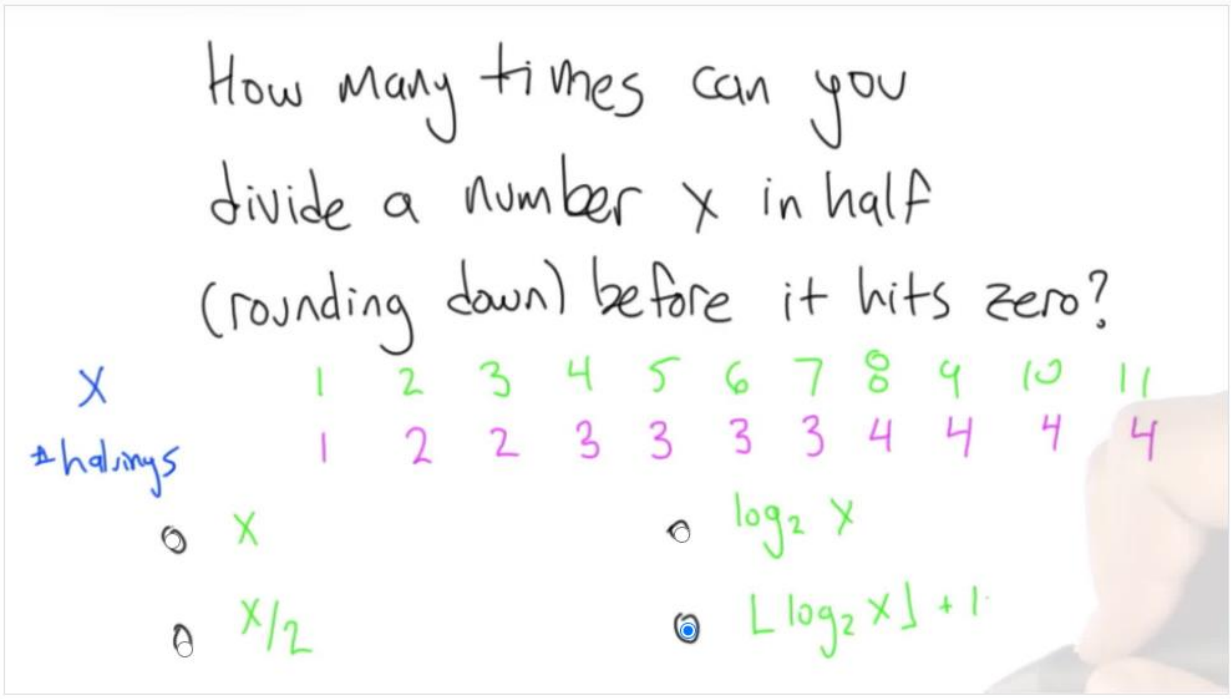
Counting Steps

```
import math

def time(n):
    """ Return the number of steps
    necessary to calculate
    `print countdown(n)`"""
    steps = 0
    steps = 3 + 2 * math.ceil(n/5.0)
    return steps

def countdown(x):
    y = 0
    while x > 0:
        x = x - 5
        y = y + 1

    return math.ceil(y*2)+3

print countdown(50)
```

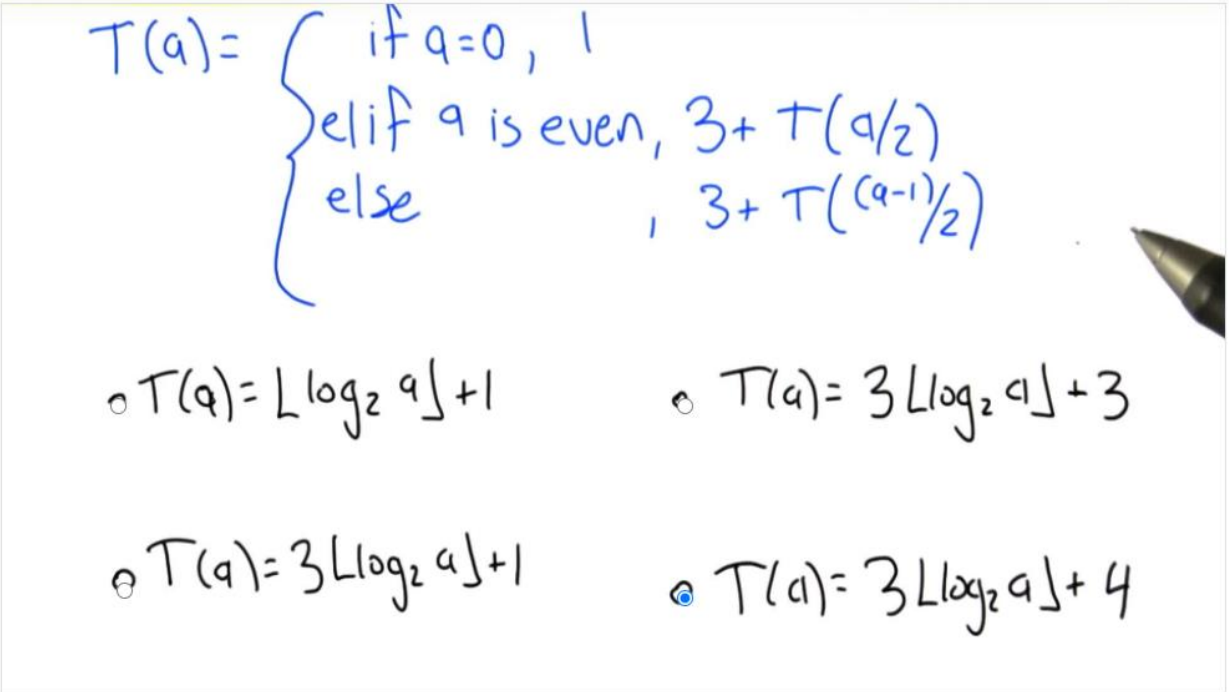
Steps for Naïve

```python
# counting steps in naive as a function of a

def naive(a, b):
    x = a
    y = b
    z = 0
    while x > 0:
        z = z + y
        x = x - 1
    return z

def time(a):
    # The number of steps it takes to execute naive(a, b)
    # as a function of a
    steps = 0
    steps = 2*a +3
    return steps

print (time(5))
```

Halving

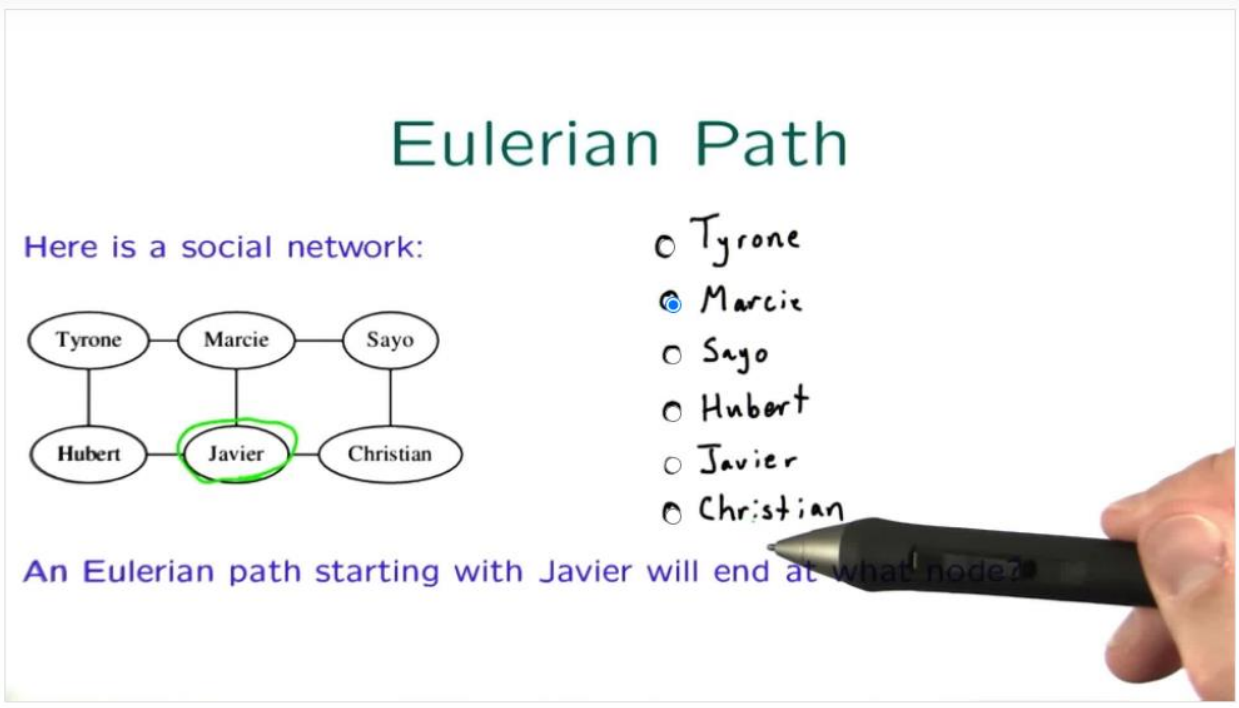


Recurrence Relation

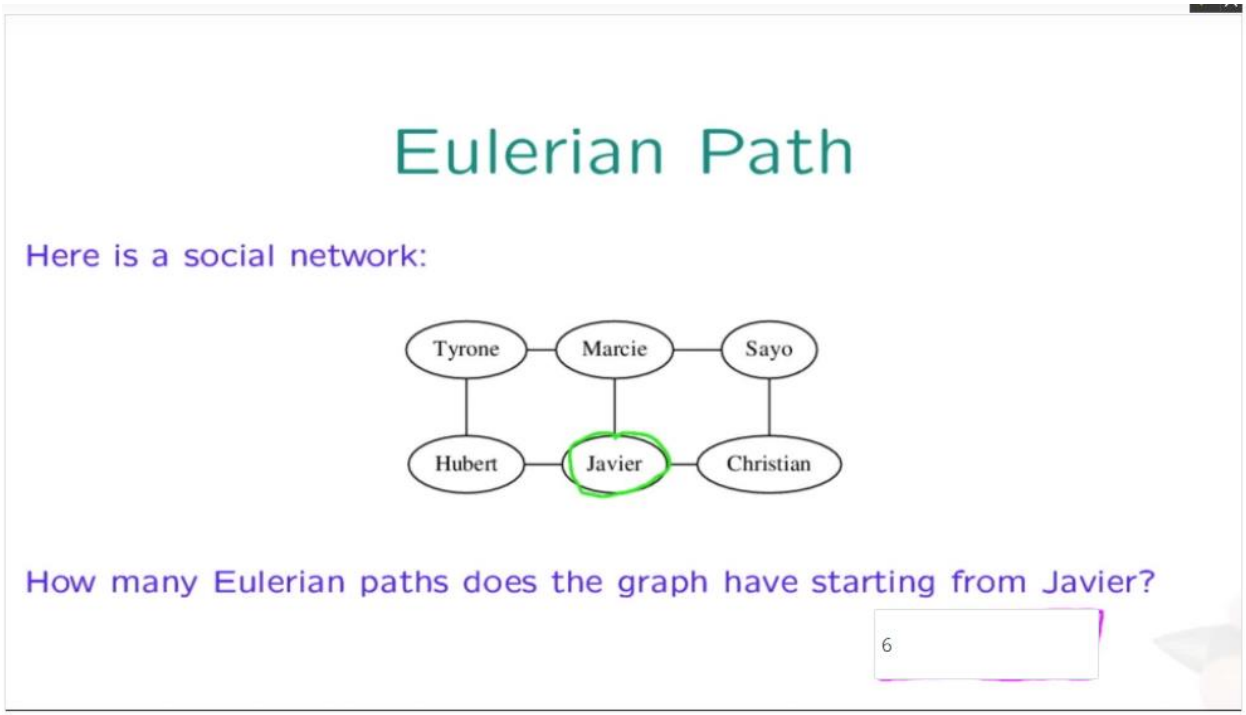


Lección 2

Eulerian Path



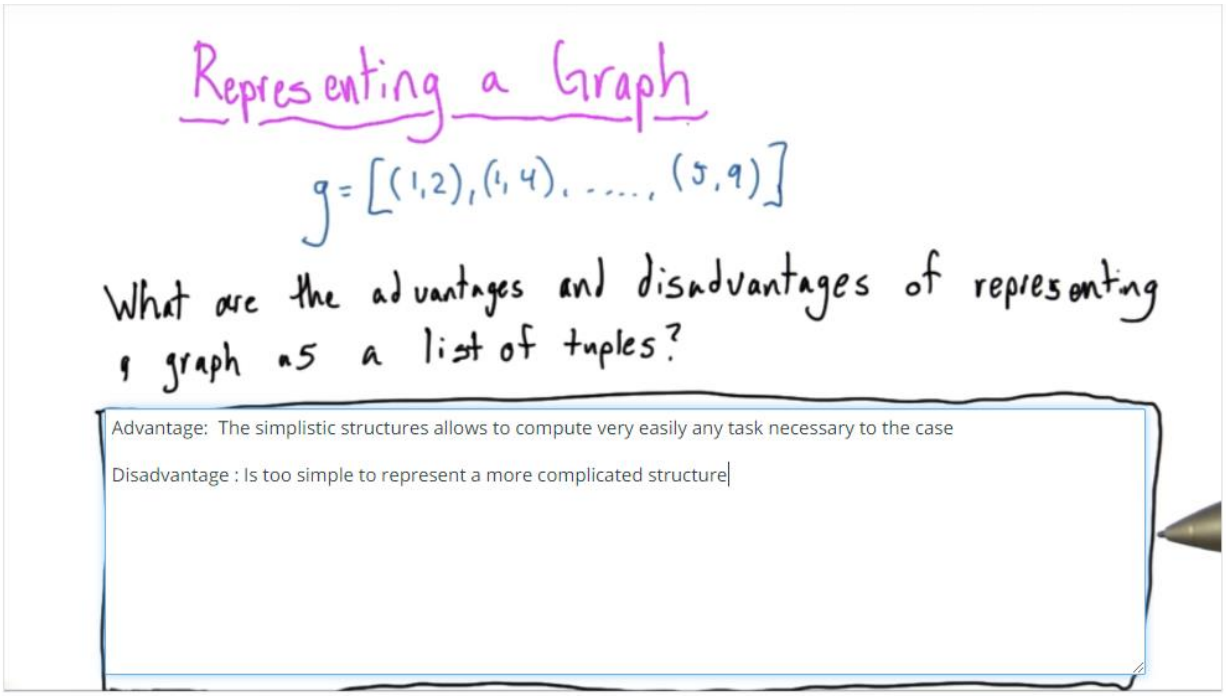
Counting Eulerian Paths



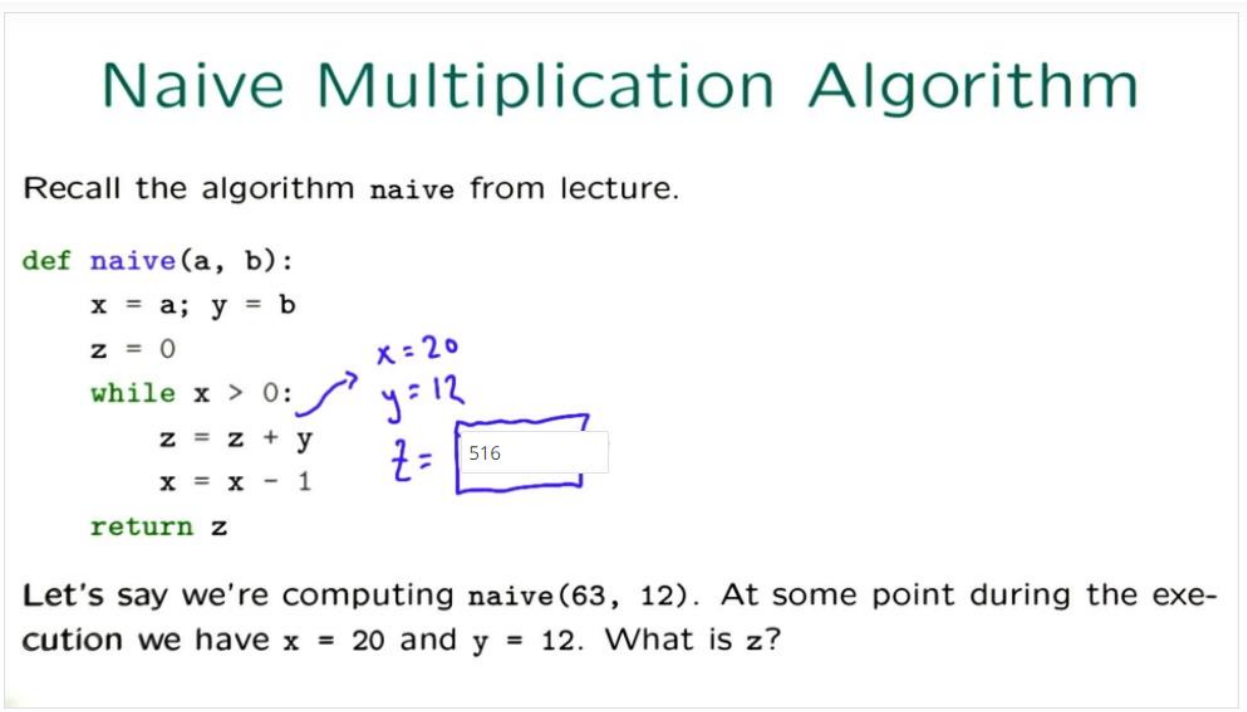
Create Graph with Eulerian Tour

```python
import itertools

def create_tour(nodes):
    # nodes = [1,2,3]
    # return [(1,3), (1,2), (2,3)]
    return list(itertools.permutations(nodes, 2))

#########

def get_degree(tour):
    degree = {}
    for x, y in tour:
        degree[x] = degree.get(x, 0) + 1
        degree[y] = degree.get(y, 0) + 1
    return degree

def check_edge(t, b, nodes):
    """
    t: tuple representing an edge
    b: origin node
    nodes: set of nodes already visited
    if we can get to a new node from `b` following `t`
    then return that node, else return None
    """
    if t[0] == b:
        if t[1] not in nodes:
            return t[1]
    elif t[1] == b:
```

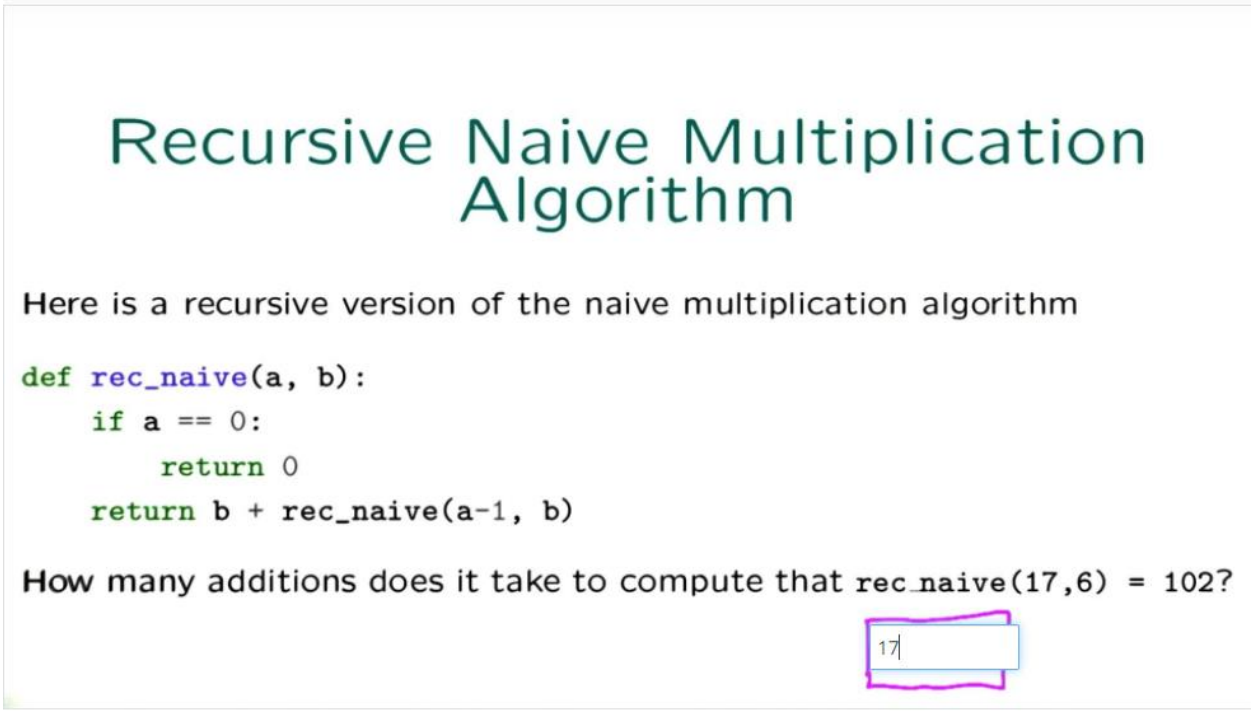
Representing a Graph



Naïve Multiplication Algorithm



Recursive Naïve



Russian Multiplication Algorithm

Russian Multiplication Algorithm

Recall the algorithm russian from the lecture.

```
def russian(a, b):
    x = a; y = b
    z = 0
    while x > 0:
        if x % 2 == 1:
            z = z + y
        y = y << 1
        x = x >> 1
    return z
```

x = 7
z = 84
y = 96

Let's say we're computing `russian(63,12)`. At some point during the execution, we have `x = 7` and `z = 84`. What is `y` at this moment?

Clique

Clique

Here's a loop:

```
def clique(n):
    print "in a clique..."
    for j in range(n):
        for i in range(j):
            print i, "is friends with", j
```

How many units of time does it take to execute `clique(4)`? Count each print statement as one unit and count each time `range` is evaluated as one unit.

12

General Clique

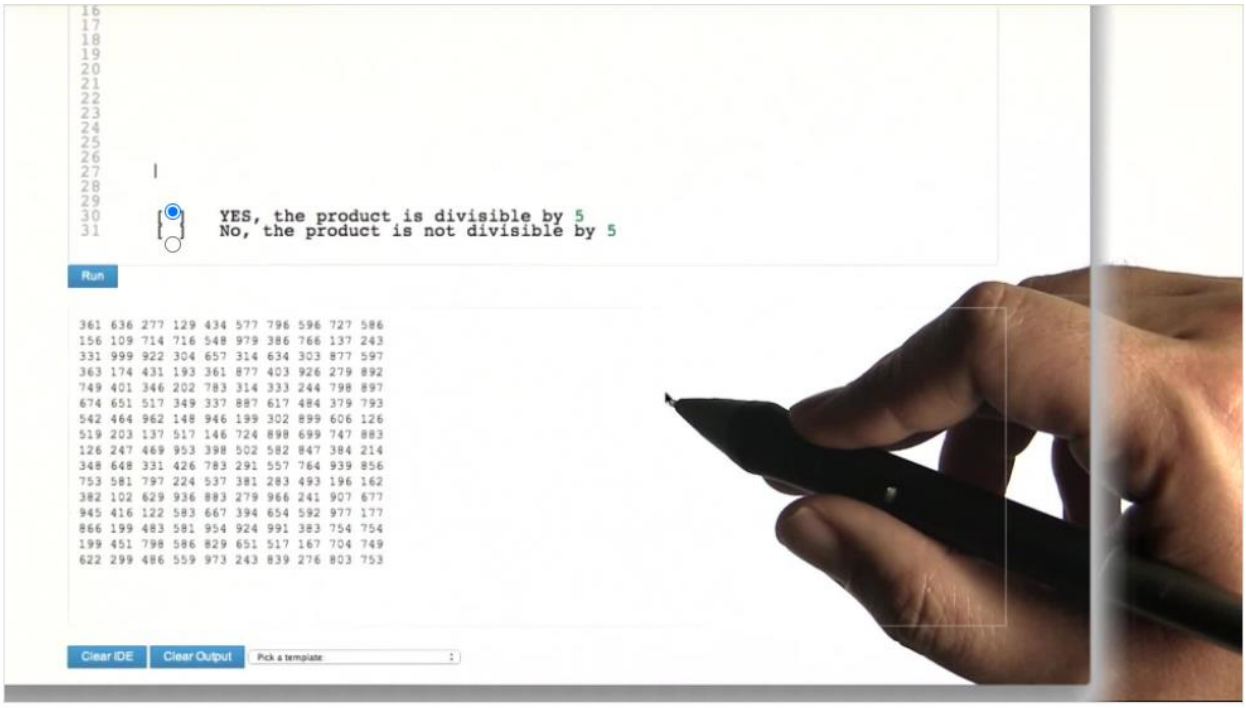
```
1
2  # Write a function, `count`
3  # that returns the units of time
4  # where each print statement is one unit of time
5  # and each evaluation of range also takes one unit of time
6
7  def count(n):
8      # Your code here to count the units of time
9      # it takes to execute clique
10     return 2 + (1+n)*n/2
11
12 def clique(n):
13     print "in a clique..."
14     for j in range(n):
15         for i in range(j):
16             print i, "is friends with", j
17
18 if __name__ == '__main__':
19     print count(4)
```

Challenge Find Eulerian Tour

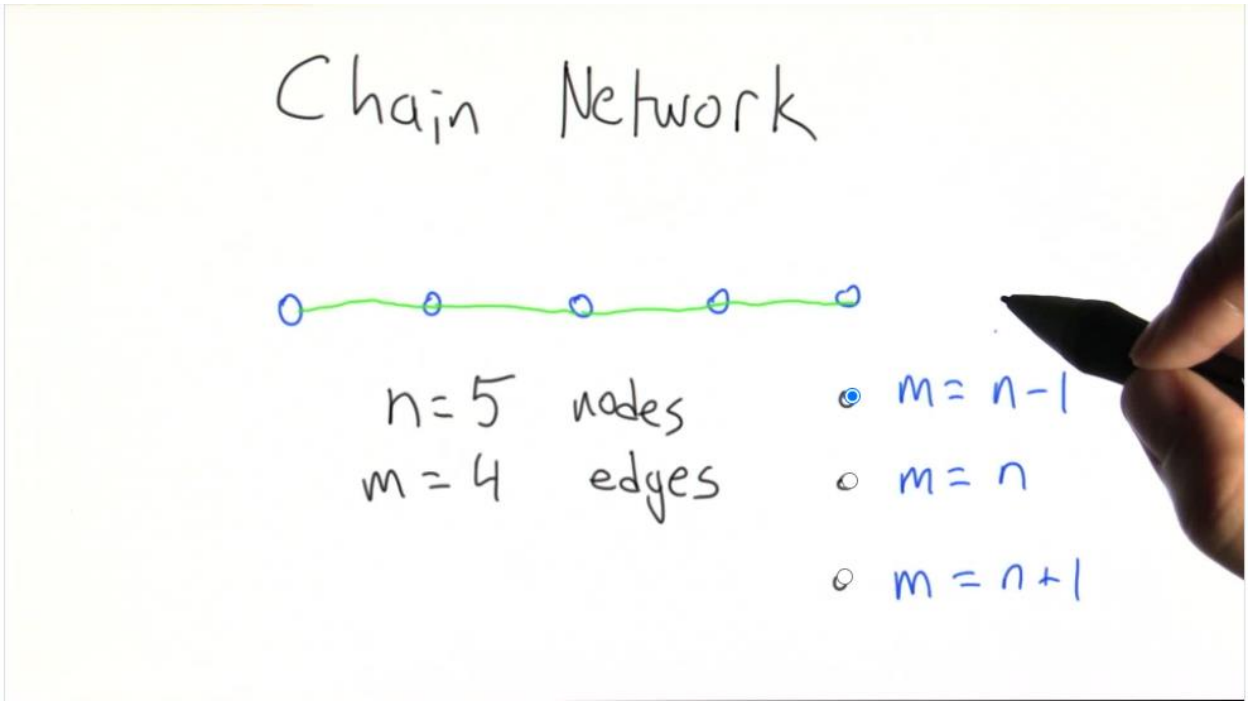
```
def find_eulerian_tour(graph):

    def _next_node(edge, current):
        return edge[0] if current == edge[1] else edge[1]

    def _remove_edge(raw_list, discard):
        return [item for item in raw_list if item != discard]

    search = [[[], graph[0][0], graph]]
    while search:
        path, node, unexplore = search.pop()
        path += [node]

        if not unexplore:
            return path

        for edge in unexplore:
            if node in edge:
                search += [[path, _next_node(edge, node), _remove_edge(unexplore, edge)]]

if __name__ == '__main__':
    graph = [(1, 2), (2, 3), (3, 1), (3, 4), (4, 3)]
    print find_eulerian_tour(graph)
```

LECCIÓN 3

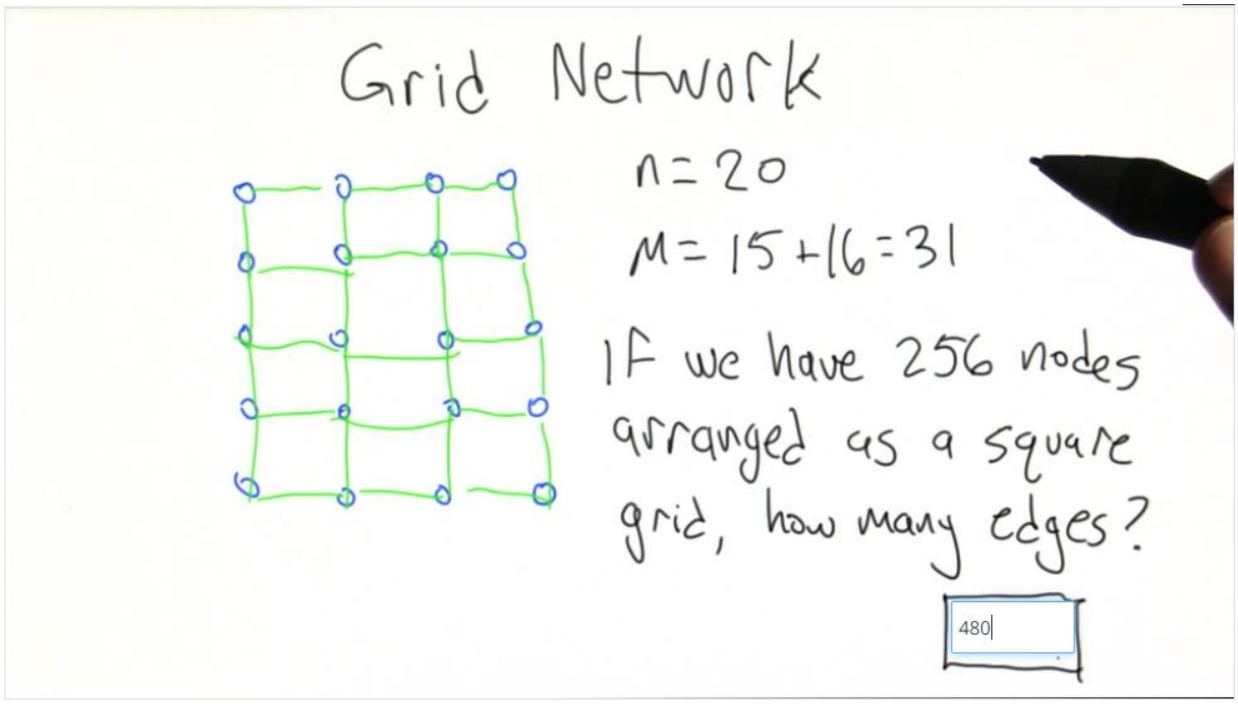
Divisible by Five



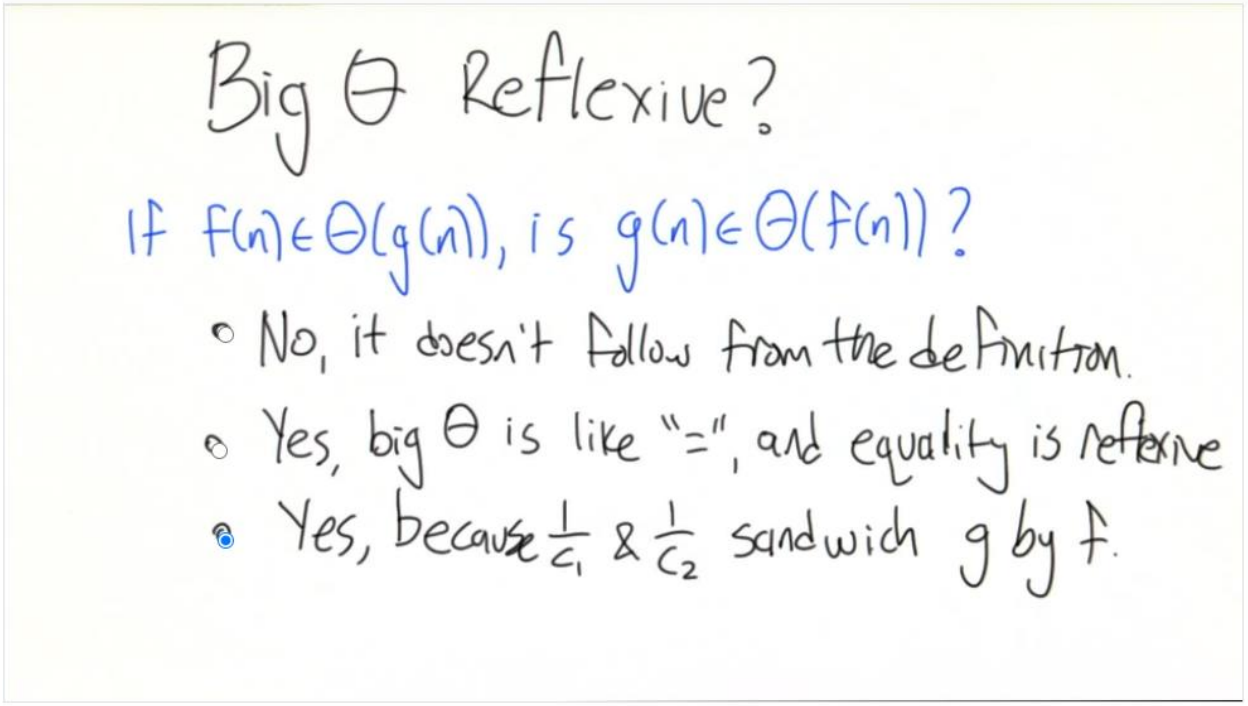
Chain Network



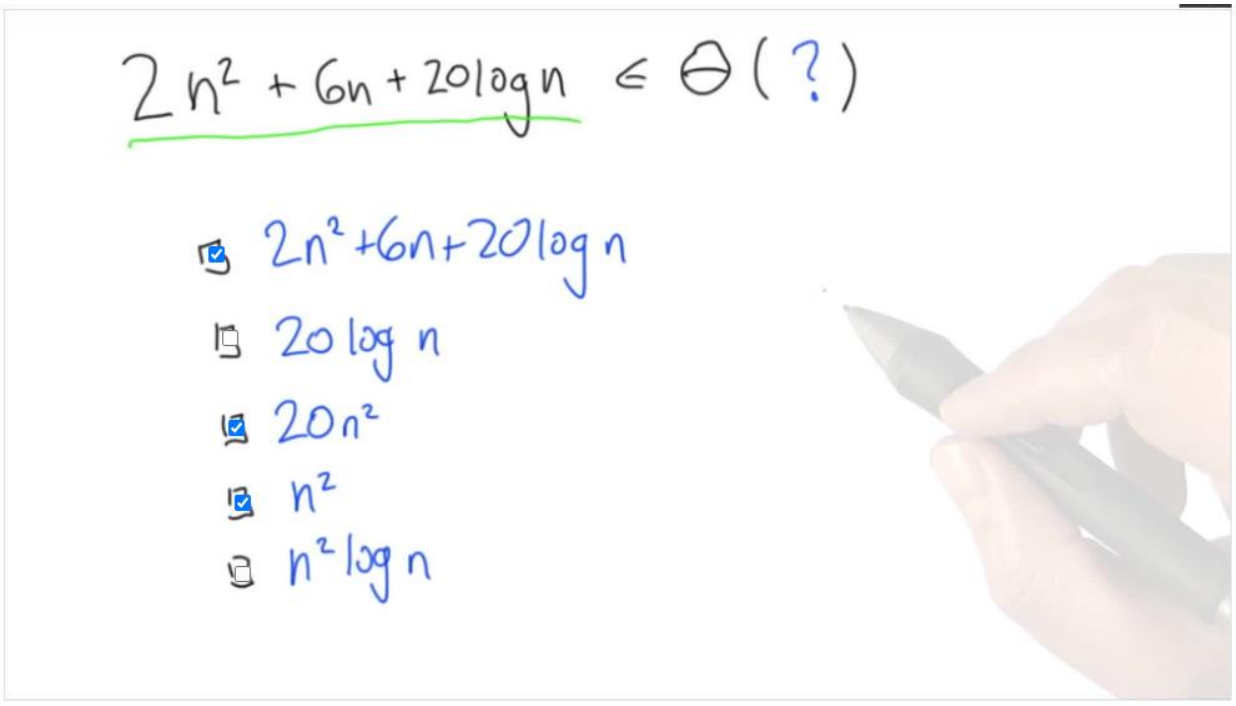
Grid Network



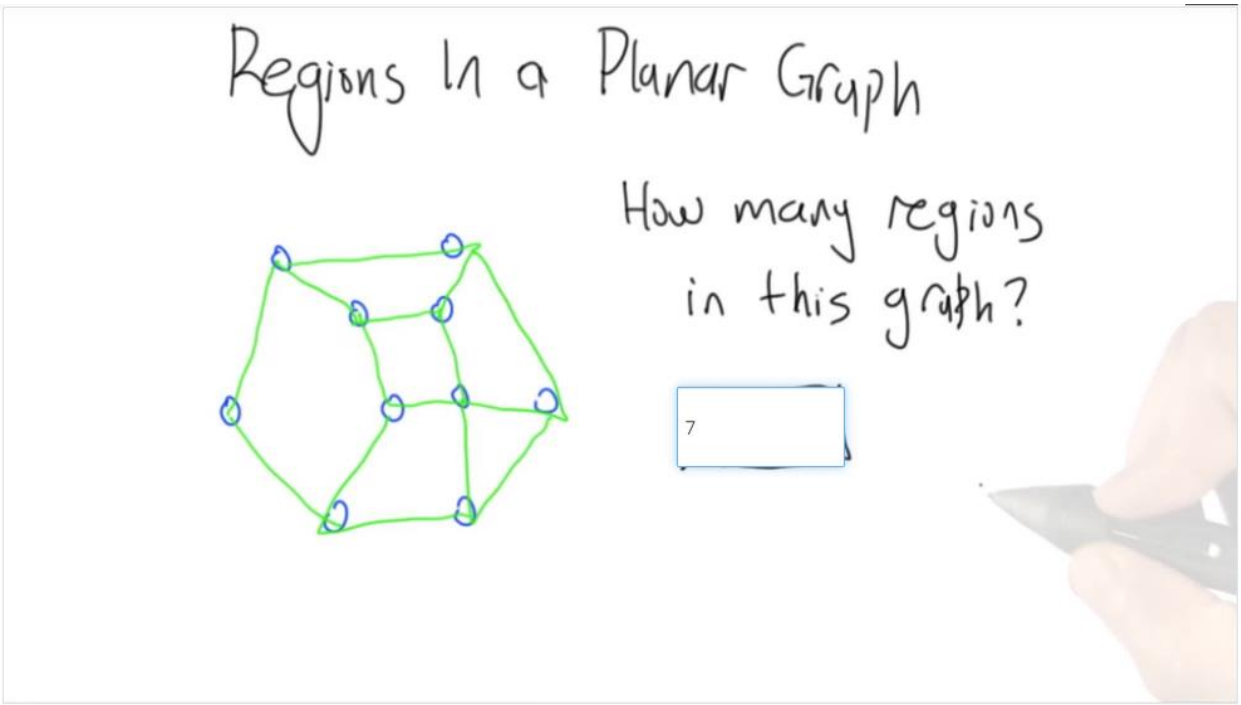
Big-Theta Reflexive



Big-Theta Practice



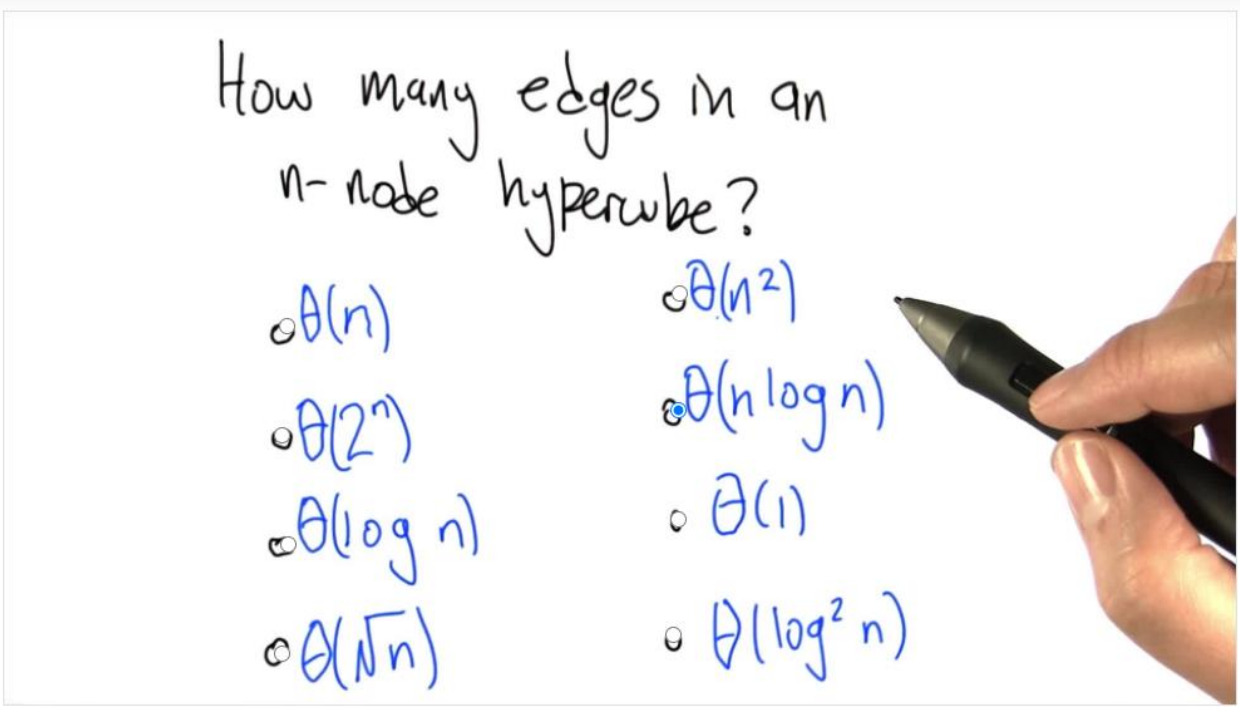
Regions on a Planar Graph



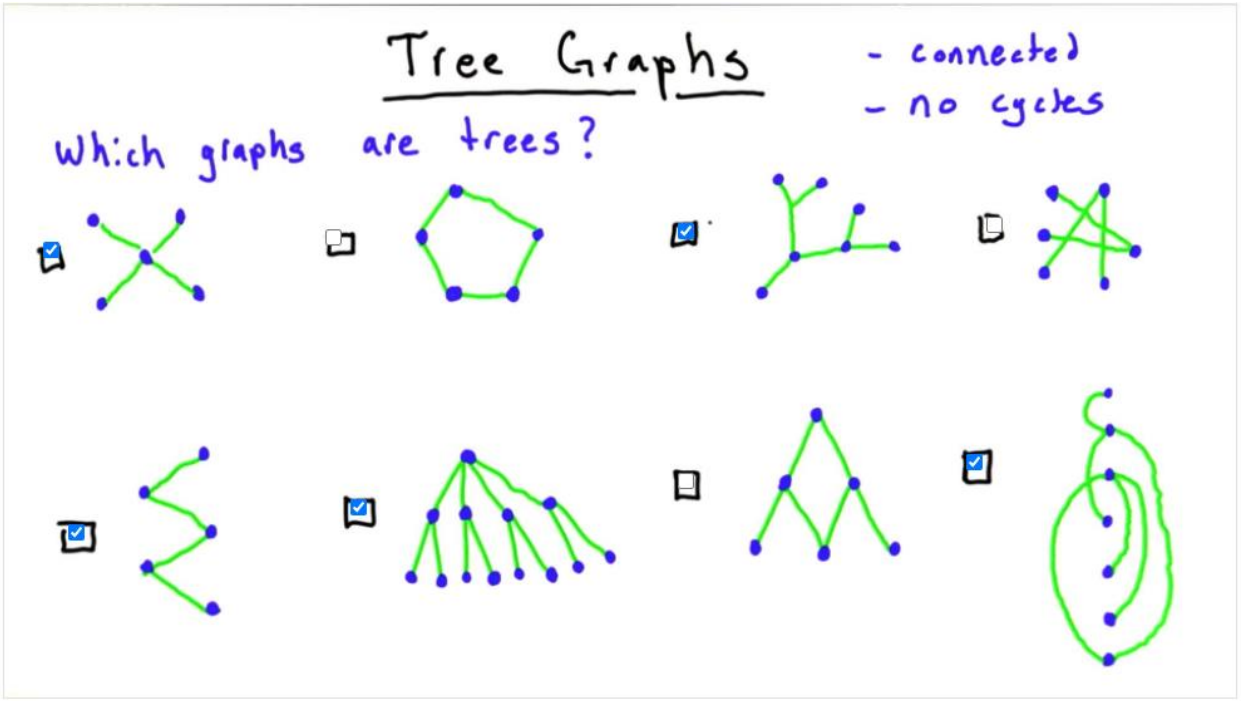
Complete Graph

```python
def make_link(G,node1,node2):
    if node1 not in G:
        G[node1]={}
    (G[node1])[node2]=1
    if node2 not in G:
        G[node2]={}
    (G[node2])[node1]=1
    return G
#
# How many edges in a complete graph on n nodes?
#

def clique(n):
    # Return the number of edges
    # Try to use a mathematical formula...
    #Graph
    G={}
    #edges
    for i in range(n):
        for j in range(n):
            if i<j: make_link(G,i,j)

    return sum([len(G[node]) for node in G.keys()])/2

for n in range (1,10):
    print n, clique(n),n*(n-1)/2
```

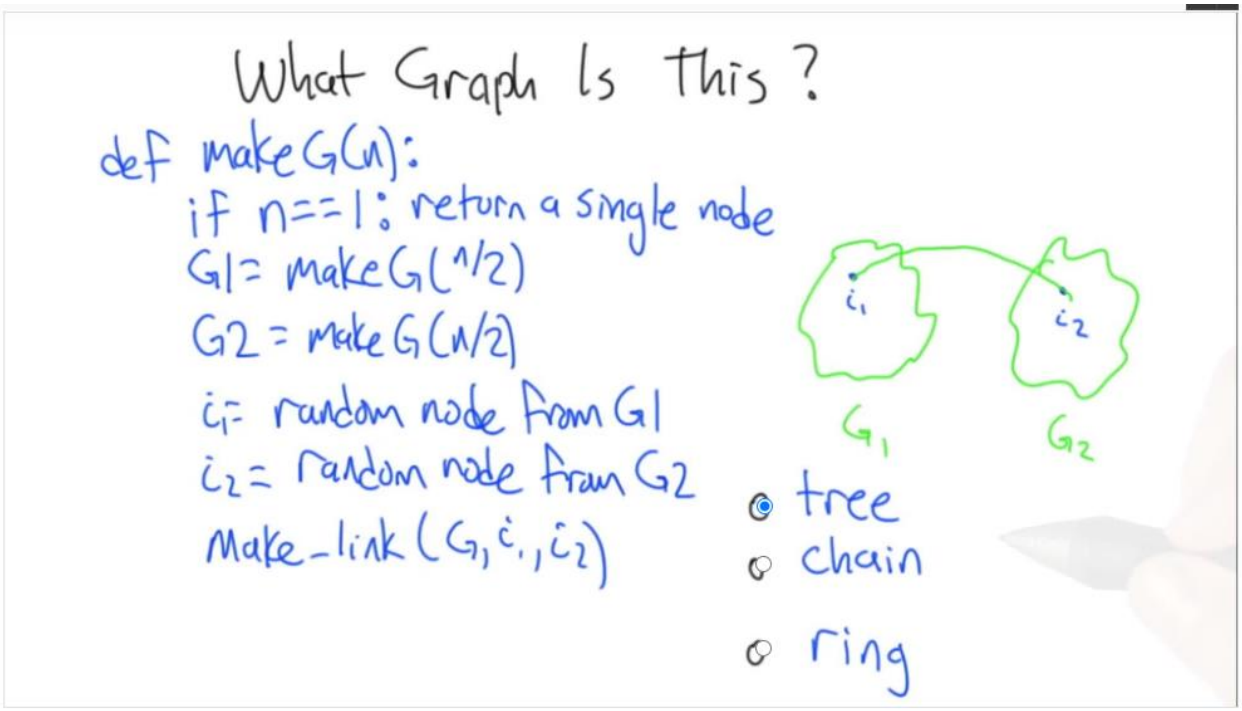
Hypercube Edges



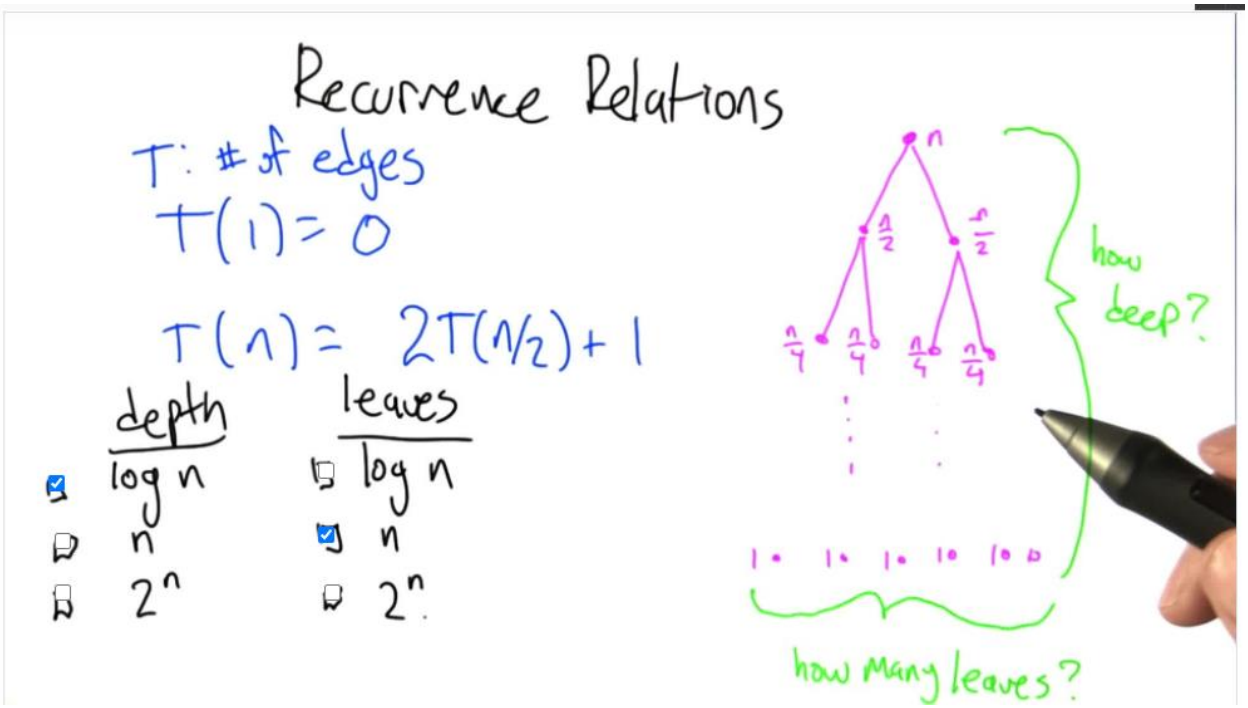
Tree Graphs



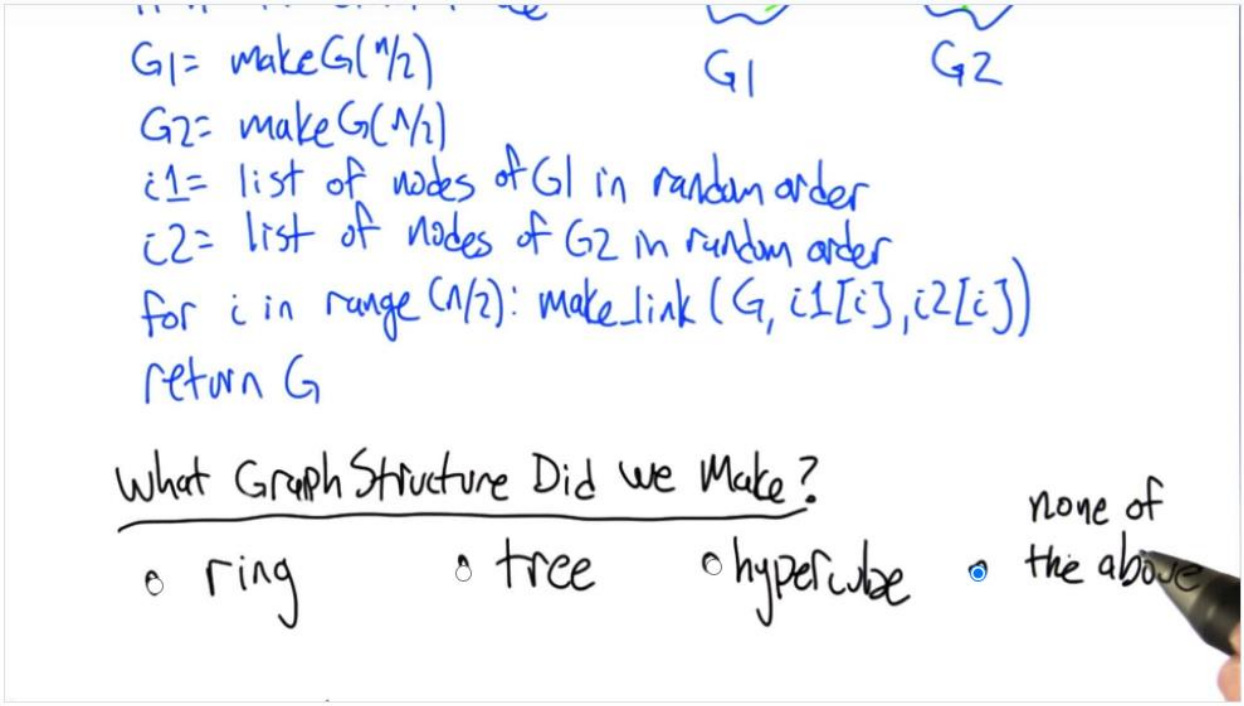
Recursive Graphs



Recurrence Relation



Tangled Hypercube

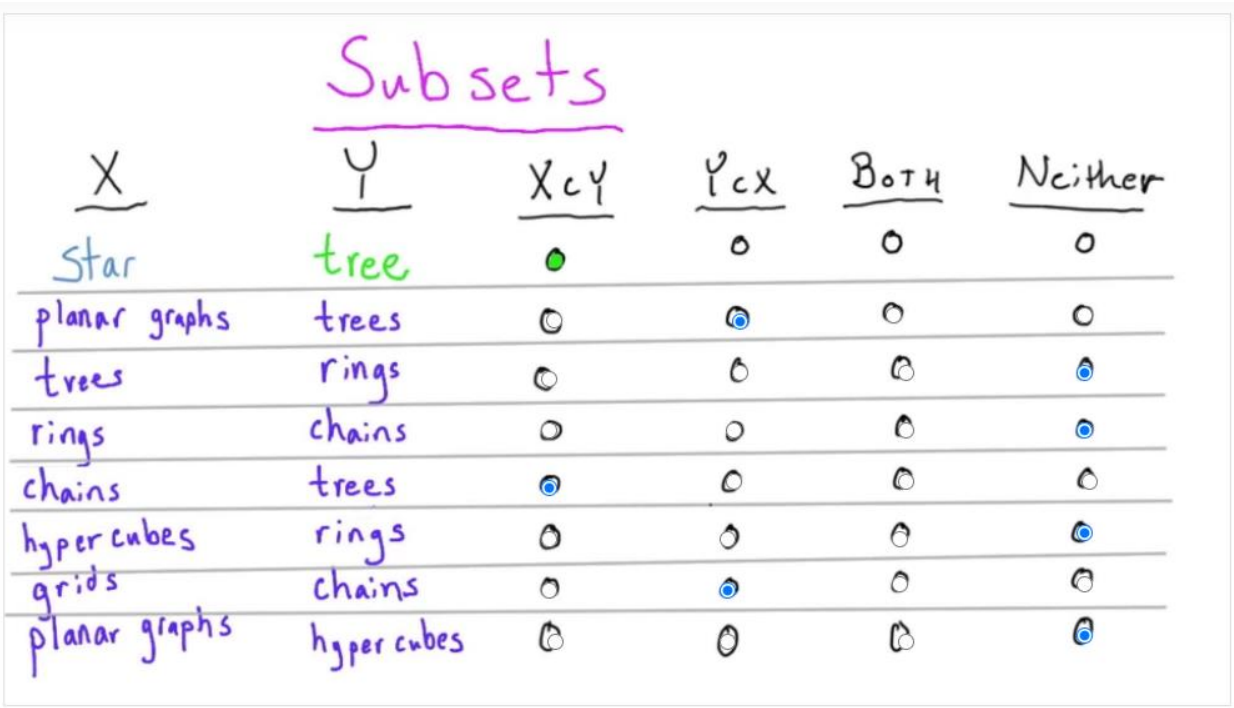


LECCIÓN 4

Star Network

```
def star_network(n):
    # return number of edges
    return n-1

print star_network(5)
```

Subsets



Function Comparison

Function Comparision

$4n^2 + (\log n)^7 + 9n(\log n)^2 + n^{\frac{2}{3}}$ is

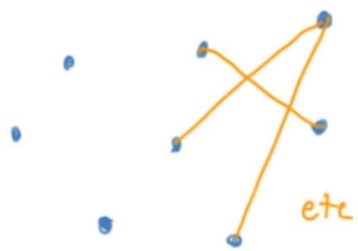
- [x] $\Theta(n^2)$
- [ ] $\Theta((\log n)^7)$
- [ ] $\Theta(n^{\frac{2}{3}})$
- [ ] $O(n)$
- [x] $O(n^2)$
- [x] $O(n^3)$

Planar Graphs

Planar Graphs

Draw yourself a planar graph with 8 nodes and 15 edges.

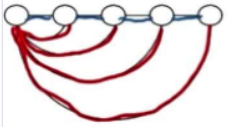
How many regions does the graph have? 9



Combination Locks

Combination Lock

A "combination lock" graph on n nodes consists of the edges in a chain **and** the edges in a star, with the left end of the chain as the center of the star. Here's the five node version:

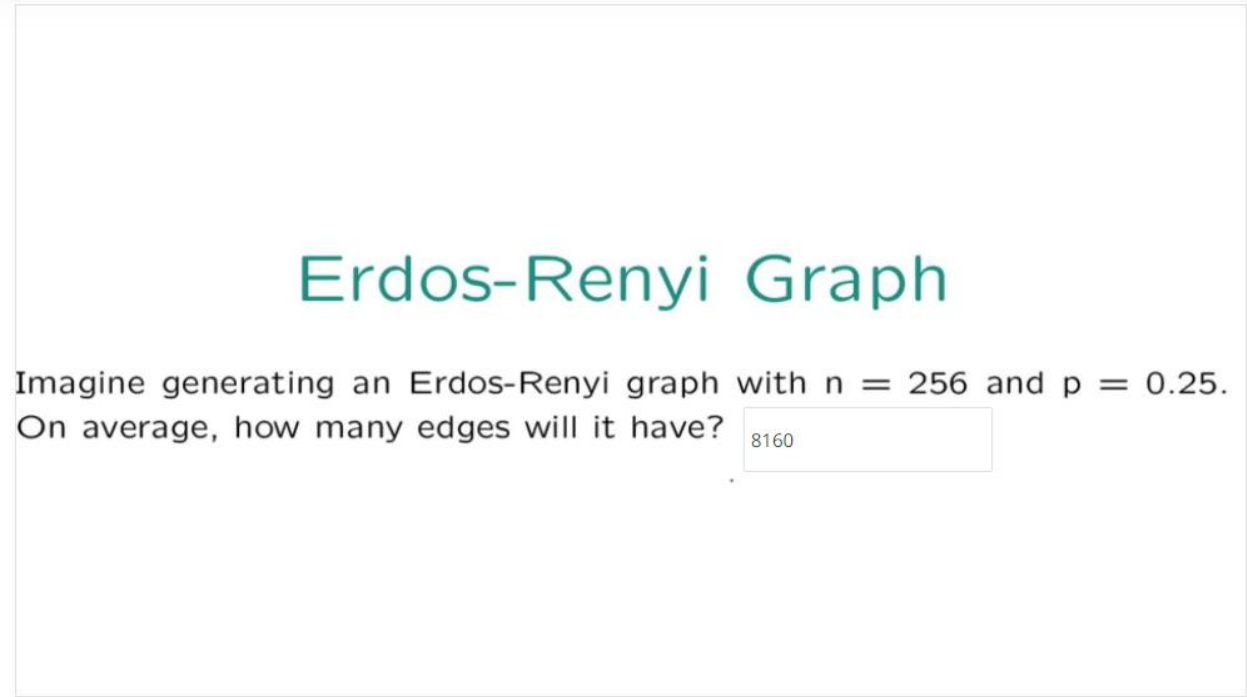


The number of edges is
- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$

Make a Combination Lock

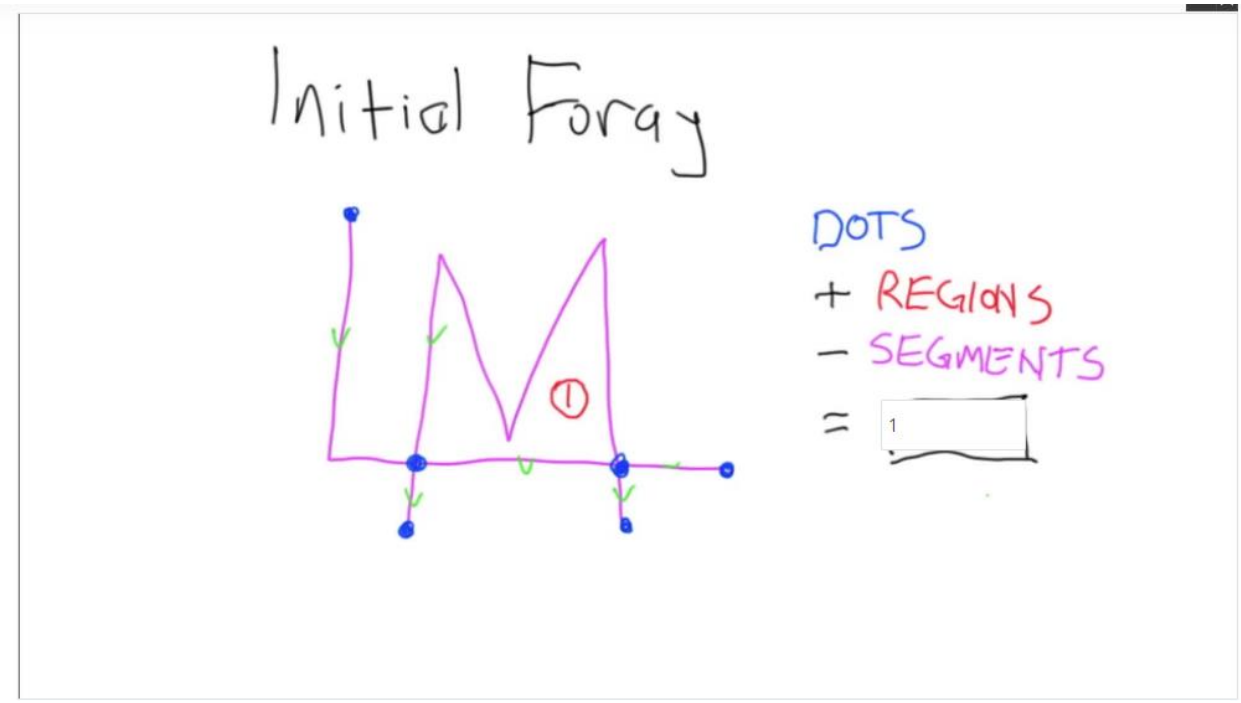
```
10 def create_combo_lock(nodes):
11     G = {}
12     # your code here
13     make_link(G, nodes[0], nodes[1])
14     for a in range(2,len(nodes)):
15         make_link(G, nodes[a-1], nodes[a])
16         make_link(G, 0, nodes[a])
17     return G
18
```

Erdos-Renyi

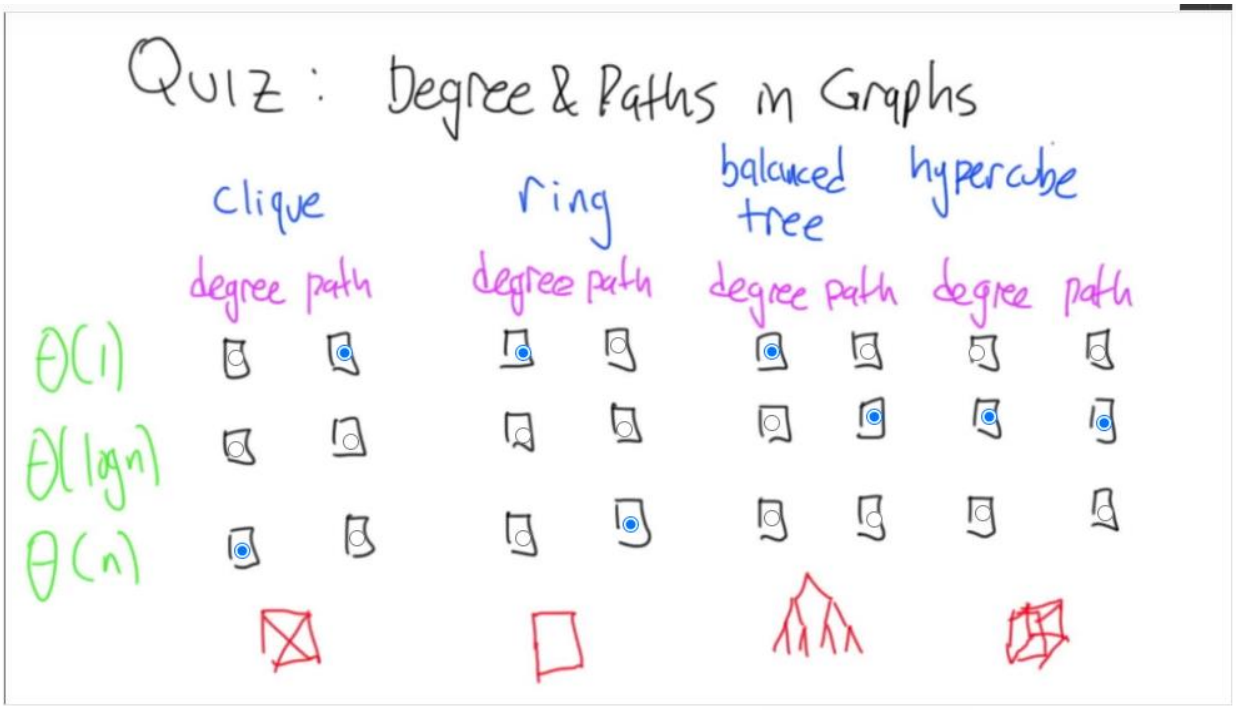


LECCIÓN 6

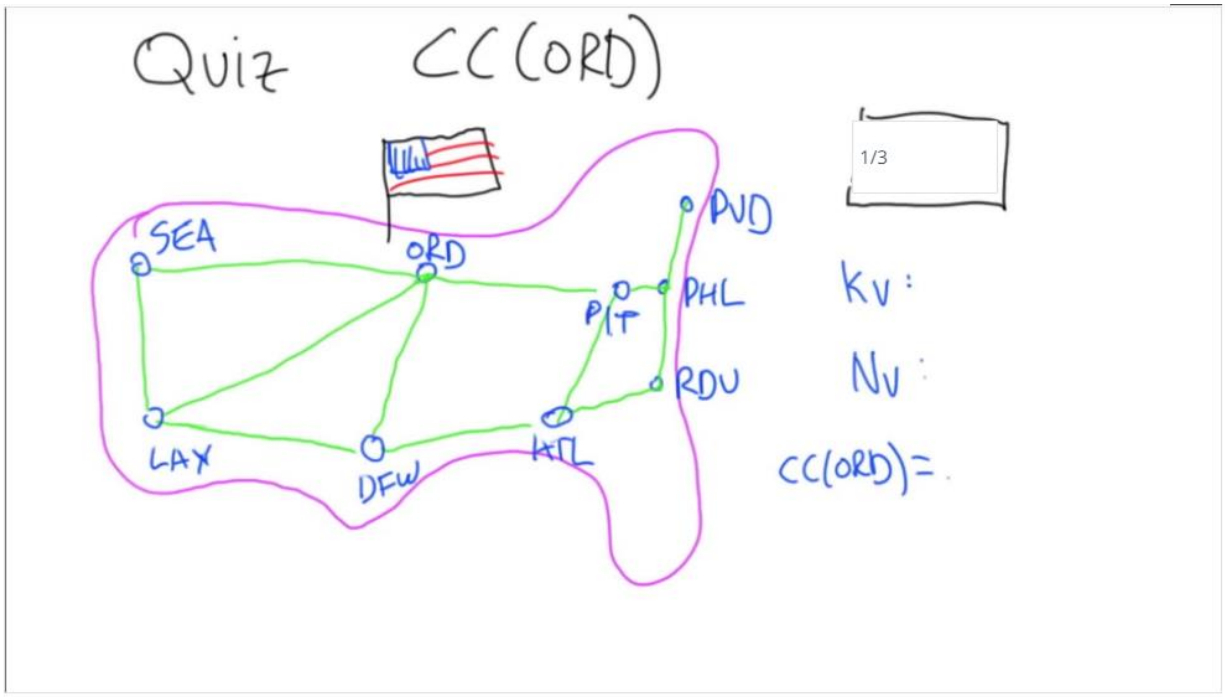
Initial Foray



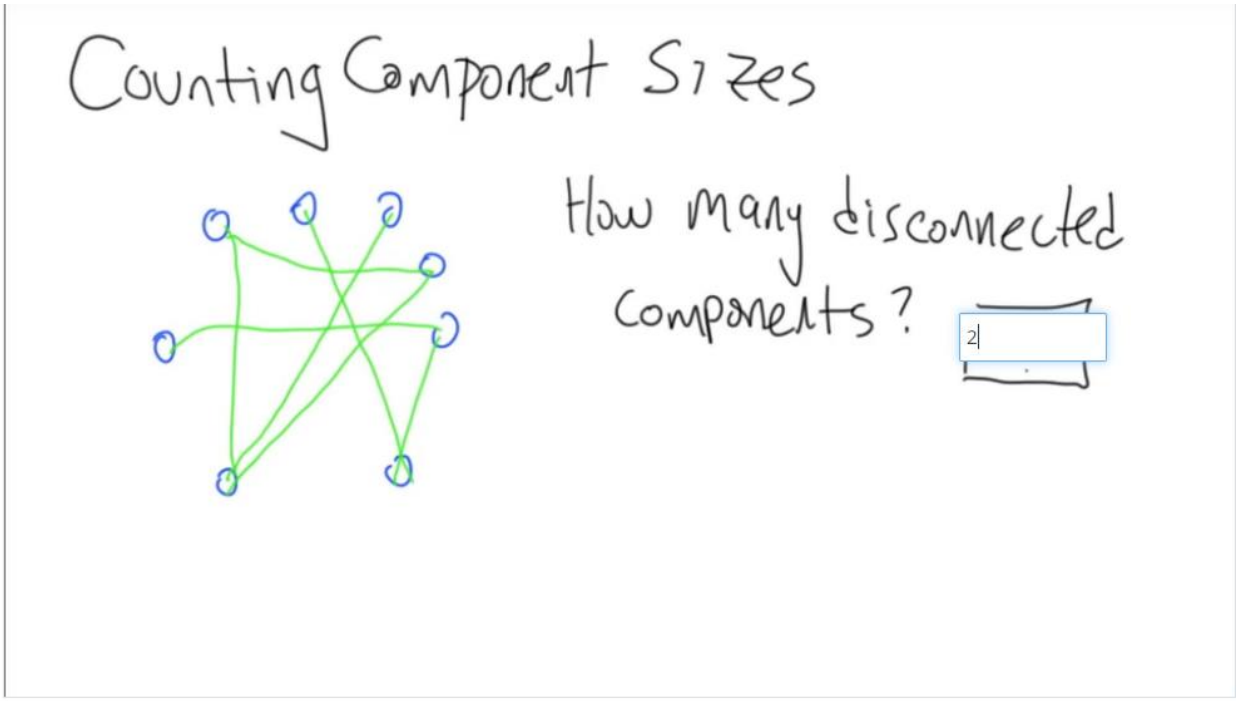
Properties of Social Networks



Clustering Coefficient Quiz



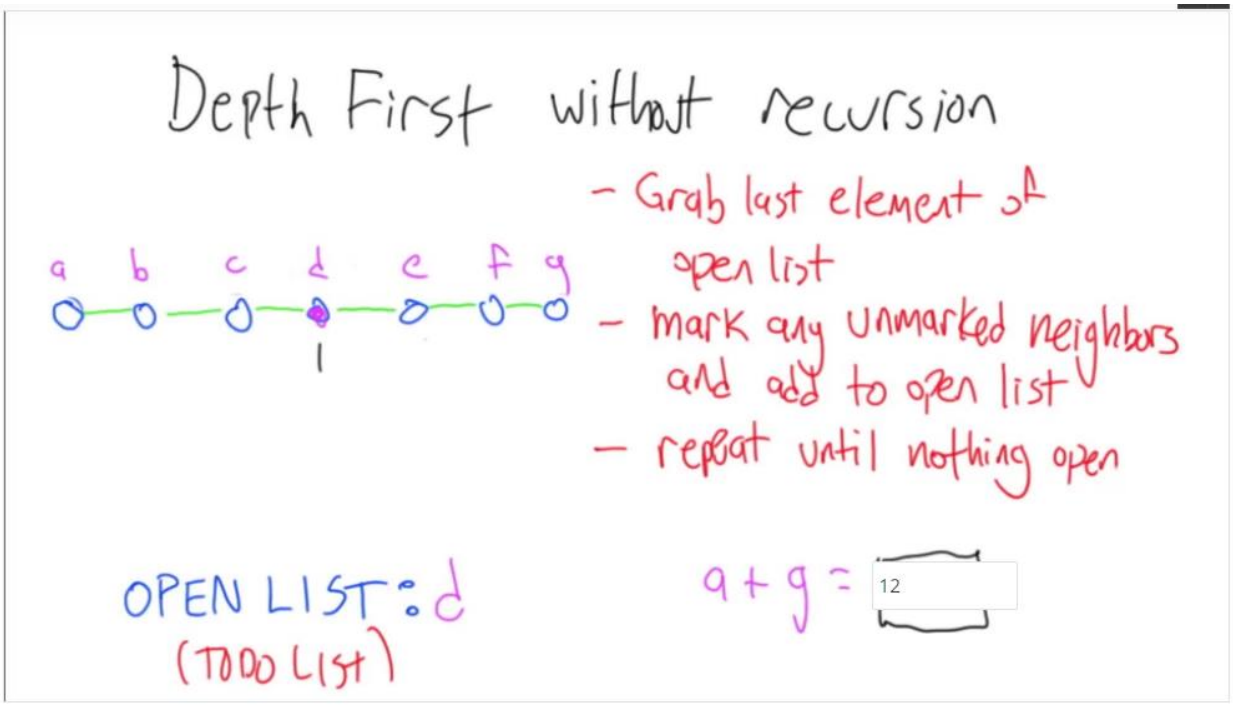
Connected Components



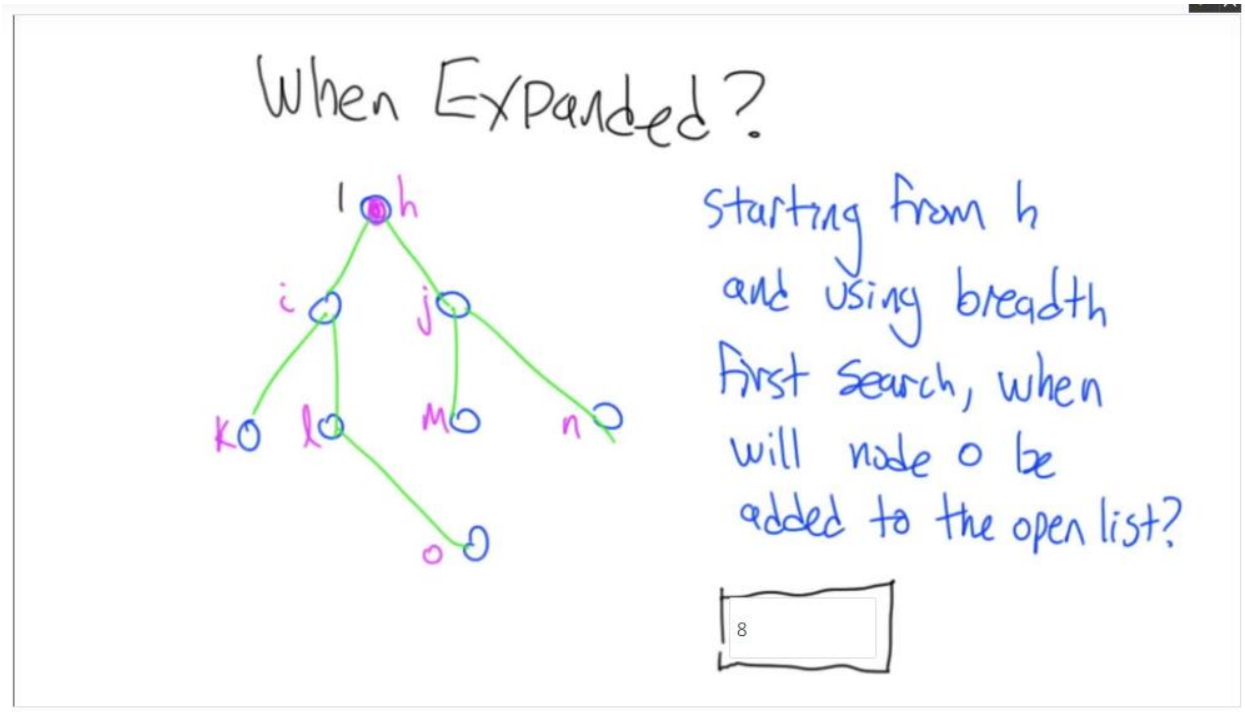
Checking Pairwise Connectivity

```
def check_connection(G, v1, v2):
    marked = {}
    mark_component (G,v1,marked)
    return v2 in marked
```

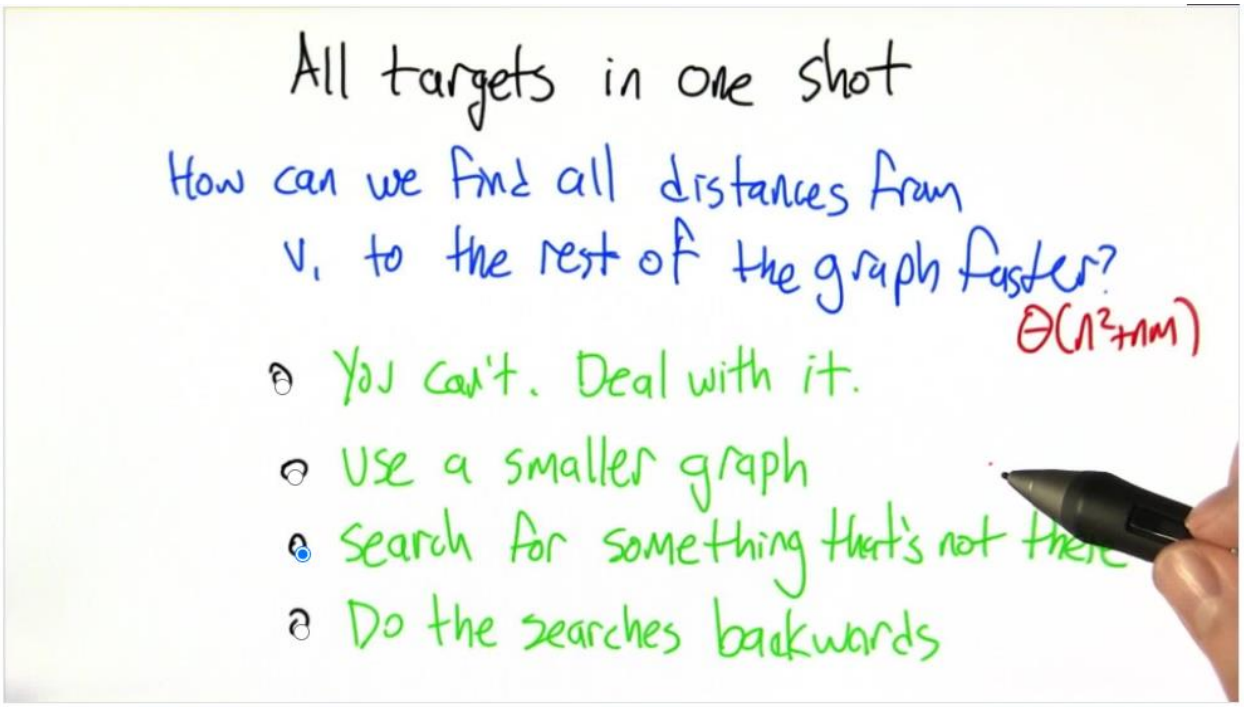
Depth First without Recursion



Searching a Tree

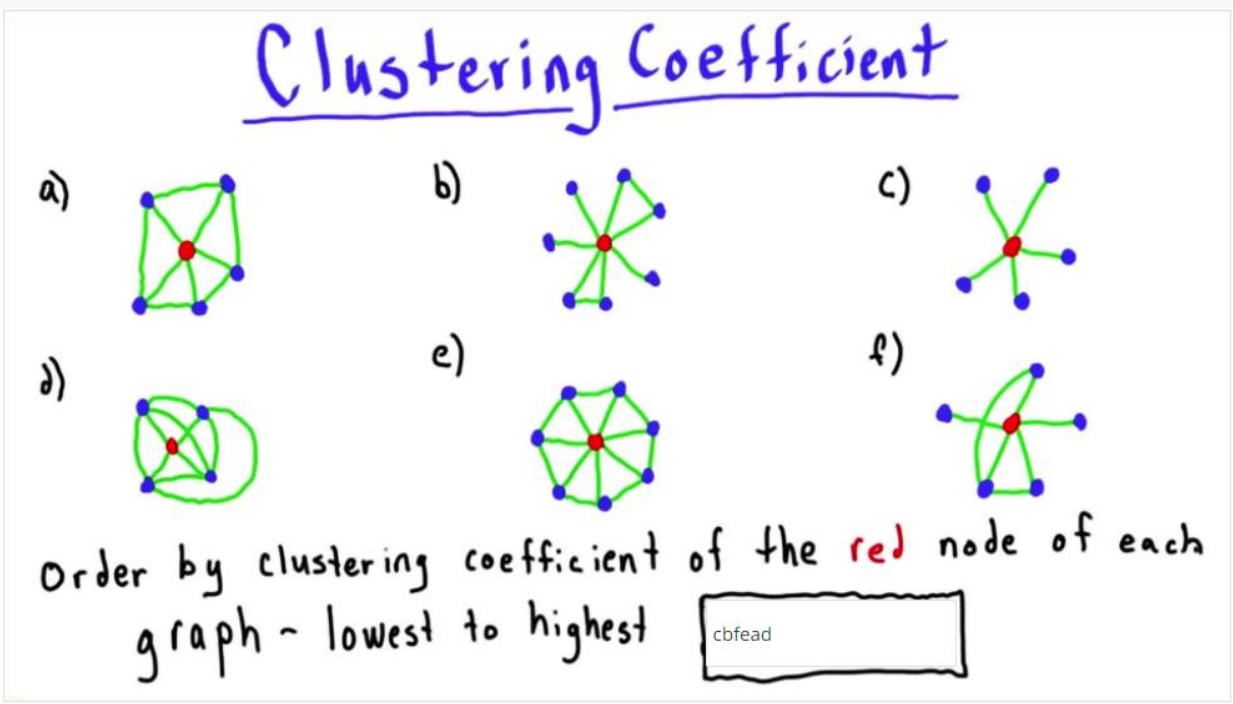


Single Source Shortest Paths

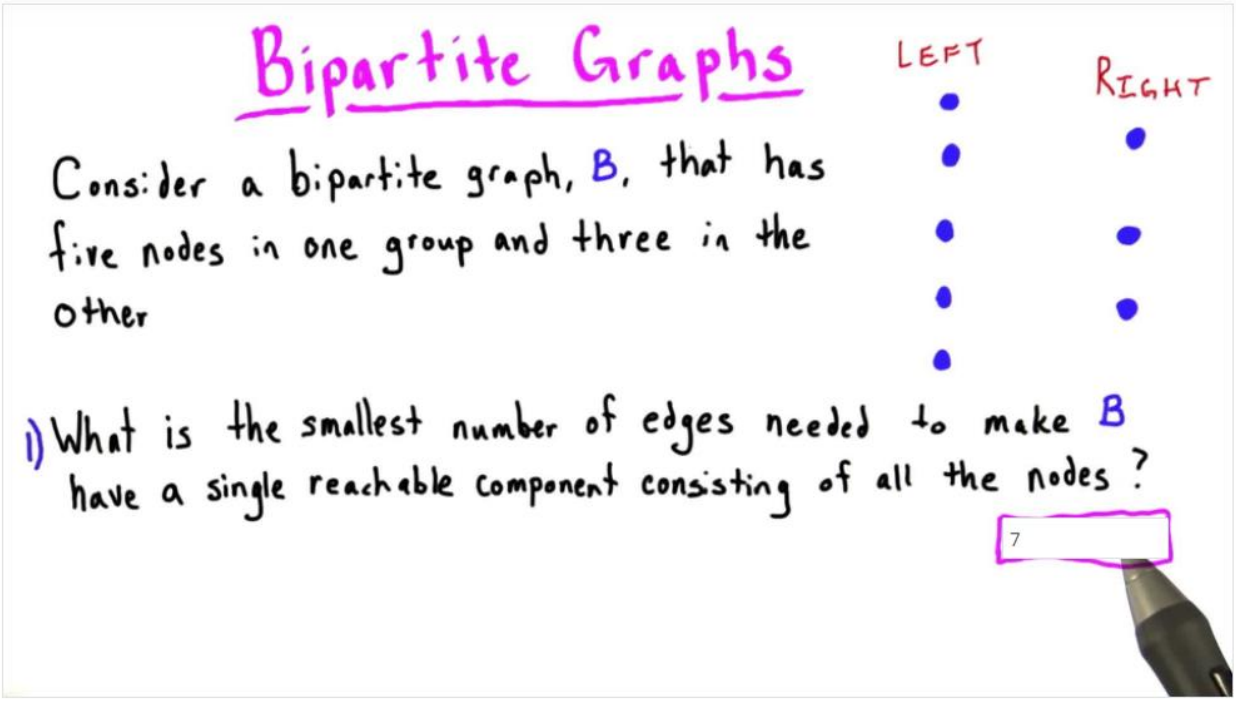


LECCIÓN 7

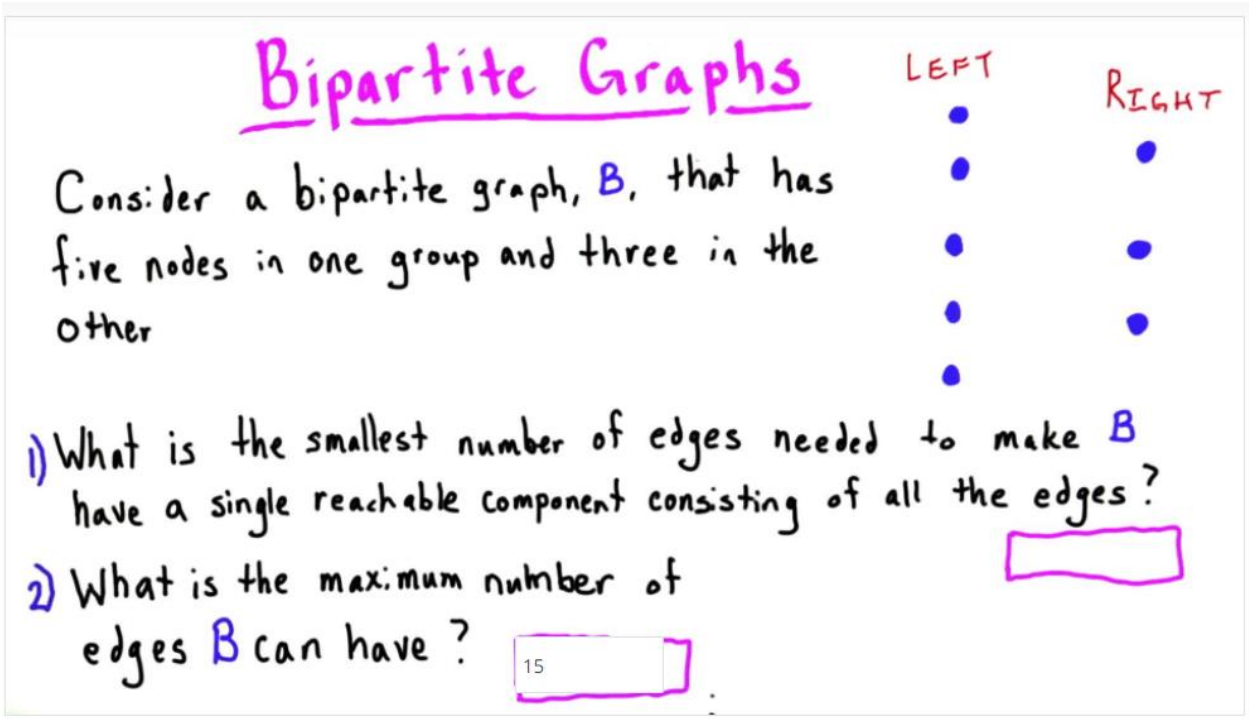
Clustering Coefficient



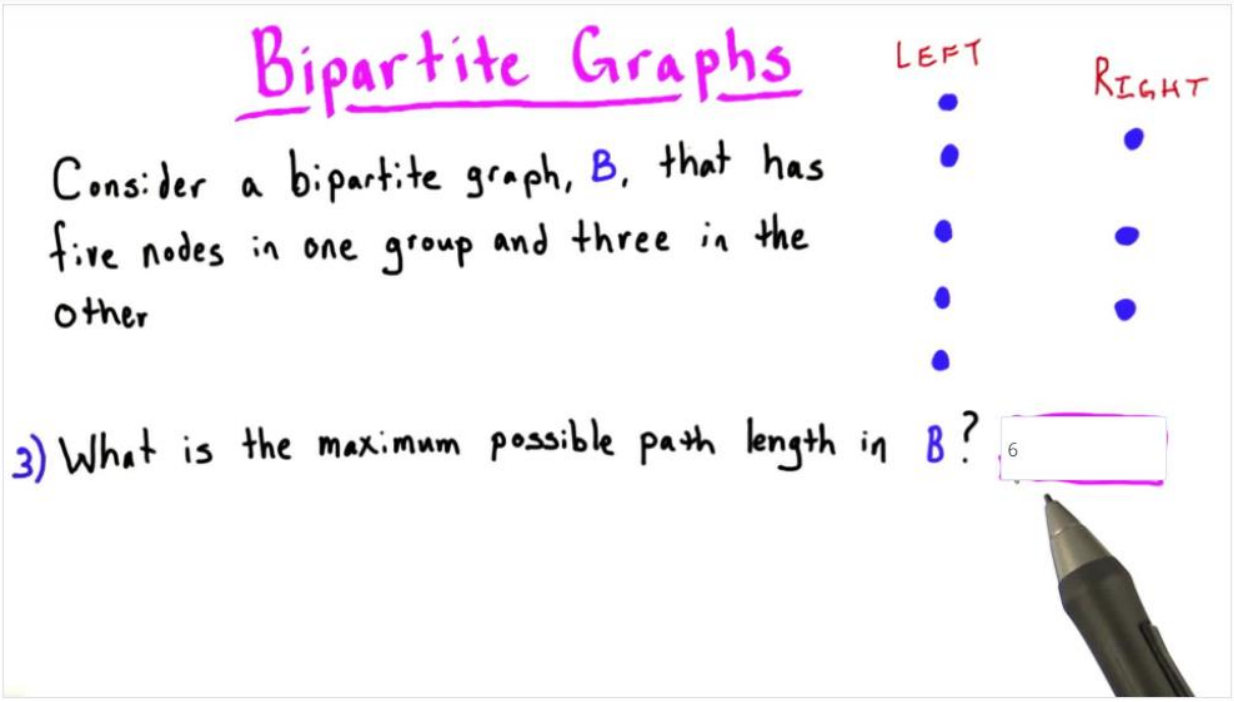
Bipartite I



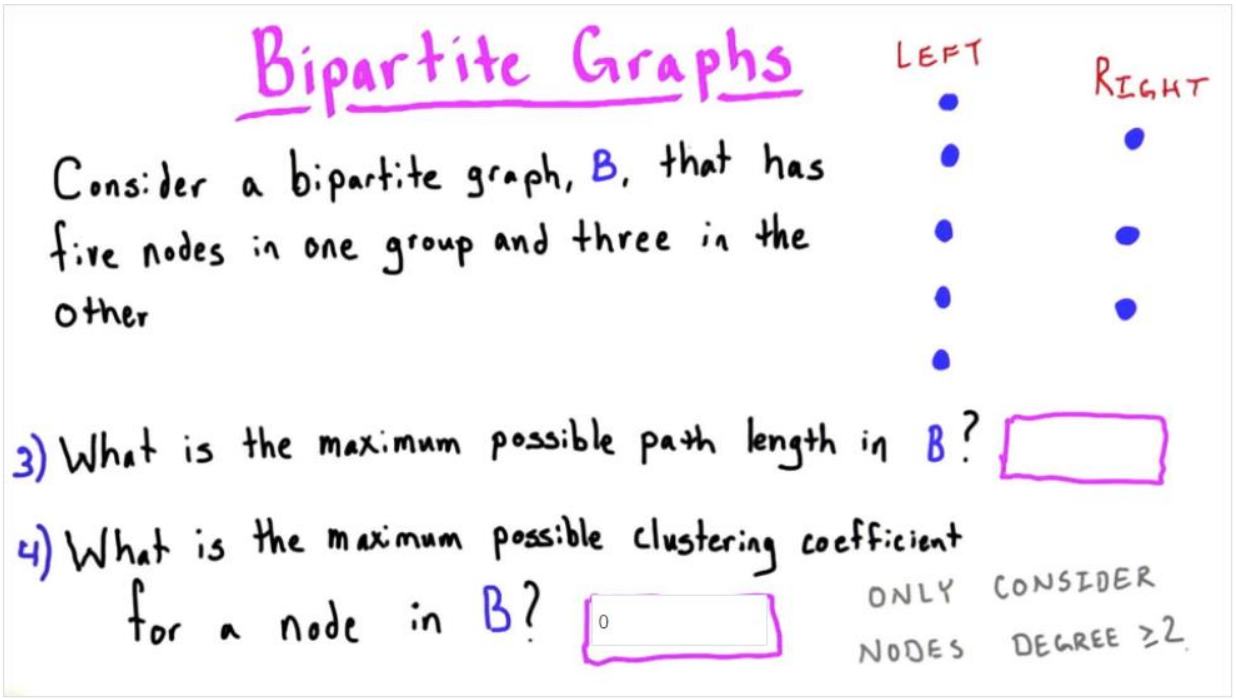
Bipartite II



Bipartite III



Bipartite IV



Mark Component

```
5
6  def mark_component(G, node, marked):
7      open_list = [node]
8      total_marked = 1
9      marked[node] = True
10     while len(open_list) > 0:
11         node = open_list.pop()
12         for neighbor in G[node]:
13             if neighbor not in marked:
14                 open_list.append(neighbor)
15                 marked[neighbor] = True
16                 total_marked += 1
17     return total_marked
18
```

Centrality

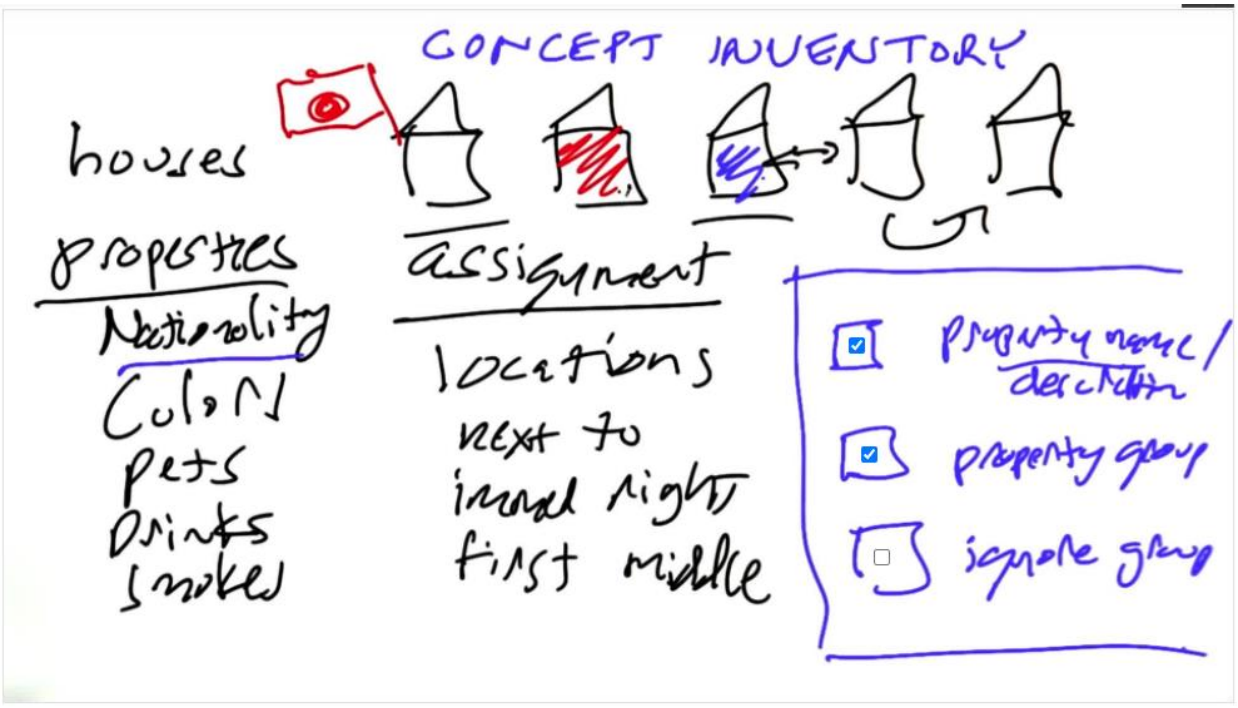
```python
def centrality_max(G, v):
    distance_from_start = {}
    open_list = [v]
    distance_from_start[v] = 0
    while len(open_list) > 0:
        current = open_list[0]
        del open_list[0]
        for neighbor in G[current].keys():
            if neighbor not in distance_from_start:
                distance_from_start[neighbor] = distance_from_start[current] + 1
                open_list.append(neighbor)
    return max(distance_from_start.values())

```

Bridge Edges

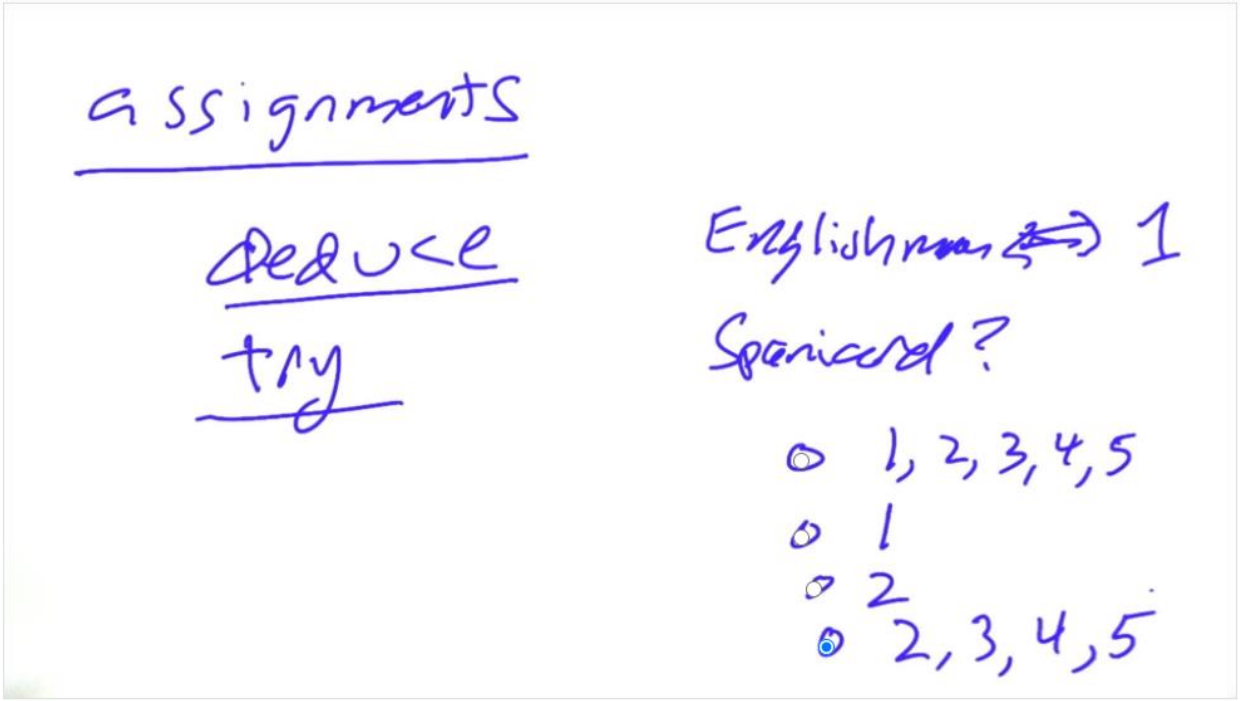
```python
#
# First some utility functions
#

def make_link(G, node1, node2, r_or_g):
    # modified make_link to apply
    # a color to the edge instead of just 1
    if node1 not in G:
        G[node1] = {}
    (G[node1])[node2] = r_or_g
    if node2 not in G:
        G[node2] = {}
    (G[node2])[node1] = r_or_g
    return G

def get_children(S, root, parent):
    """returns the children from following the
    green edges"""
    return [n for n, e in S[root].items()
            if ((not n == parent) and
                (e == 'green'))]

def get_children_all(S, root, parent):
    """returns the children from following
    green edges and the children from following
    red edges"""
    green = []
    red = []
    for n, e in S[root].items():
```

LECCIÓN 8

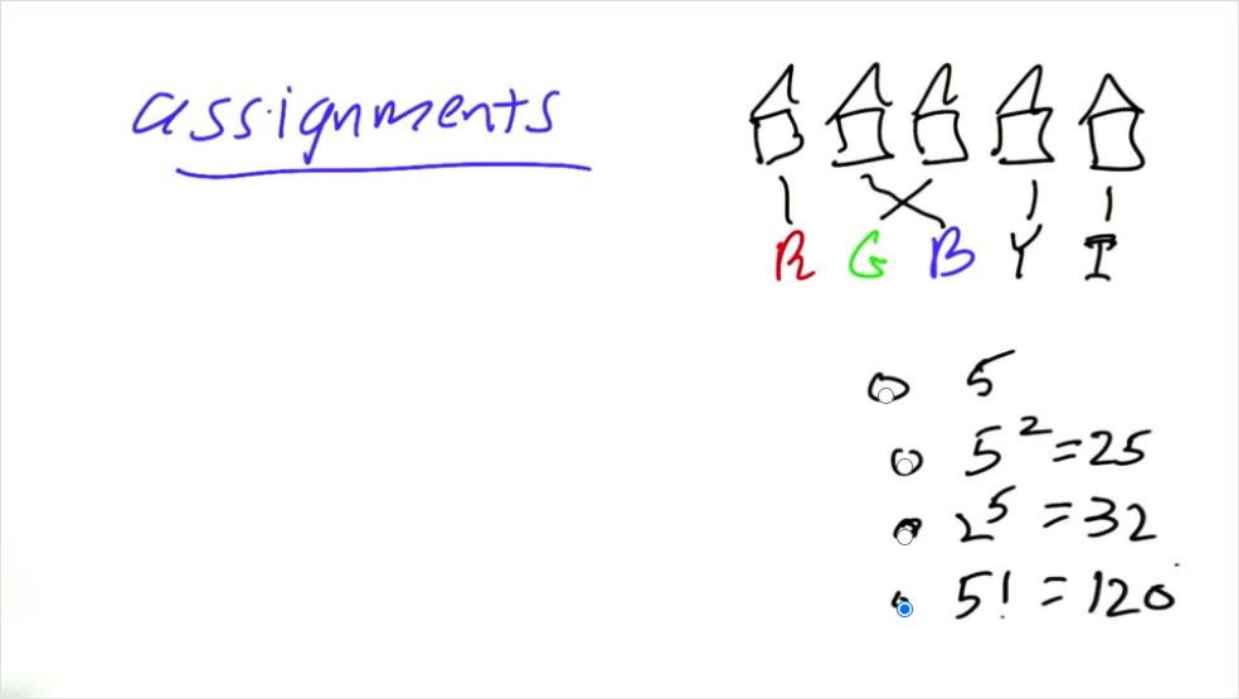
Zebra Puzzle



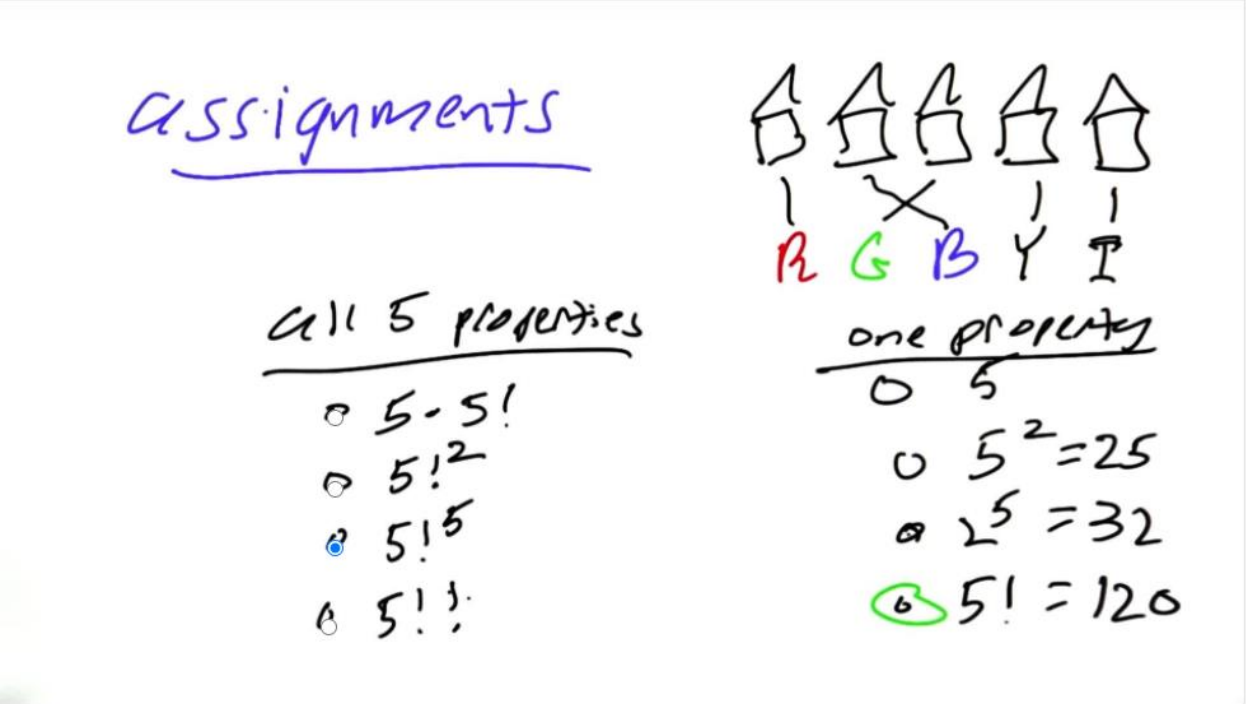
Where The Spaniard



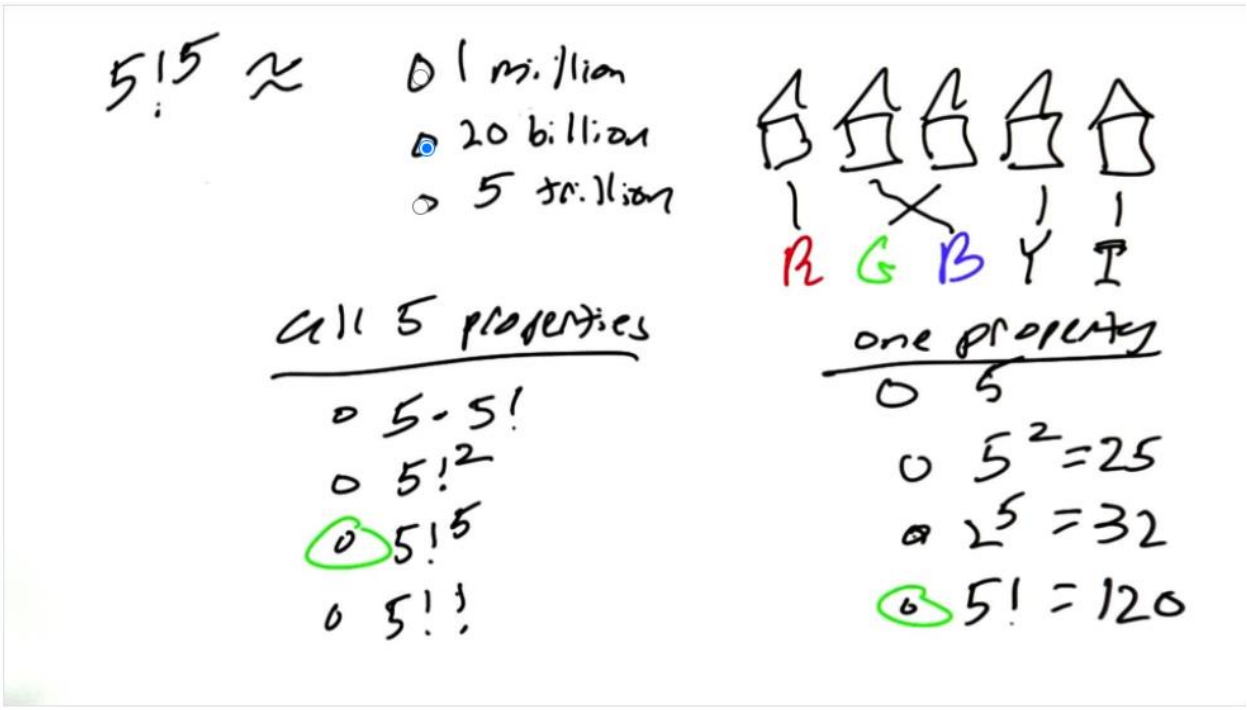
Counting Assignments



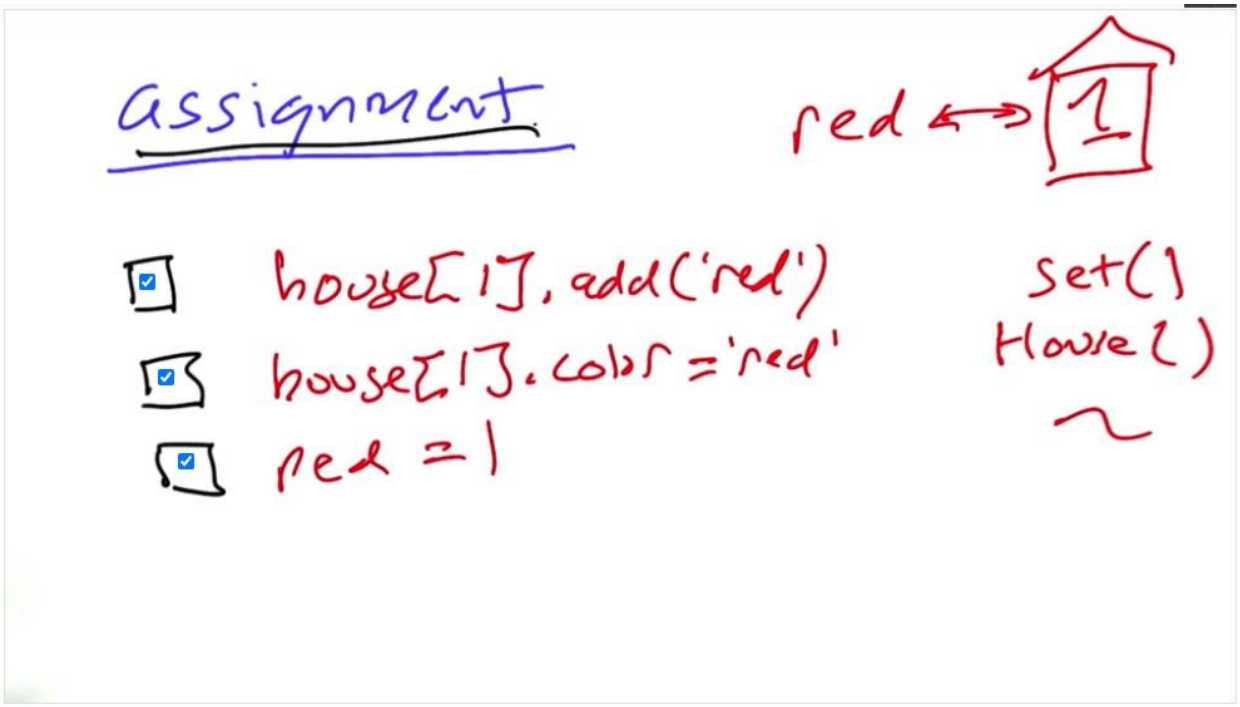
Multiple Properties



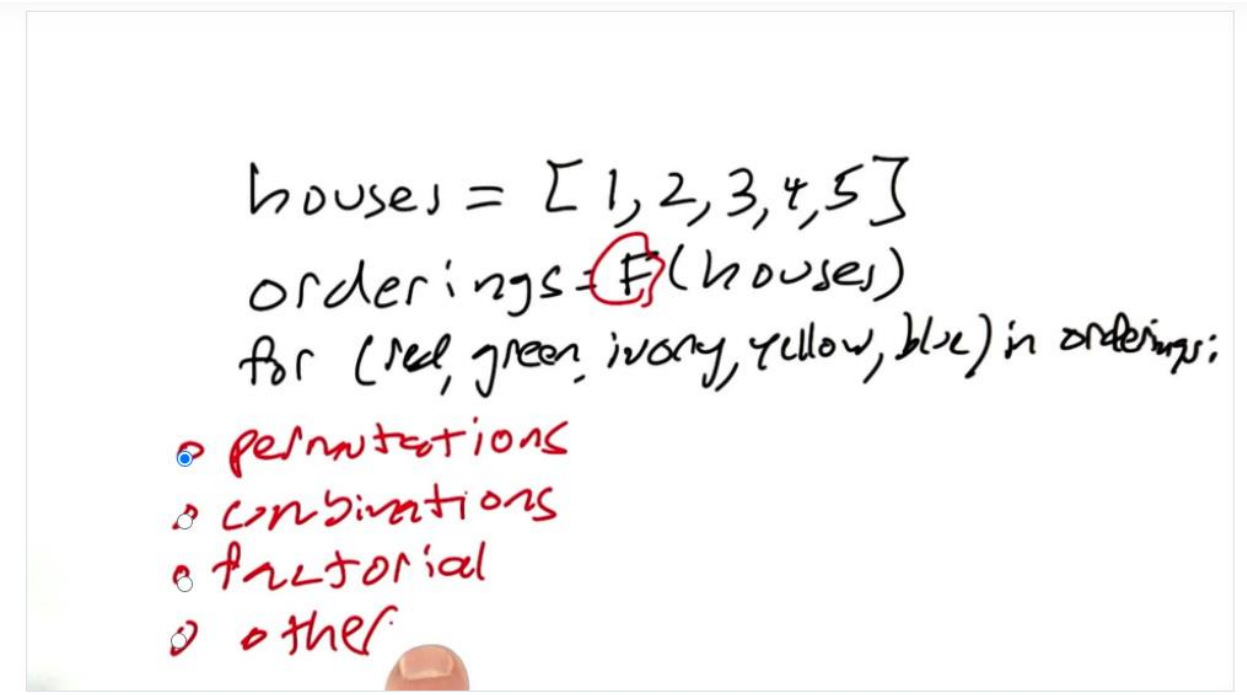
Back of the Envelope



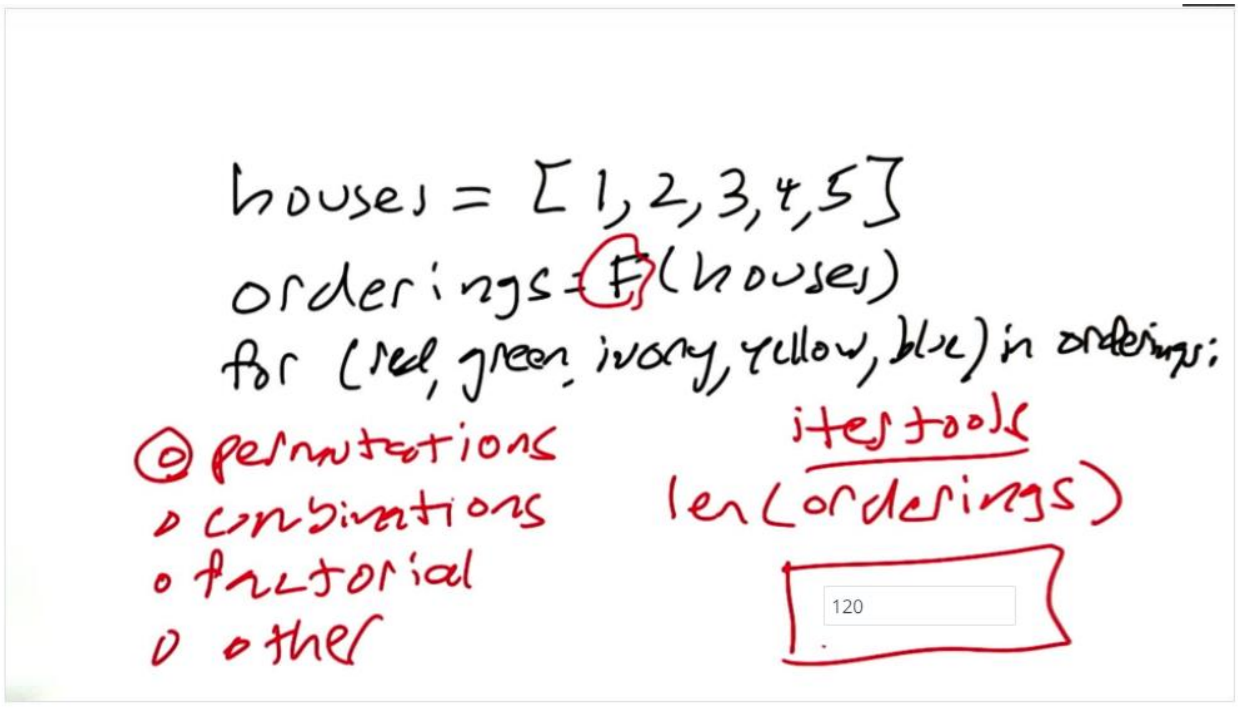
Leaving Happy Valley



Ordering Houses



Length Of Orderings



Estimating Runtime

```
1
2
3
4  import itertools
5
6  houses = [1, 2, 3, 4, 5]
7  orderings = list(itertools.permutations(houses))
8
9  for (red, green, ivory, yellow, blue) in orderings:
10   for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings:
11     for (dog, snails, fox, horse, ZEBRA) in orderings:
12       for (coffee, tea, milk, oj, WATER) in orderings:
13         for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings:
14           # constraints go here
15
16
17           # 1 sec
18           # 1 min
19           # 1 hour
20           # 1 day
```

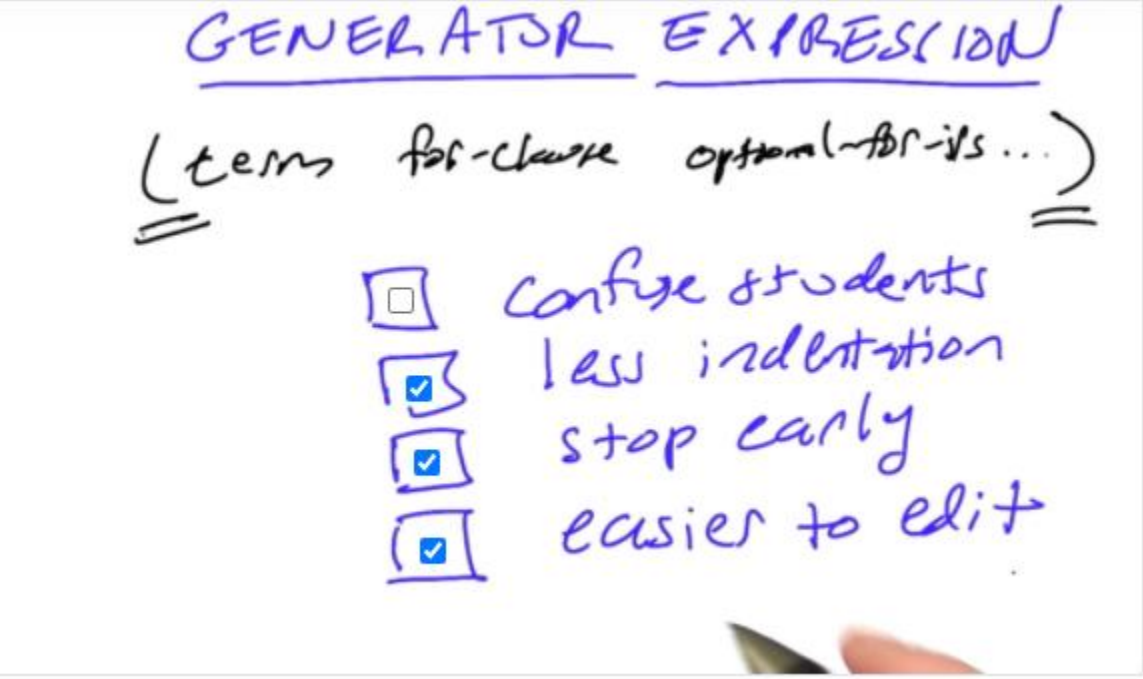
Red Englishman

```
4  import itertools
5
6  houses = [1, 2, 3, 4, 5]
7  orderings = list(itertools.permutations(houses))
8
9  for (red, green, ivory, yellow, blue) in orderings:
10   for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings:
11     for (dog, snails, fox, horse, ZEBRA) in orderings:
12       for (coffee, tea, milk, oj, WATER) in orderings:
13         for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings:
14           if ( Englishman == red ): #2
15
16
17           # 1 sec
18           # 1 min
19           # 1 hour
20           # 1 day
```

Neighbors

```
7
8  import itertools
9
10 houses = [1, 2, 3, 4, 5]
11 orderings = list(itertools.permutations(houses))
12
13 def imright(h1, h2):
14     "House h1 is immediately right of h2 if h1-h2 == 1."
15     return h1-h2 == 1
16
17 def nextto(h1, h2):
18     "Two houses are next to each other if they differ by 1."
19     return abs(h1-h2) == 1
20
```

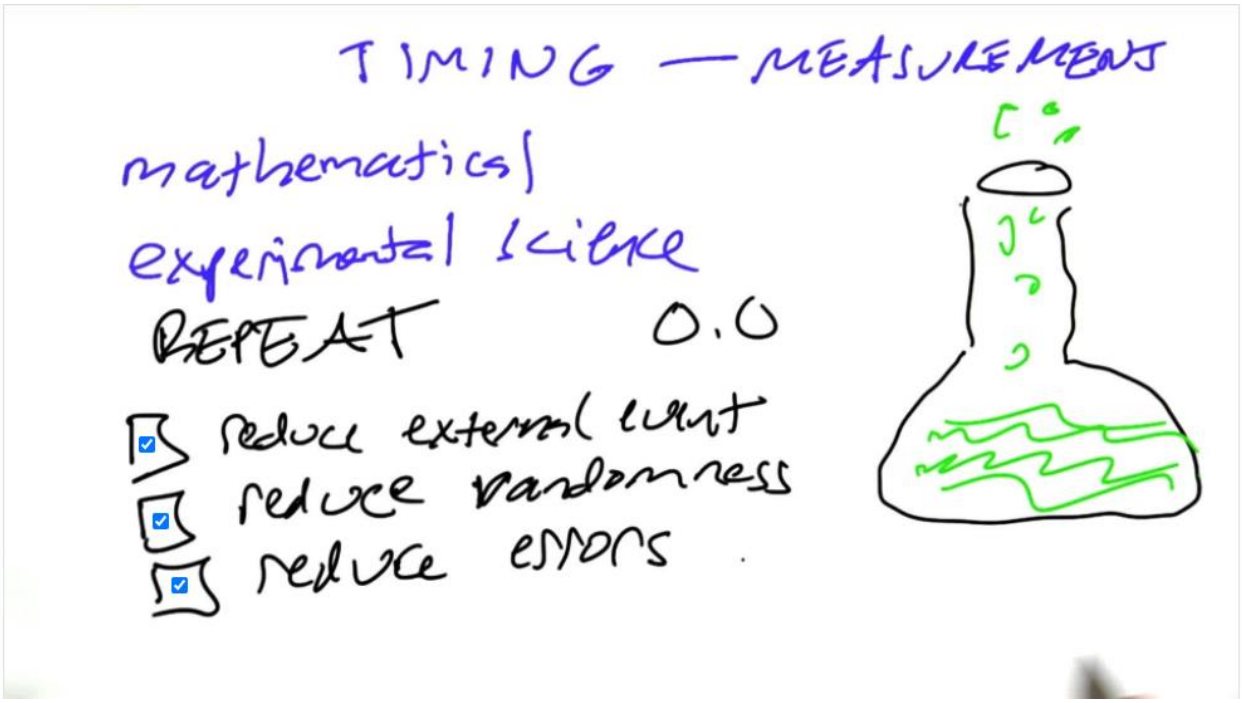
Generator Expressions



Eliminating Redundancy

```
def zebra_puzzle():
    "Return a tuple (WATER, ZEBRA) indicating their house numbers."
    houses = first, _, middle, _, _ = [1, 2, 3, 4, 5]
    orderings = list(itertools.permutations(houses))  #1
    return next((WATER, ZEBRA)
            for (red, green, ivory, yellow, blue) in orderings
            for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings
            if Englishman is red             #2
            for (dog, snails, fox, horse, ZEBRA) in orderings
            for (coffee, tea, milk, oj, WATER) in orderings
            for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings
            if Spaniard is dog               #3
```

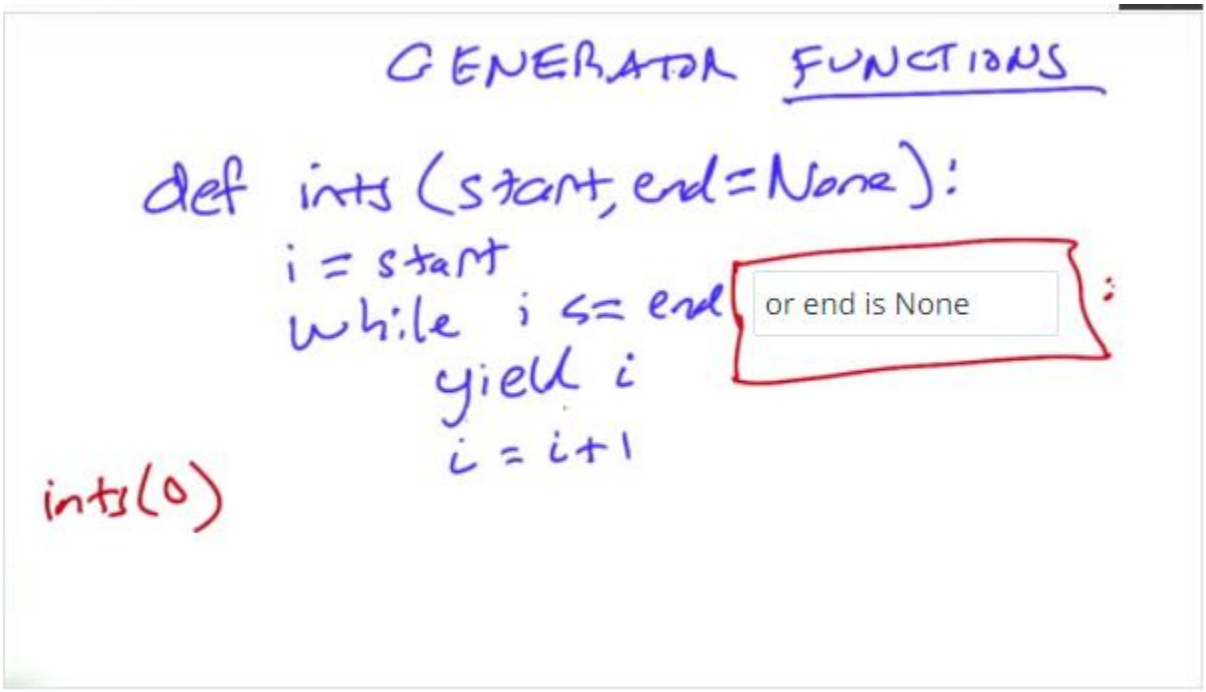
Good Science



Timed Calls

```
6
7  def timedcalls(n, fn, *args):
8      """Call fn(*args) repeatedly: n times if n is an int, or up to
9      n seconds if n is a float; return the min, avg, and max time"""
0      if isinstance(n, int):
1          times = [timedcall(fn, *args)[0] for _ in range(n)]
2      else:
3          times = []
4          while sum(times) < n:
5              times.append(timedcall(fn, *args)[0])
6      return min(times), average(times), max(times)
7
```

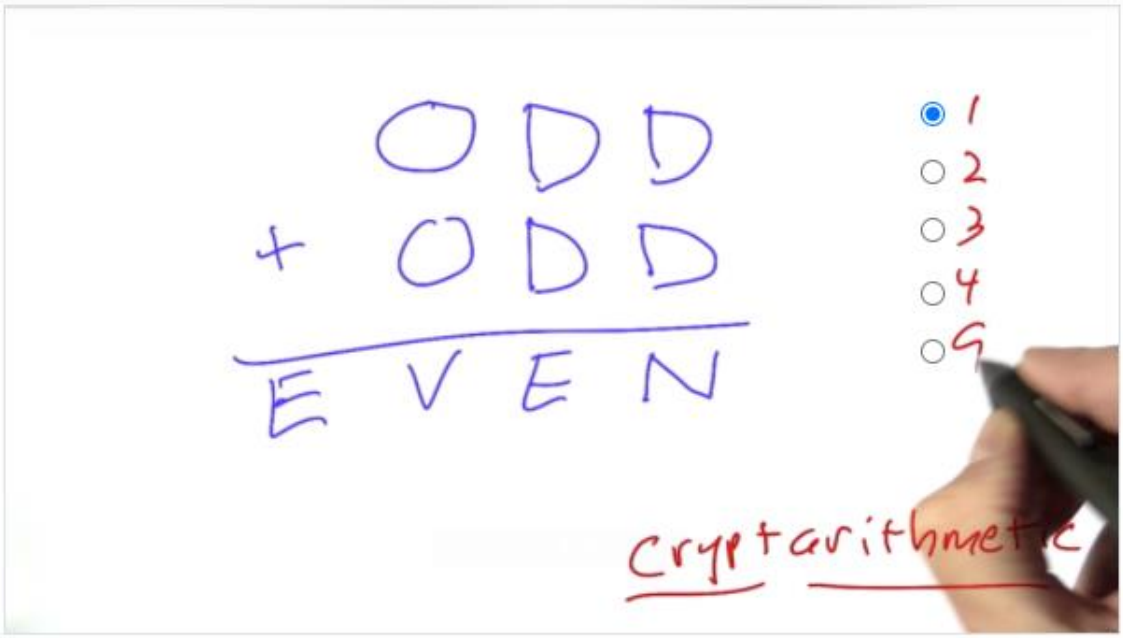
Yeilding Results



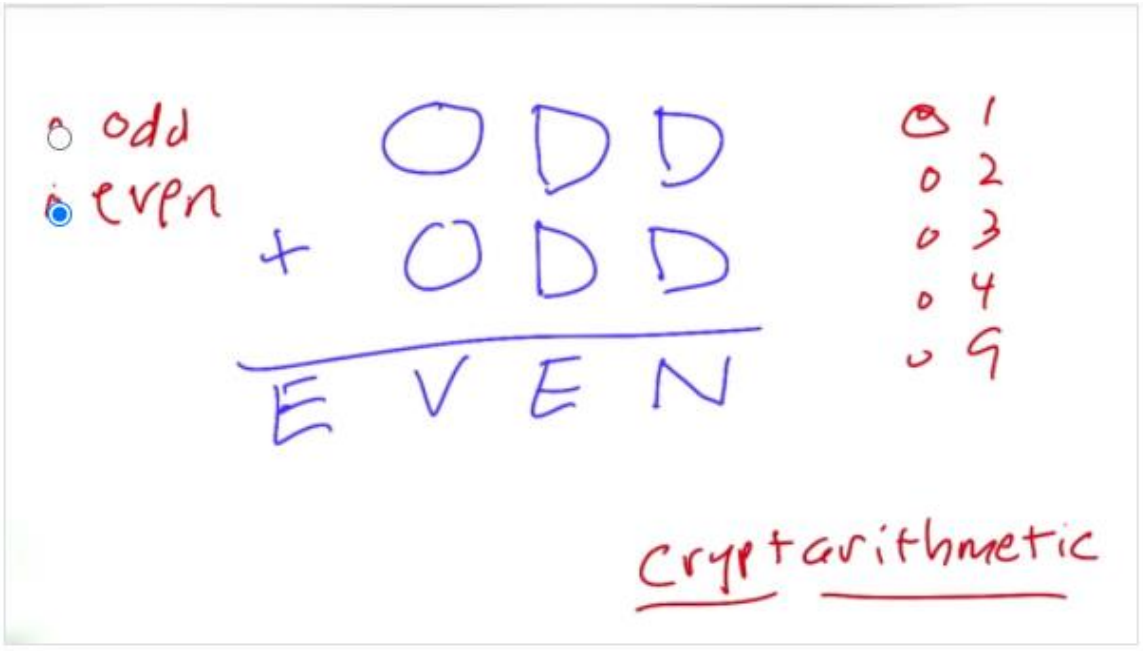
All Ints

```
14 def all_ints():
15     "Generate integers in the order 0, +1, -1, +2, -2, +3, -3, ..."
16     yield 0
17     for i in ints(1):
18         yield +i
19         yield -i
```

Cryptarithmetic



Odd or Even



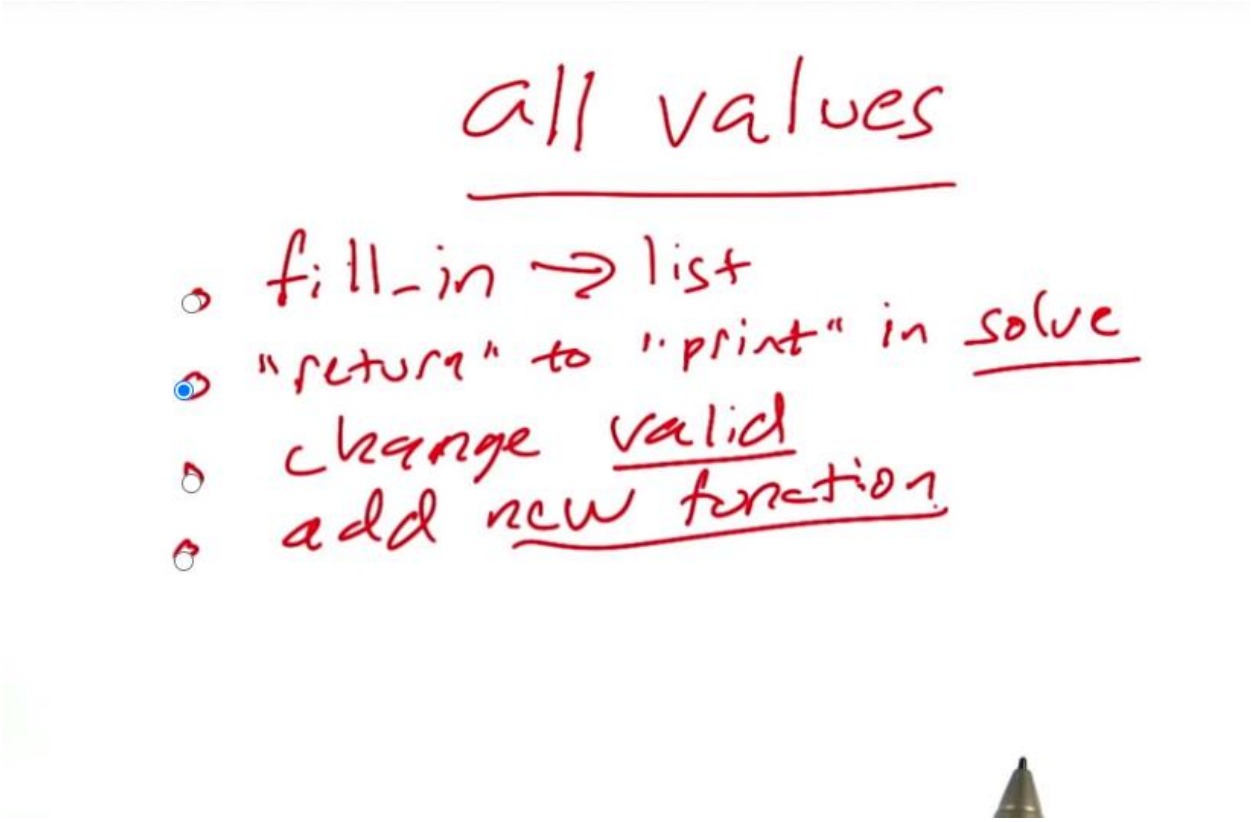
Solving Cryptarithmetic

```
def solve(formula):
    """Given a formula like 'ODD + ODD == EVEN', fill in digits to
    Input formula is a string; output is a digit-filled-in string
    for f in fill_in(formula):
        if valid(f):
            return f
```

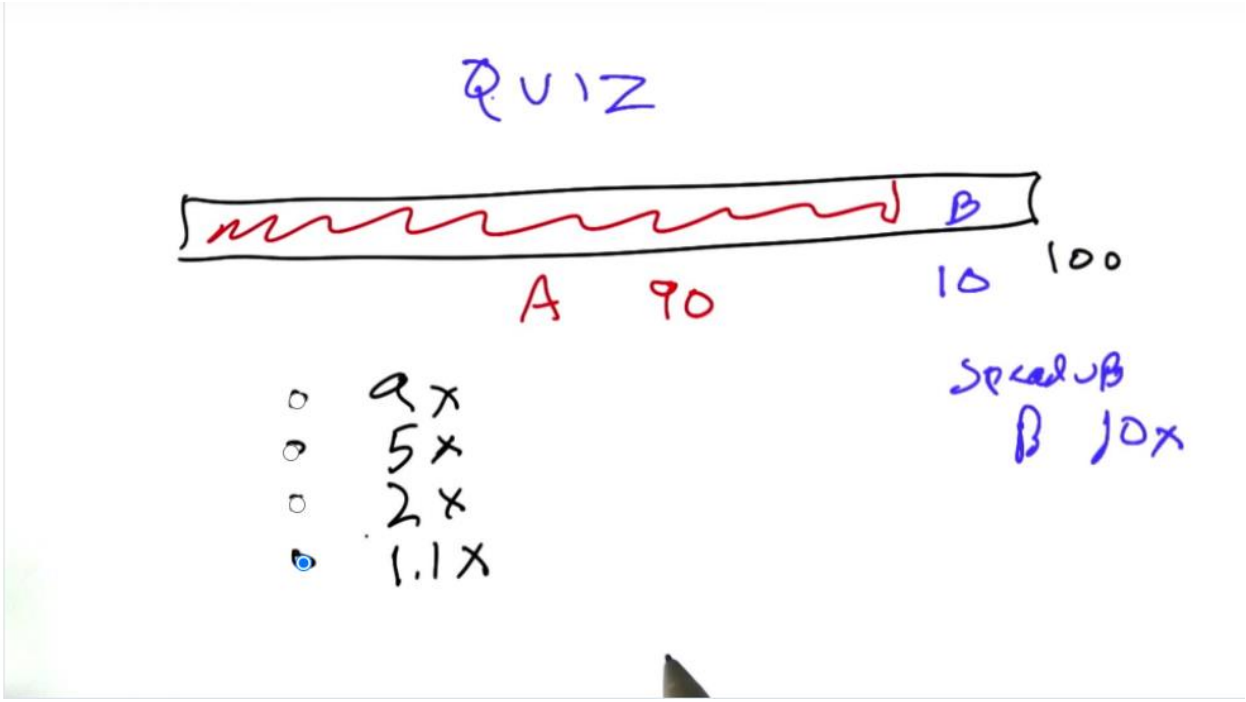
Filling In Fill In

```
def fill_in(formula):
    "Generate all possible fillings-in of letters in formula with digits."
    letters = ''.join(set(re.findall('[A-Z]', formula))) #should be a string
    for digits in itertools.permutations('1234567890', len(letters)):
        table = string.maketrans(letters, ''.join(digits))
        yield formula.translate(table)
```

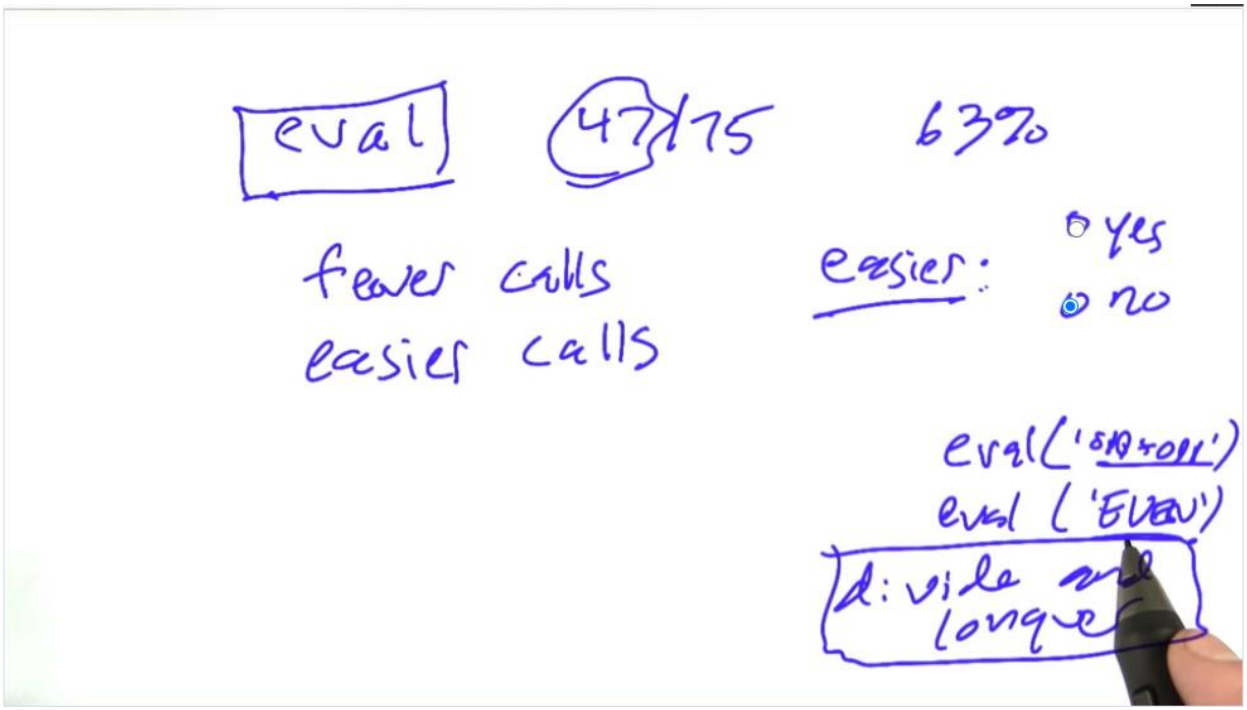
Find All Values



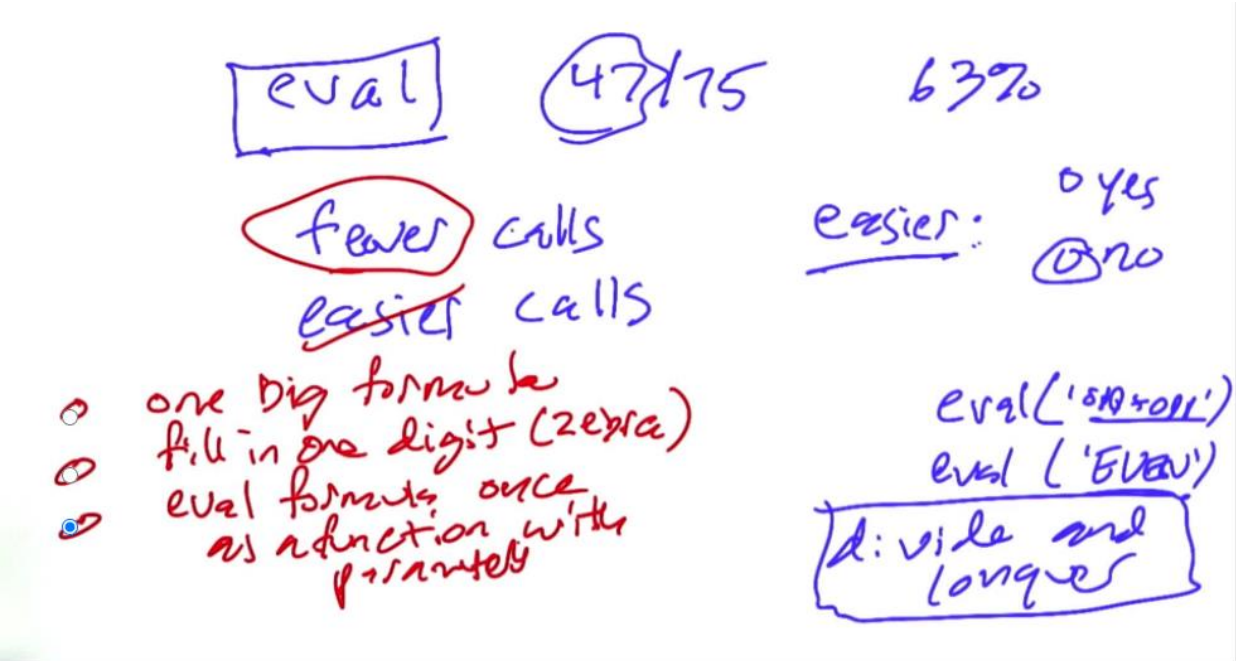
Increasing Speed



Rethinking Eval



Making Fewer Calls

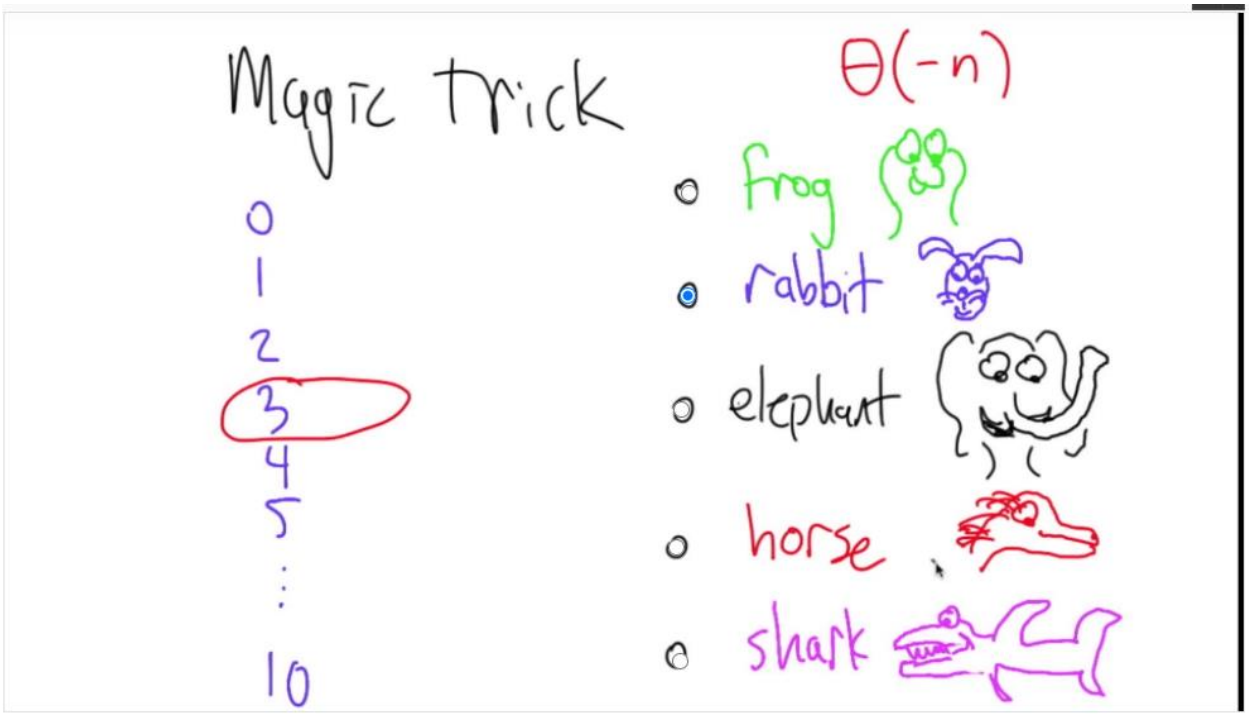


Compile World

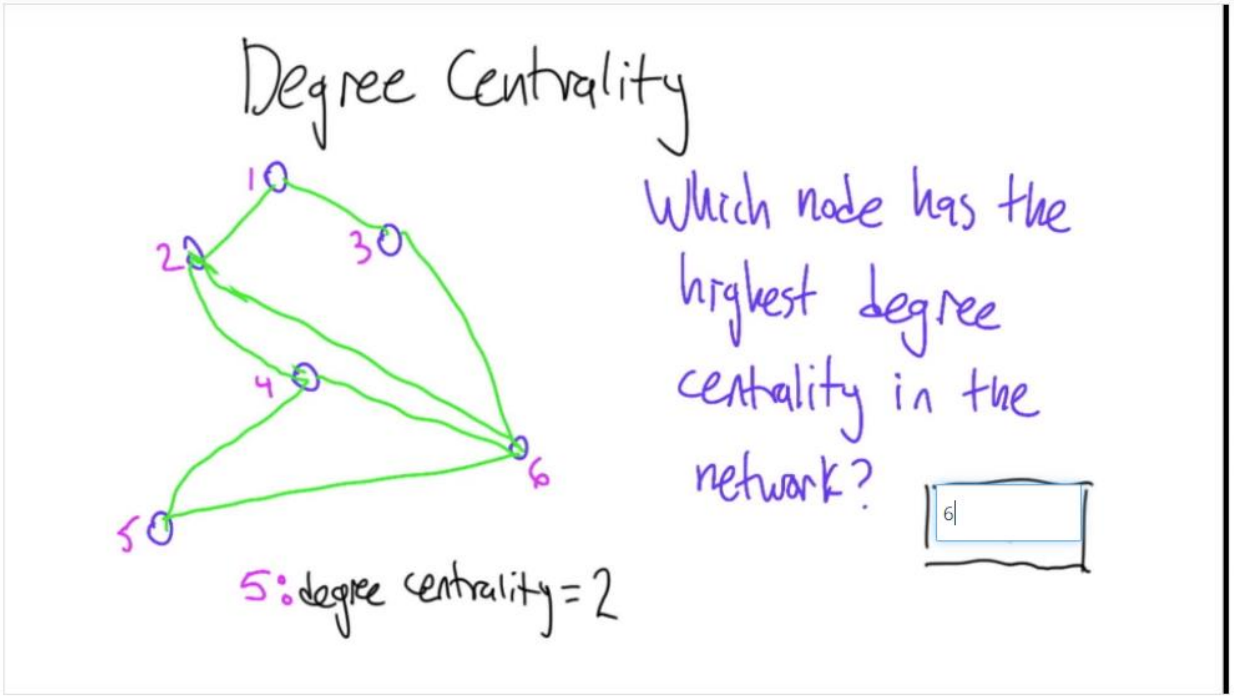
```
def compile_word(word):
    """Compile a word of uppercase letters as numeric digits.
    E.g., compile_word('YOU') => '(1*U+10*O+100*Y)'
    Non-uppercase words unchanged: compile_word('+') => '+'"""
    if word.isupper():
        terms = [('%s*%s' % (10**i, d))
                 for (i, d) in enumerate(word[::-1])]
        return '('+ '+' .join(terms) + ')'
    else:
        return word
```

LECCIÓN 9

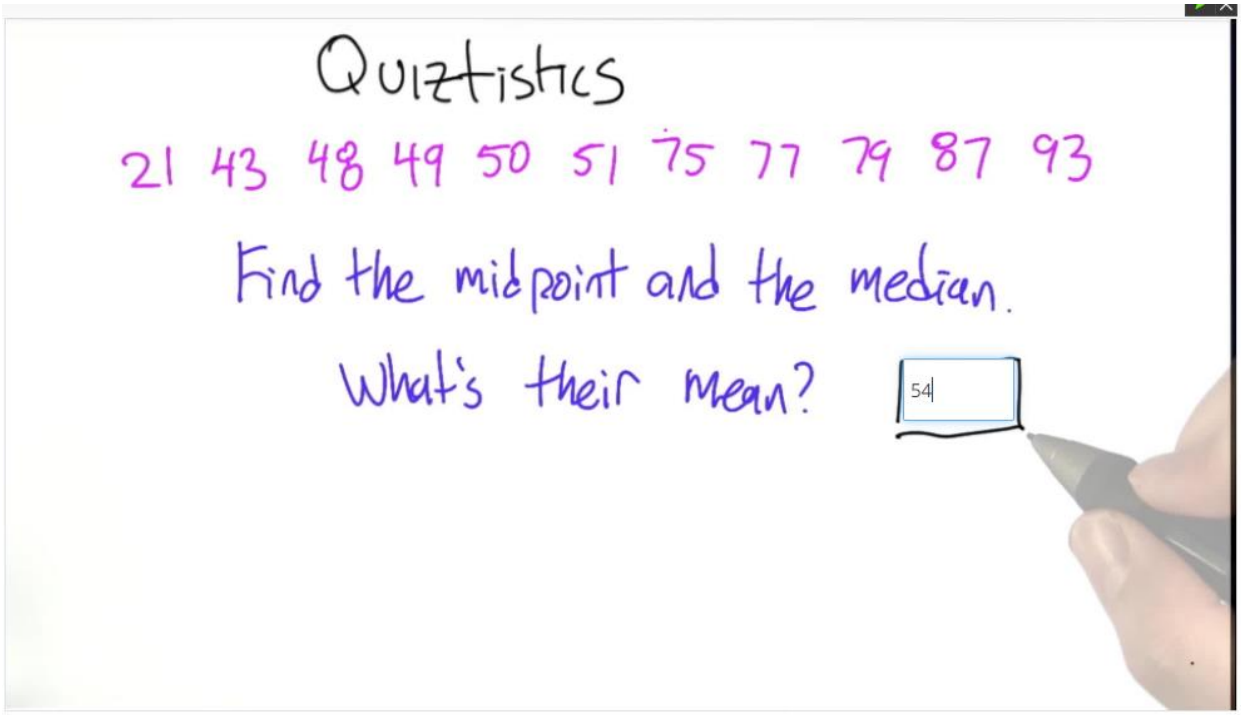
Introduction



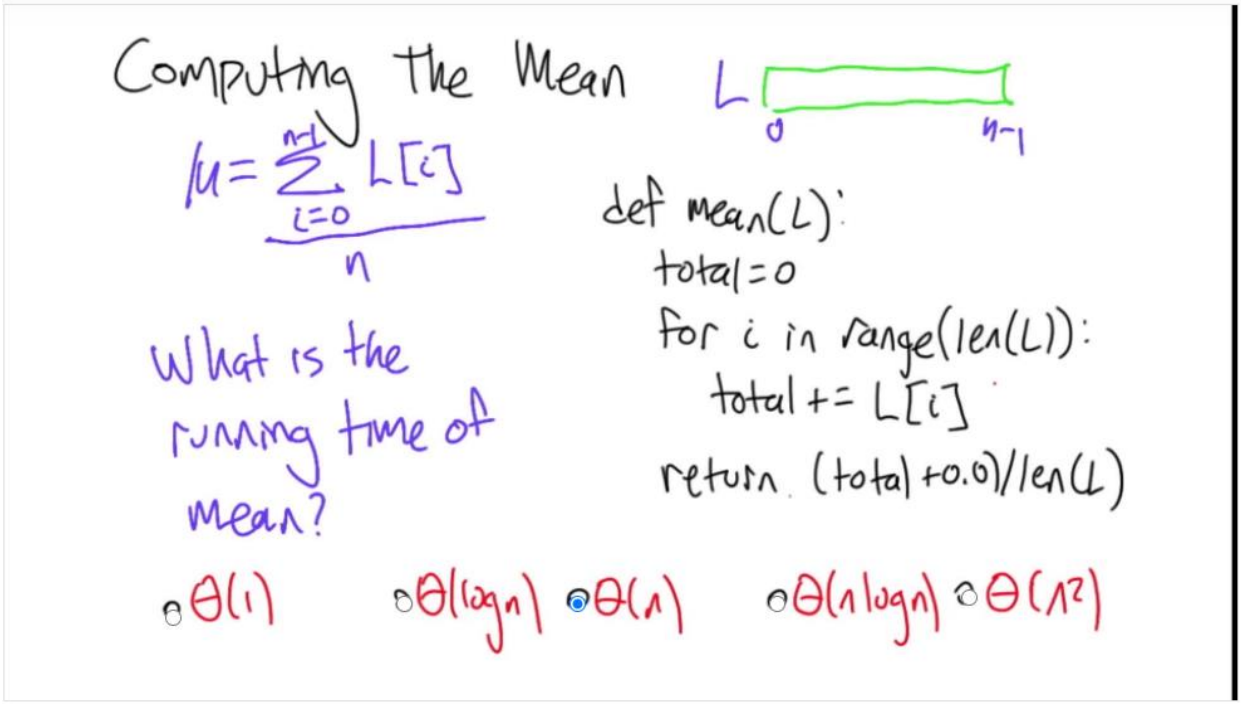
Degree Centrality



Statistics By Sorting



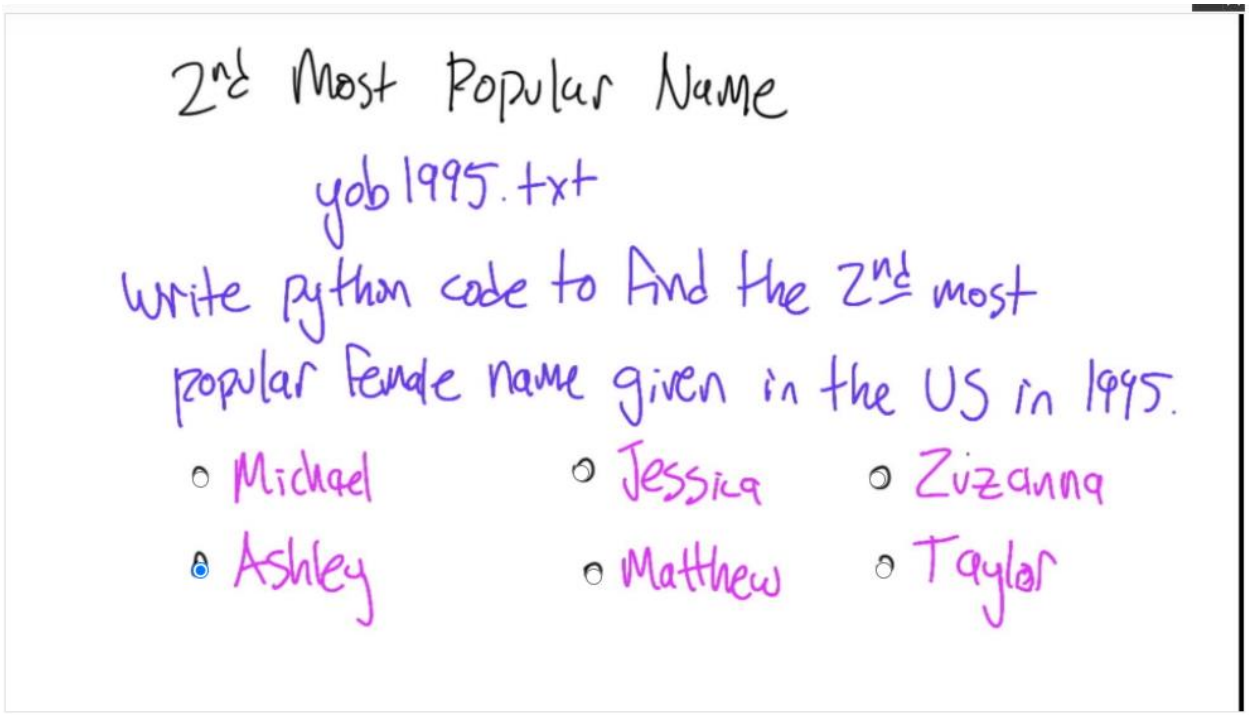
Mean



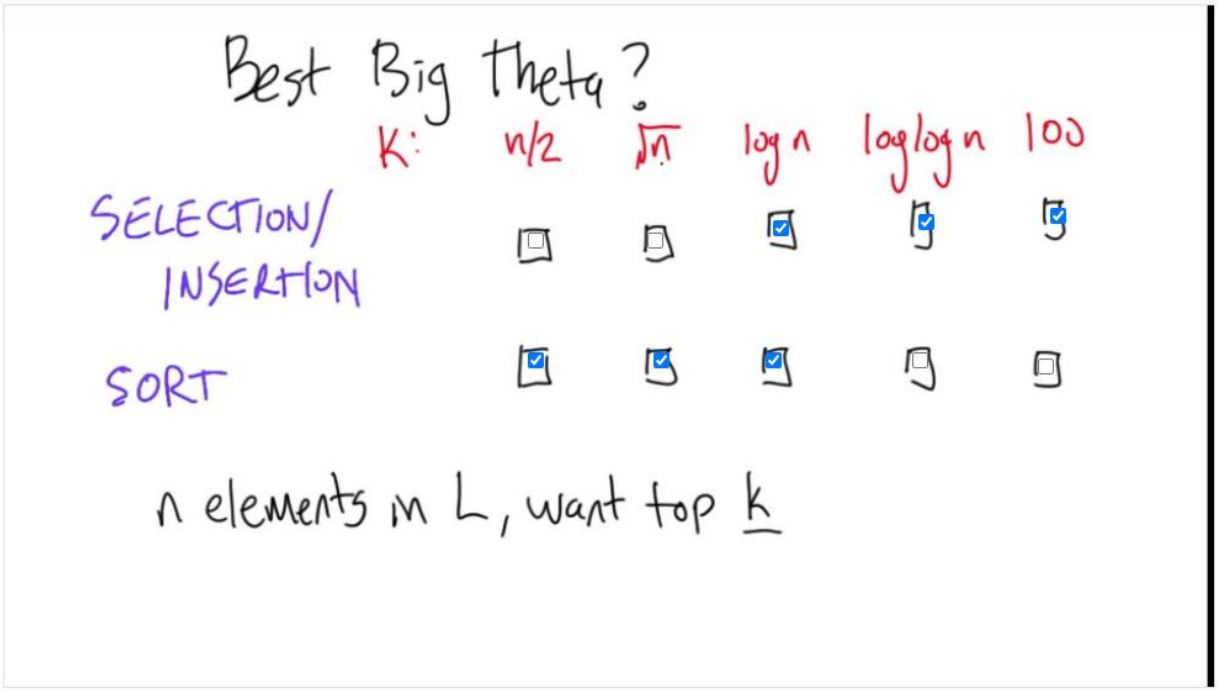
Extreme Values

```
#
# Write `max`
#

def max(L):
    max_so_far = L[0]
    for i in range(1,len(L)):
        if L[i] > max_so_far:
            max_so_far = L[i]
    return max_so_far

def test():
    L = [1, 2, 3, 4]
    assert 4 == max(L)
    L = [3, 6, 10, 9, 3]
    assert 10 == max(L)
```

Order Statistics



Top K Problem

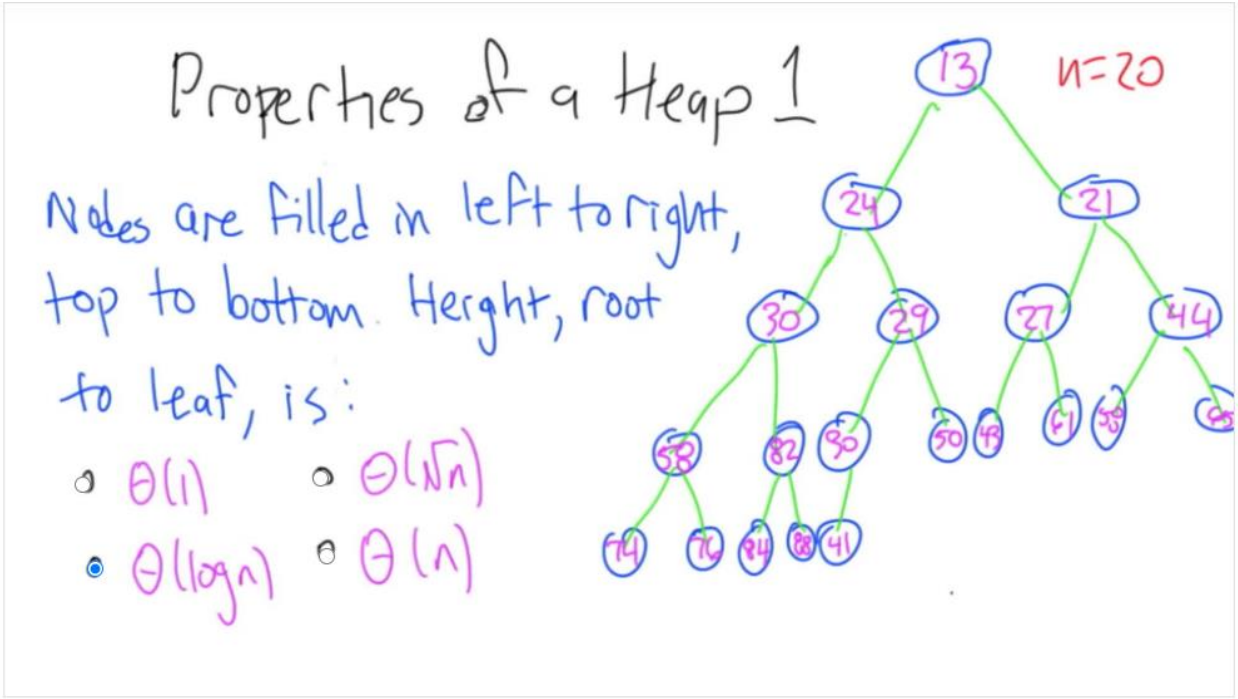


Partitioning Around V

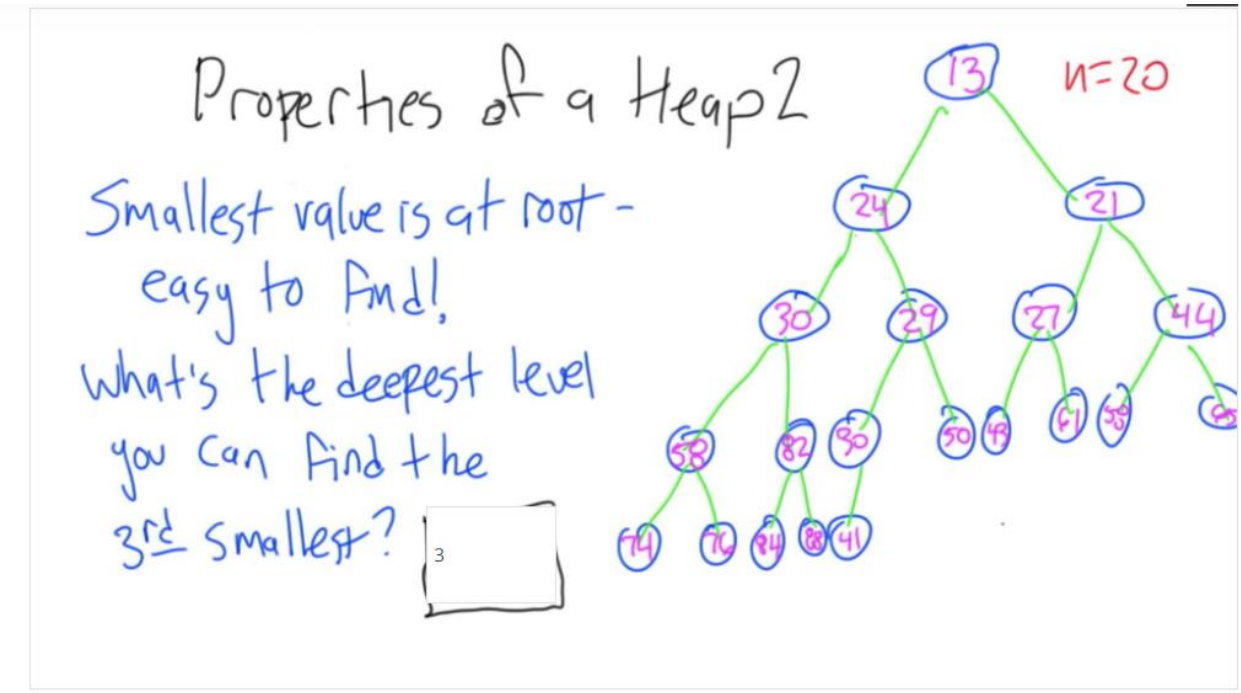
```
def partition(L, v):
    smaller = []
    bigger = []
    for each in L:
        if each < v:
            smaller.append(each)
        if each > v:
            bigger.append(each)
    # your code here
    return smaller+[v]+bigger
```

Heaps of Fun

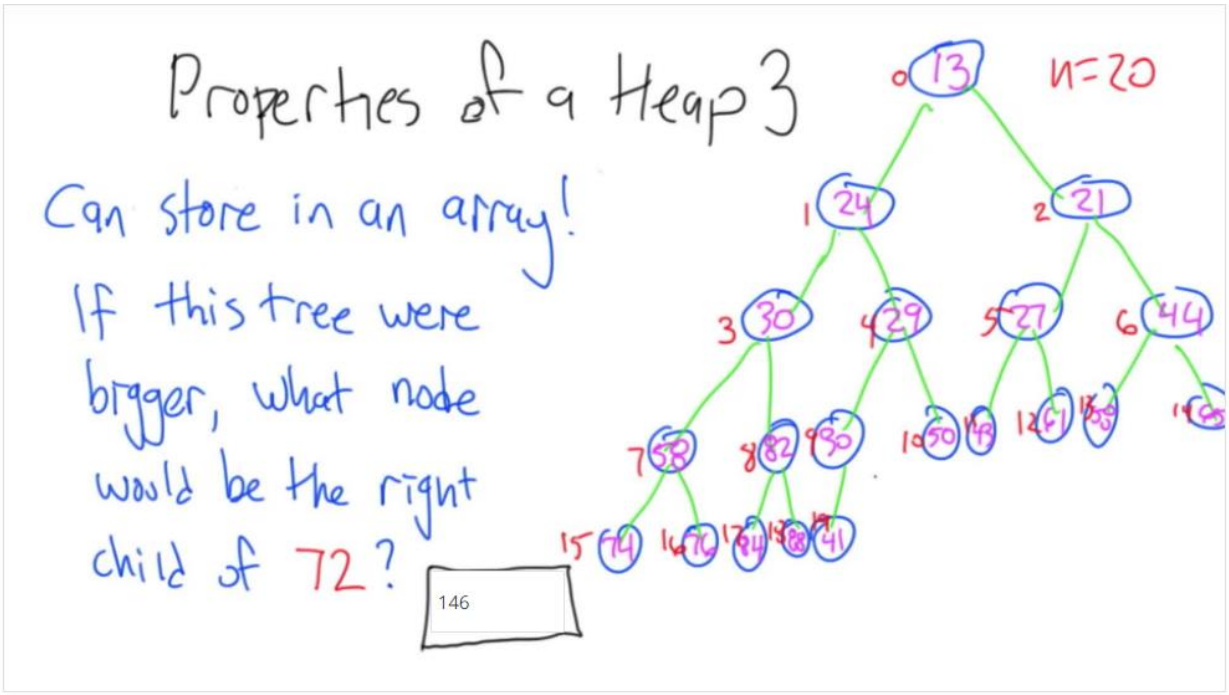
Heap Height



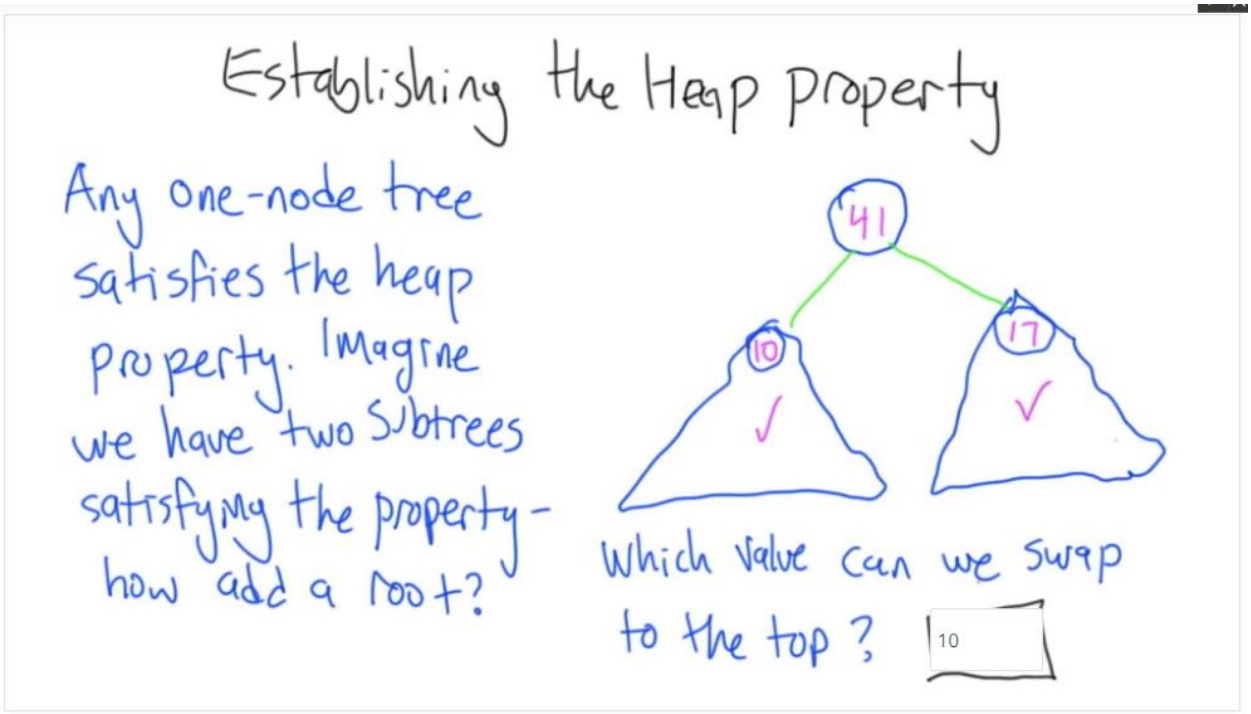
Properties of a Heap



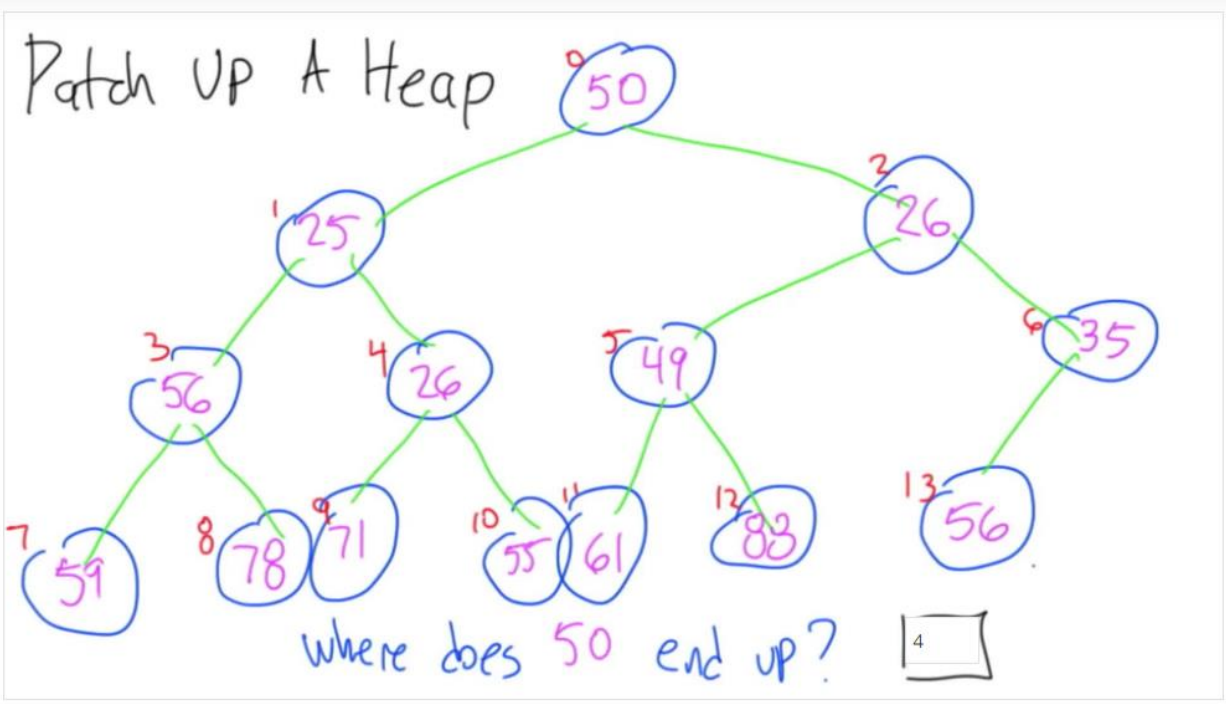
Heap Number



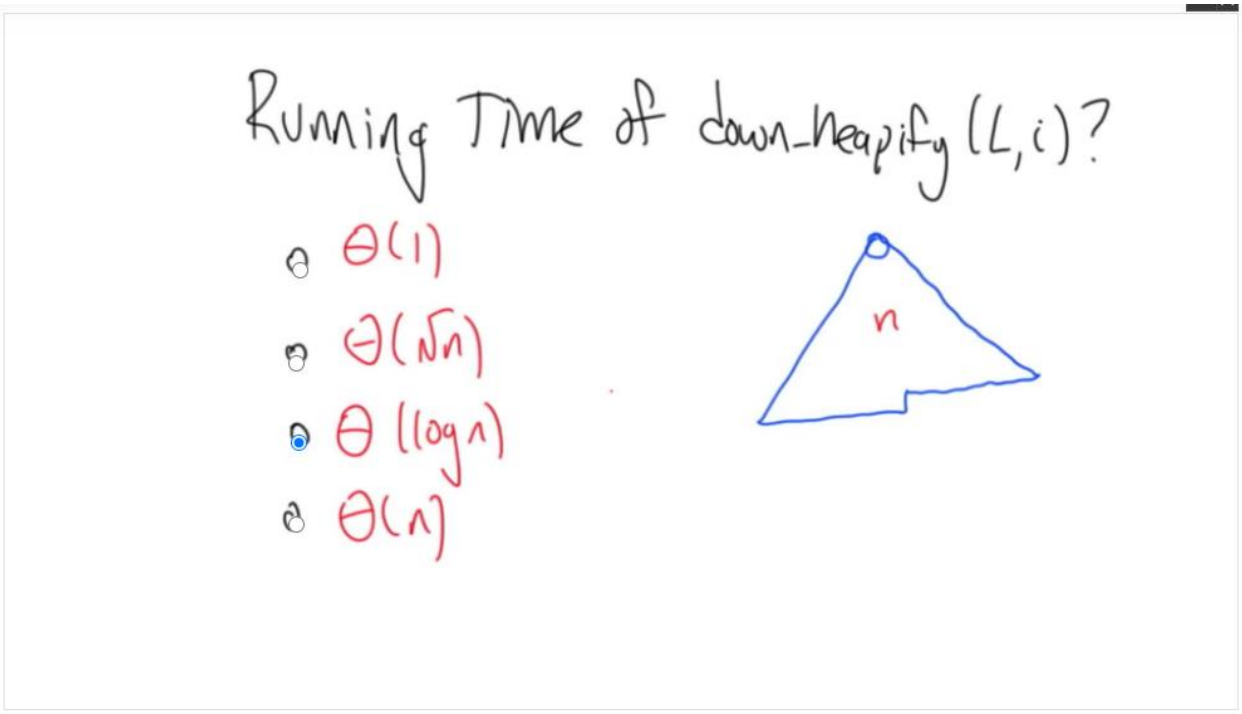
Establishing the Heap Property



Patch Up A Heap



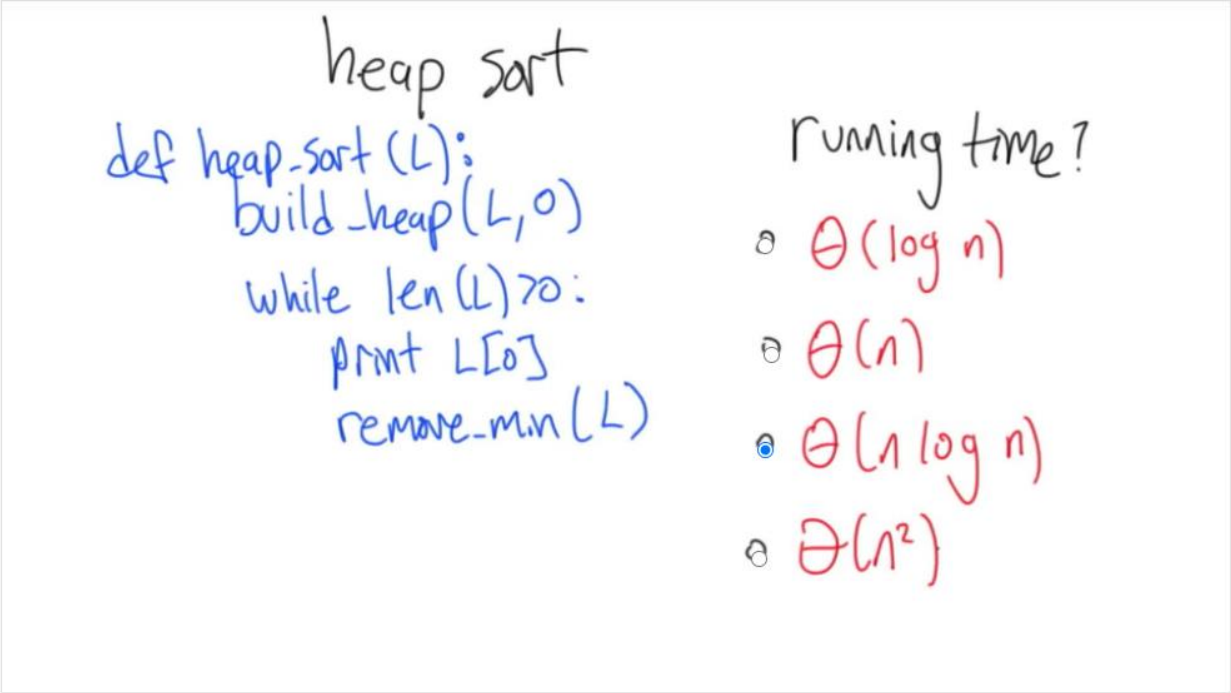
Running Time of Down Heapify



Remove Min

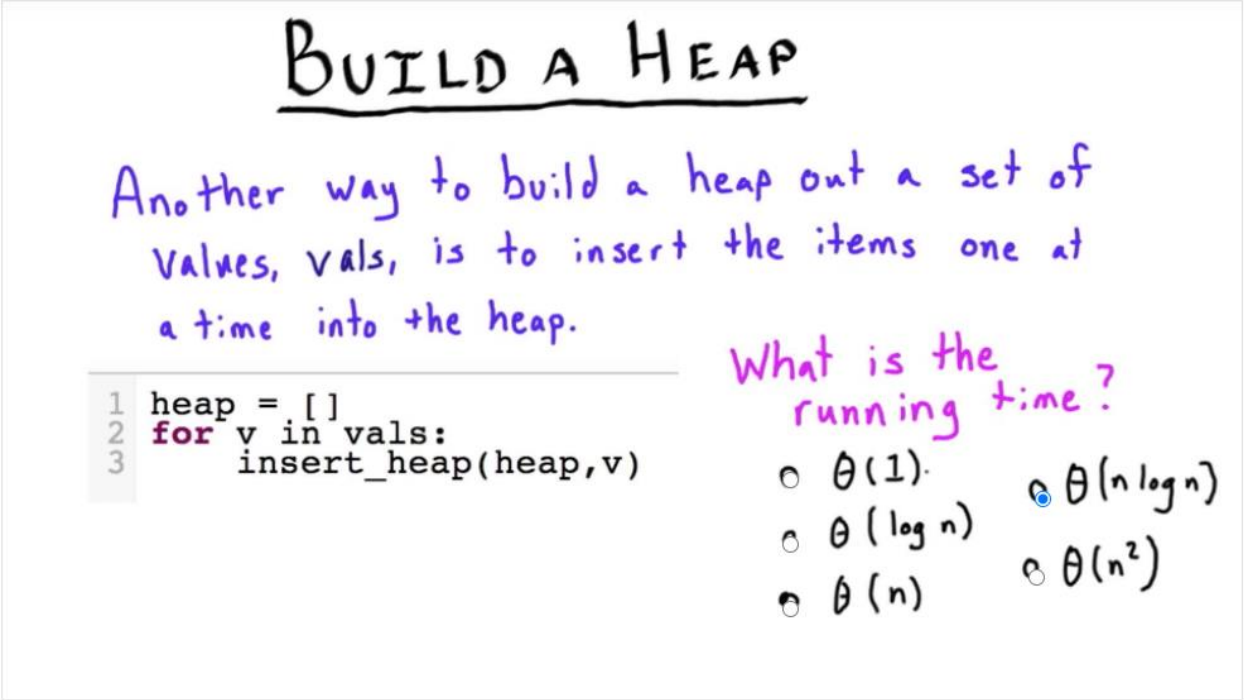
```
5  def remove_min(L):
6      L[0] = L.pop()
7      down_heapify(L,0)
8      return L
```

Heap Sort Performance



LECCIÓN 10

Build a Heap



Minimize Sum of Absolute Value

```
def minimize_absolute(L):
    L2 = L[:]
    L2.sort()
    if len(L2) % 2 == 0:
        return (L2[(len(L2) - 1)/ 2] + L2[(len(L2) - 1)/ 2]) / 2.
    else:
        return L2[(len(L2) - 1)/ 2]

```

Minimize Sum of Squares

```
#
def minimize_square(L):
    return sum(L) * 1. / len(L)
```

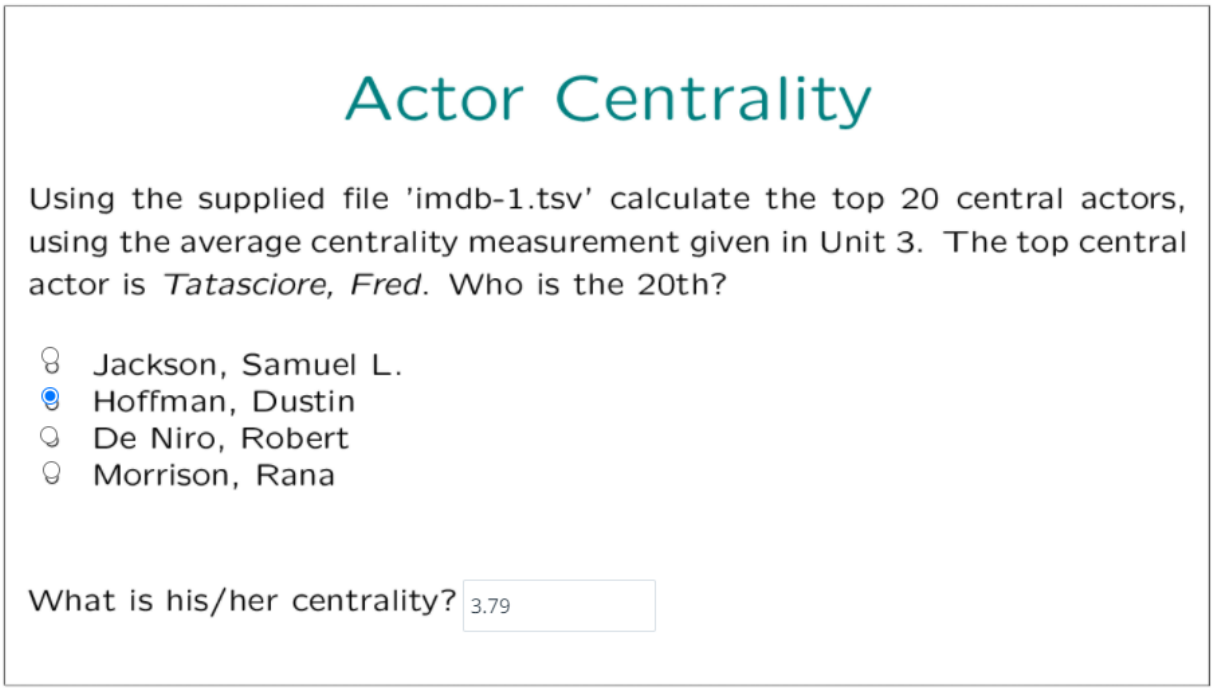
Mode

```
def mode(L):
    counts = {}
    for each in L:
        if each not in counts:
            counts[each] = 0
        counts[each] += 1
    x = 0
    y = 0
    for each in counts:
        if counts[each] > x:
            x = counts[each]
            y = each
    return y
```

Up Heapify

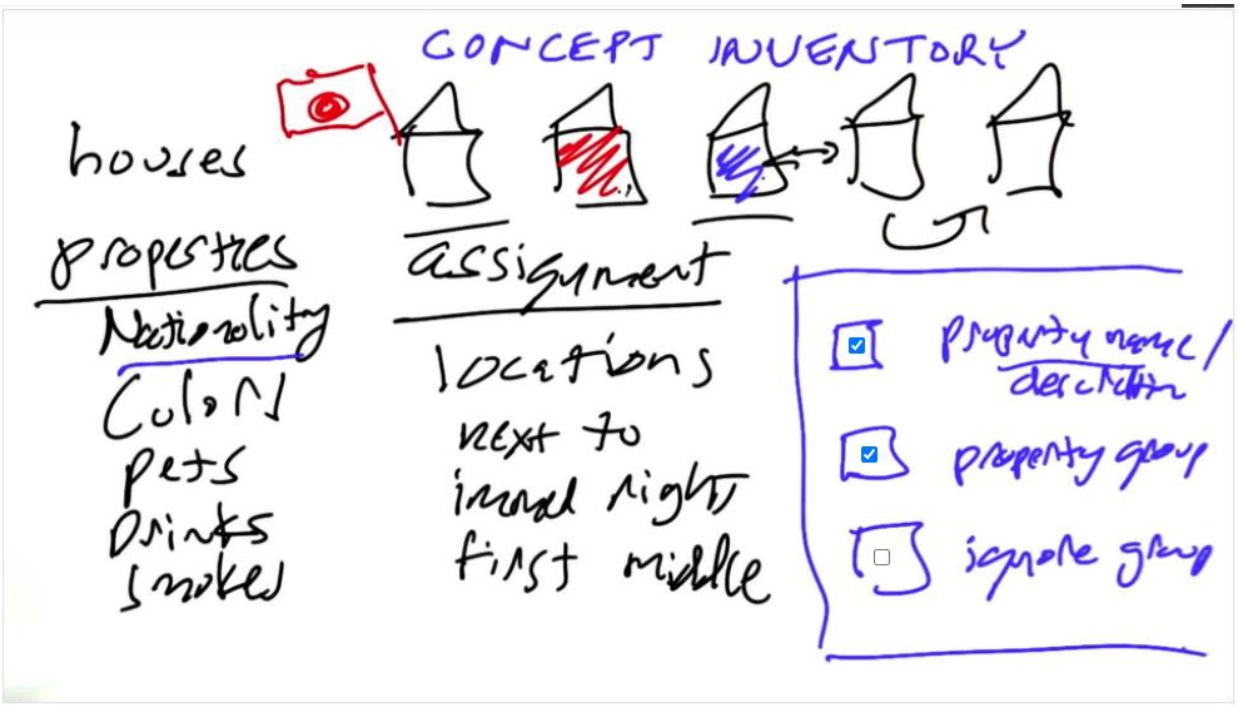
```
def up_heapify(L, i):
    while i > 0 and L[i] < L[parent(i)]:
        L[parent(i)], L[i] = L[i], L[parent(i)]
        i = parent(i)
```

Actor Centrality

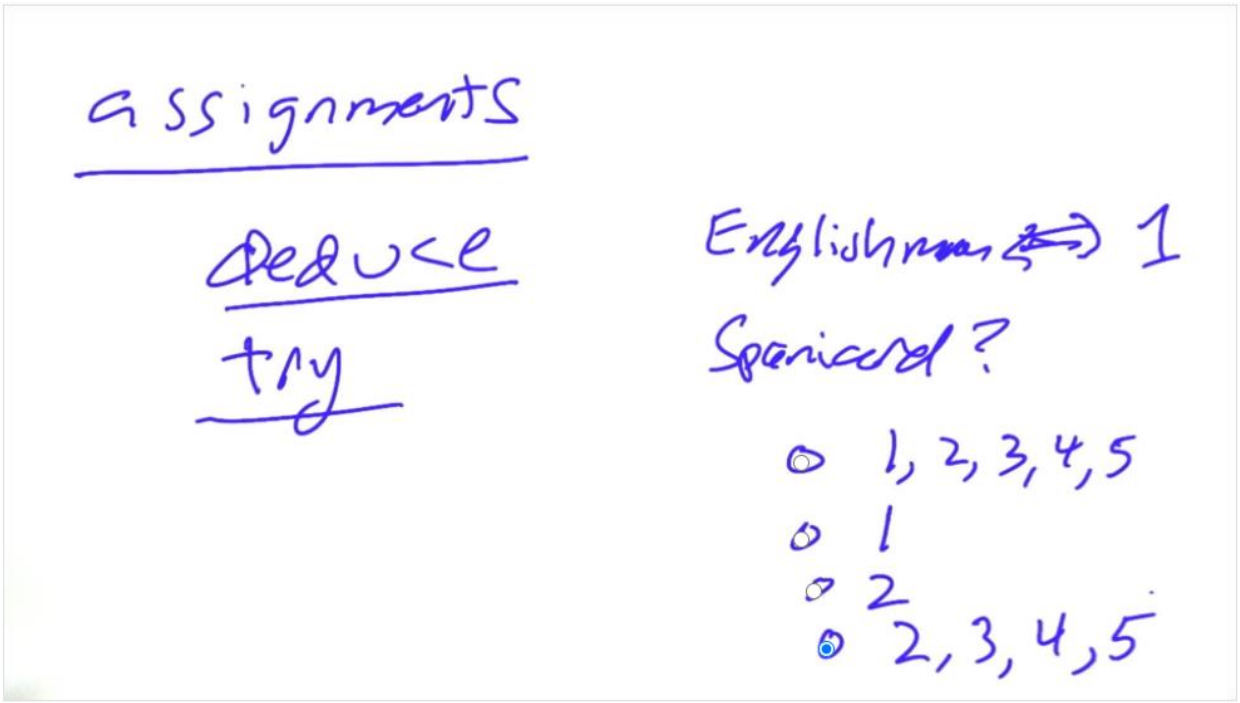


LECCIÓN 11

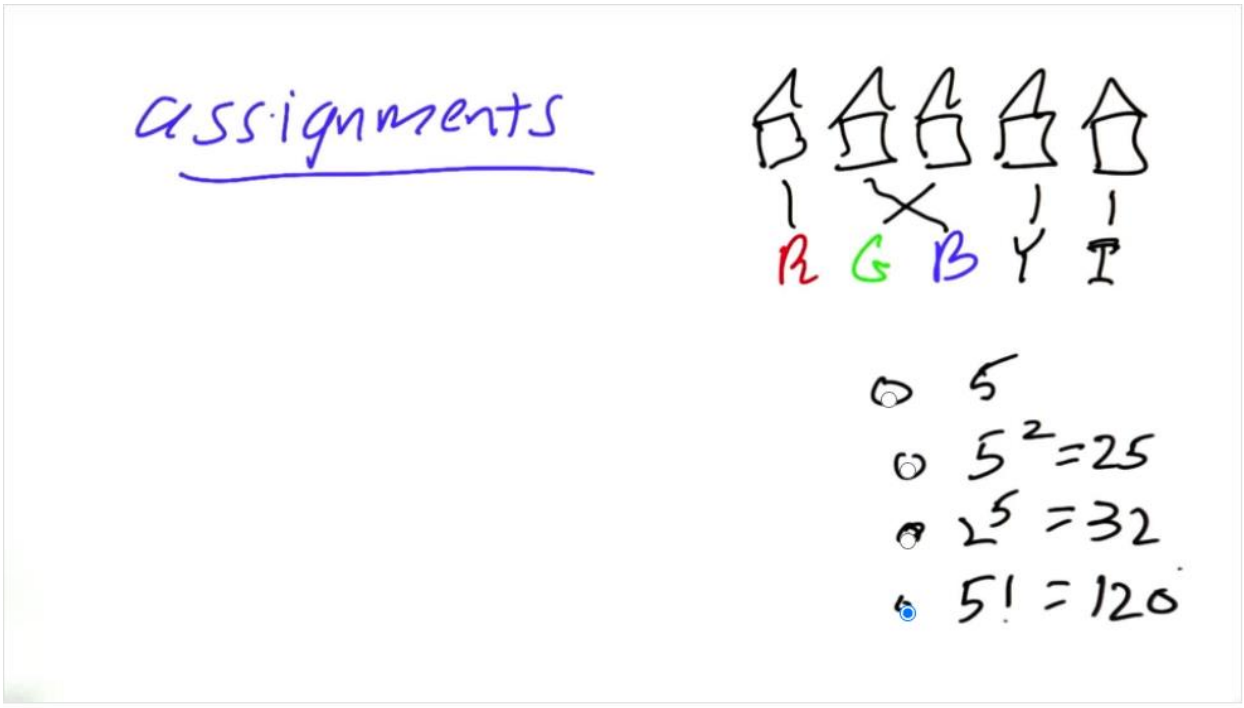
Zebra Puzzle



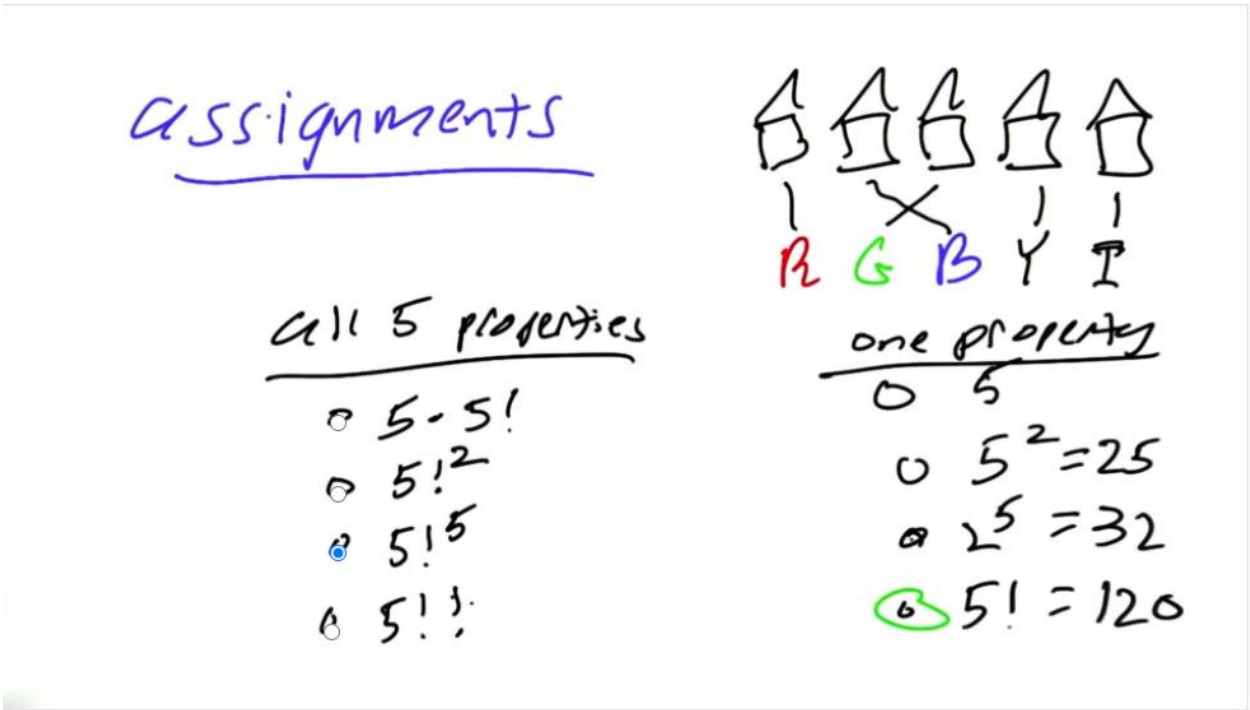
Where The Spaniard



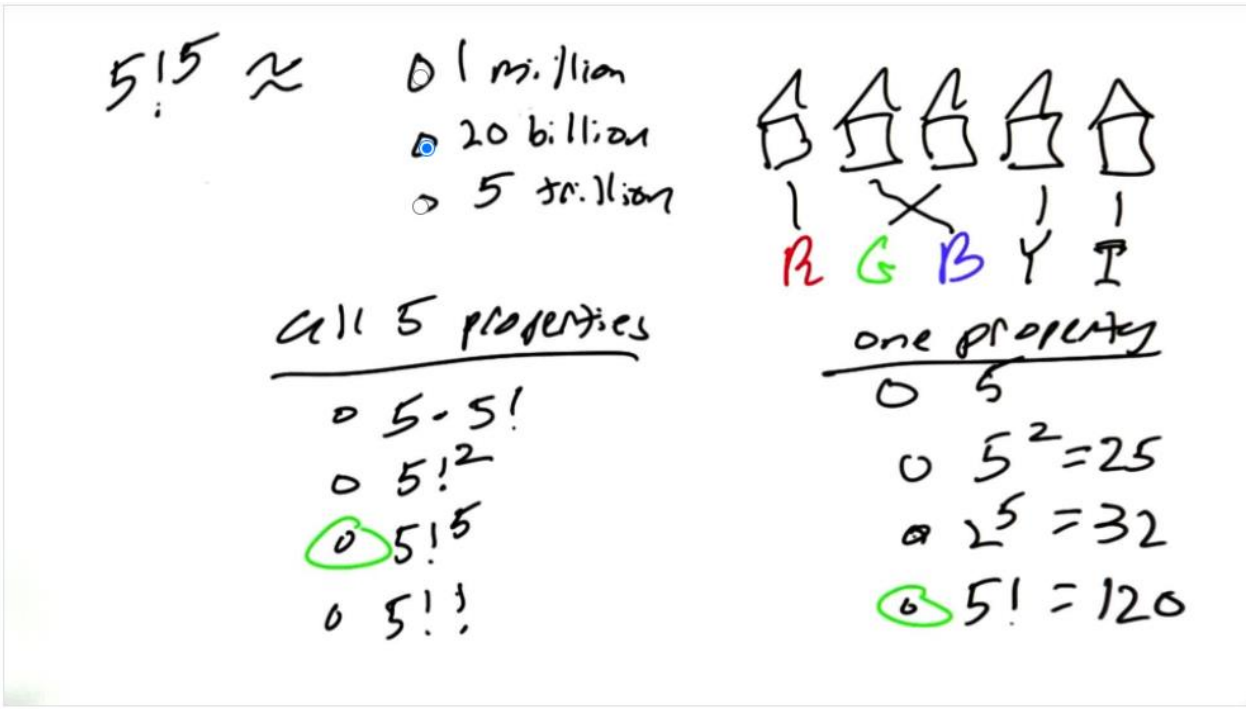
Counting Assignments



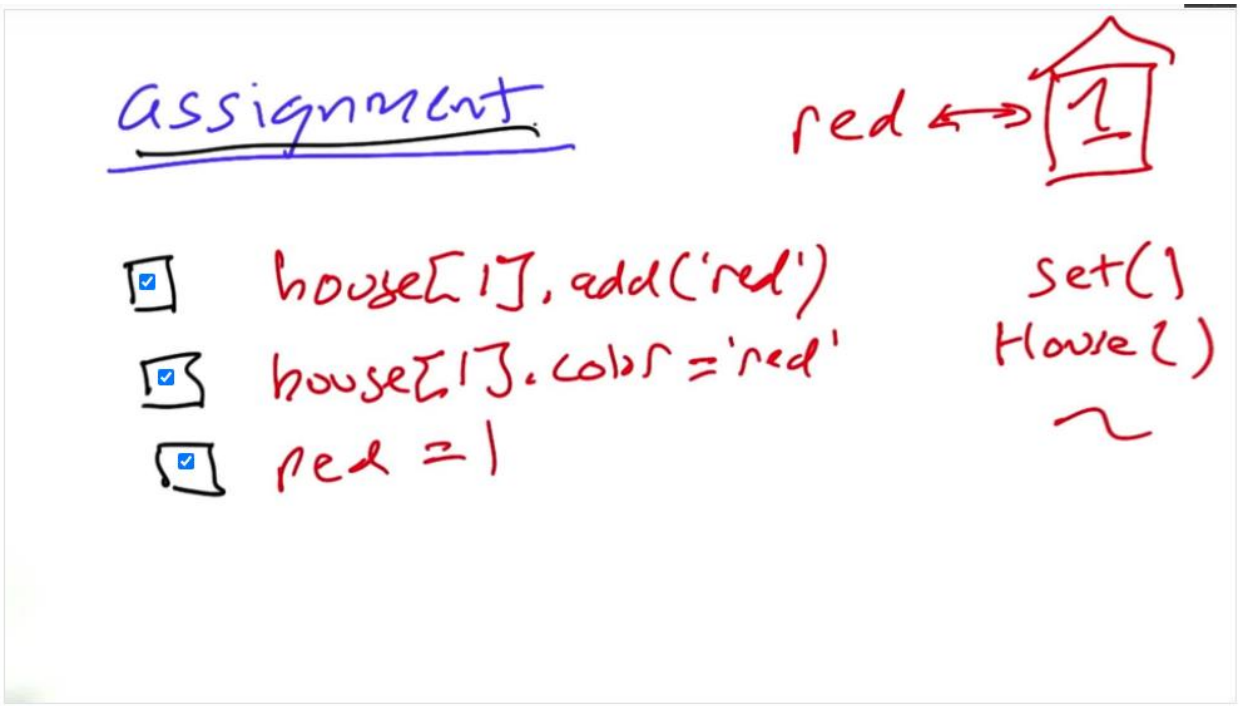
Multiple Properties



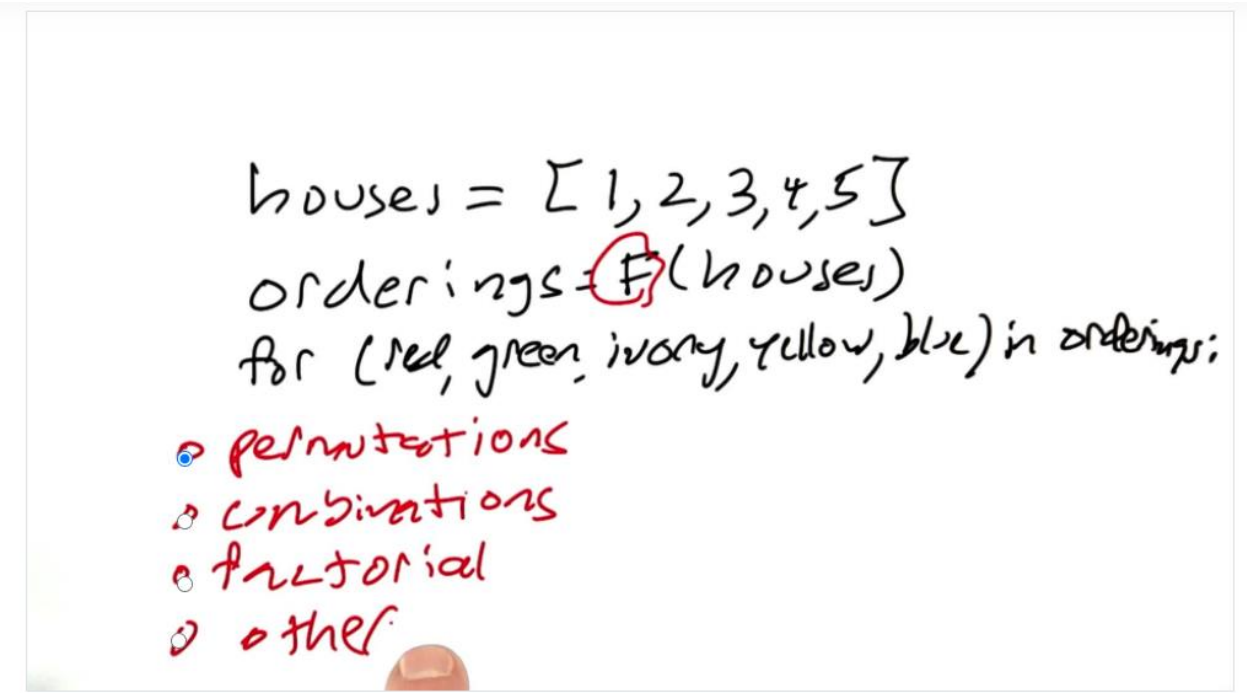
Back of the Envelope



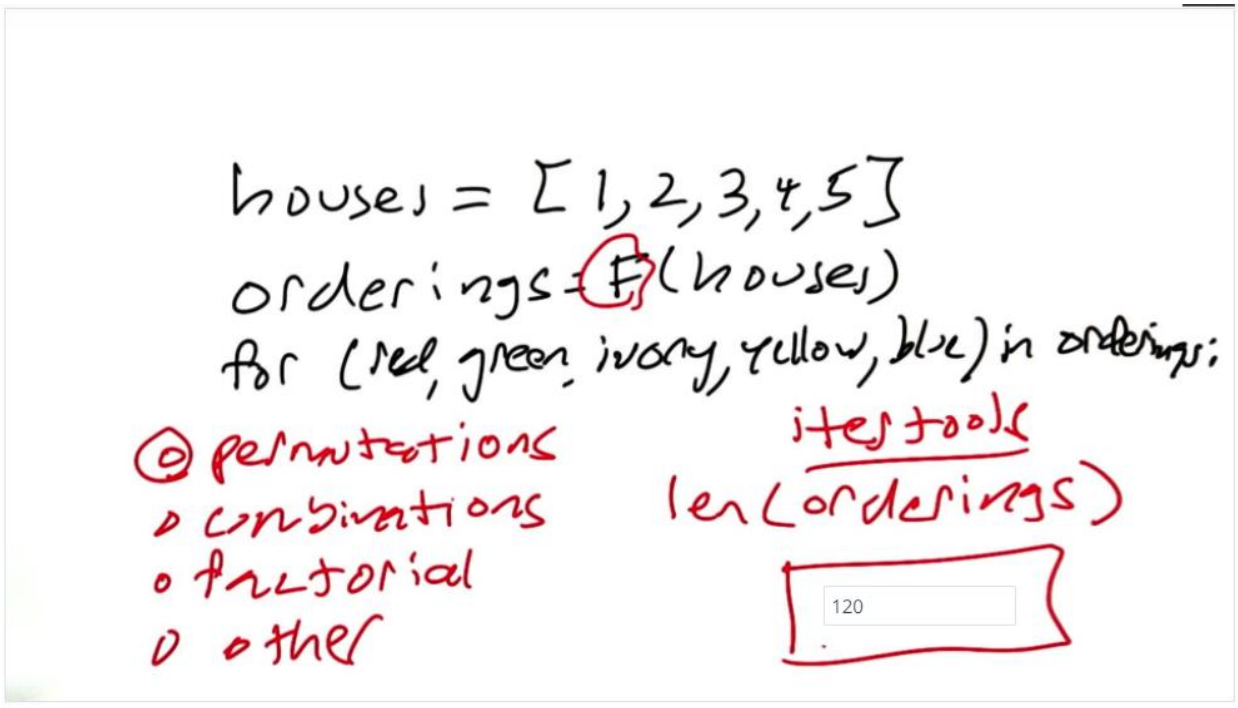
Leaving Happy Valley



Ordering Houses



Length Of Orderings



Estimating Runtime

```
1
2
3
4  import itertools
5
6  houses = [1, 2, 3, 4, 5]
7  orderings = list(itertools.permutations(houses))
8
9  for (red, green, ivory, yellow, blue) in orderings:
10   for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings:
11     for (dog, snails, fox, horse, ZEBRA) in orderings:
12       for (coffee, tea, milk, oj, WATER) in orderings:
13         for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings:
14           # constraints go here
15
16
17           # 1 sec
18           # 1 min
19           # 1 hour
20           # 1 day
```

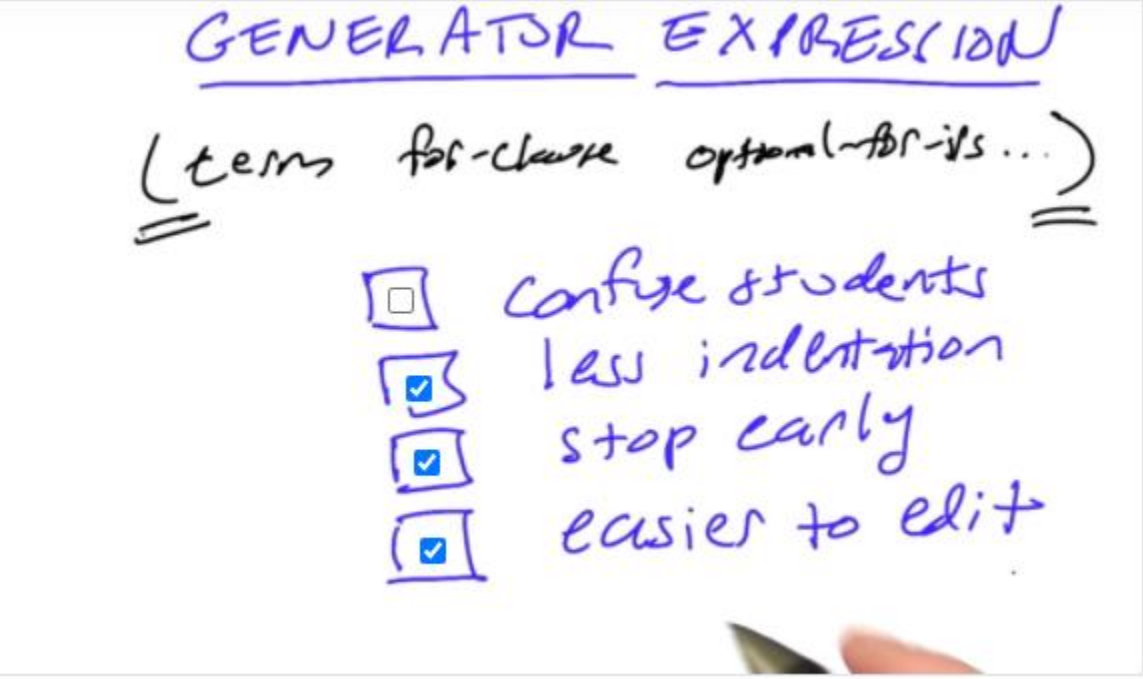
Red Englishman

```
4  import itertools
5
6  houses = [1, 2, 3, 4, 5]
7  orderings = list(itertools.permutations(houses))
8
9  for (red, green, ivory, yellow, blue) in orderings:
10   for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings:
11     for (dog, snails, fox, horse, ZEBRA) in orderings:
12       for (coffee, tea, milk, oj, WATER) in orderings:
13         for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings:
14           if ( Englishman == red ): #2
15
16
17           # 1 sec
18           # 1 min
19           # 1 hour
20           # 1 day
```

Neighbors

```
7
8  import itertools
9
10 houses = [1, 2, 3, 4, 5]
11 orderings = list(itertools.permutations(houses))
12
13 def imright(h1, h2):
14     "House h1 is immediately right of h2 if h1-h2 == 1."
15     return h1-h2 == 1
16
17 def nextto(h1, h2):
18     "Two houses are next to each other if they differ by 1."
19     return abs(h1-h2) == 1
20
```

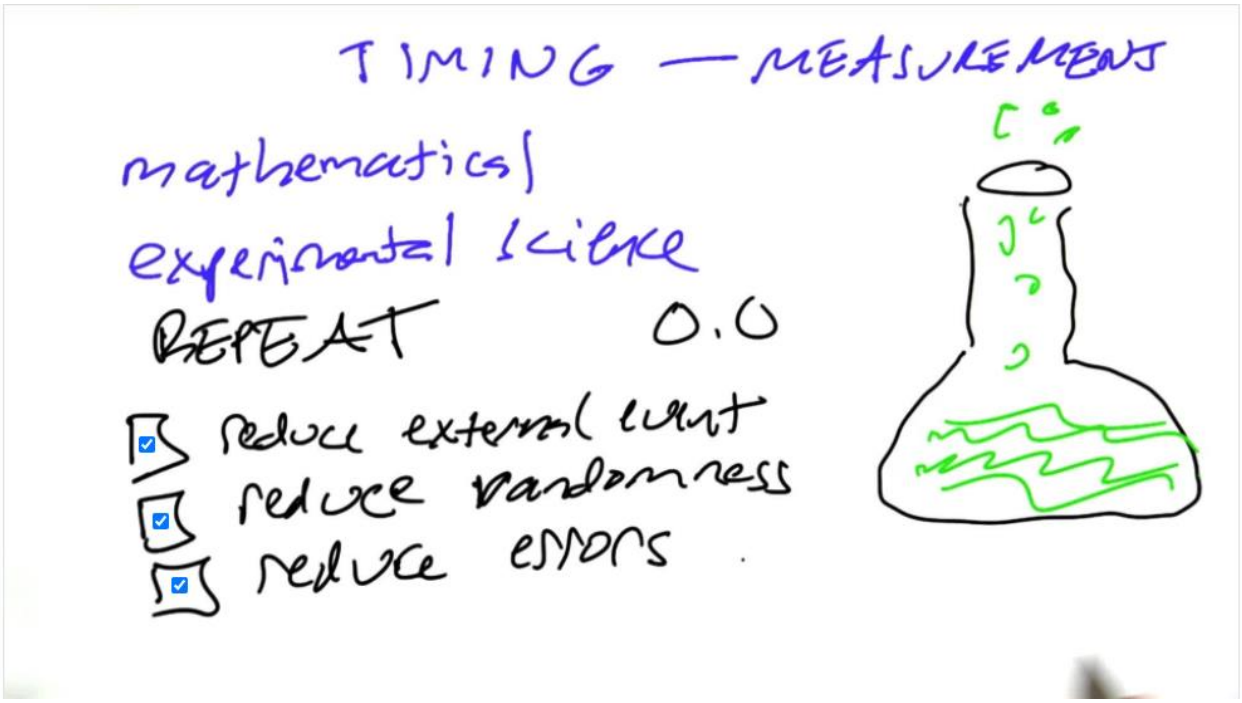
Generator Expressions



Eliminating Redundancy

```
def zebra_puzzle():
    "Return a tuple (WATER, ZEBRA) indicating their house numbers."
    houses = first, _, middle, _, _ = [1, 2, 3, 4, 5]
    orderings = list(itertools.permutations(houses))  #1
    return next((WATER, ZEBRA)
            for (red, green, ivory, yellow, blue) in orderings
            for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in orderings
            if Englishman is red             #2
            for (dog, snails, fox, horse, ZEBRA) in orderings
            for (coffee, tea, milk, oj, WATER) in orderings
            for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments) in orderings
            if Spaniard is dog               #3
```

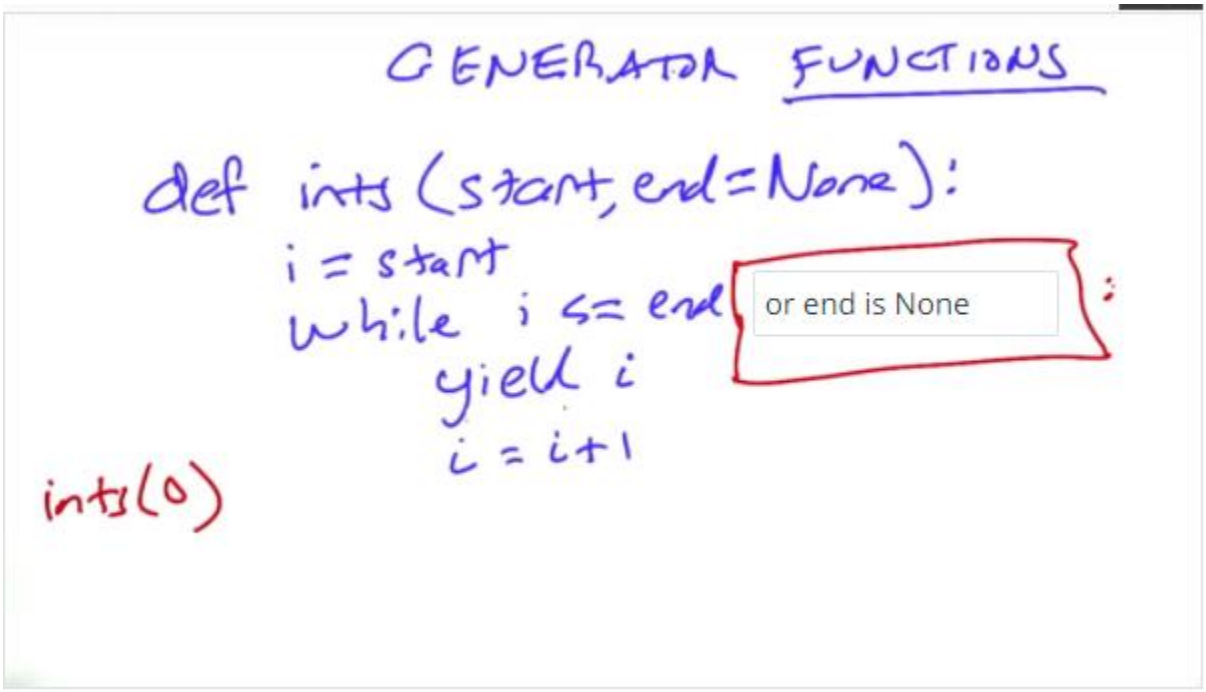
Good Science



Timed Calls

```
6
7  def timedcalls(n, fn, *args):
8      """Call fn(*args) repeatedly: n times if n is an int, or up to
9      n seconds if n is a float; return the min, avg, and max time"""
0      if isinstance(n, int):
1          times = [timedcall(fn, *args)[0] for _ in range(n)]
2      else:
3          times = []
4          while sum(times) < n:
5              times.append(timedcall(fn, *args)[0])
6      return min(times), average(times), max(times)
7
```

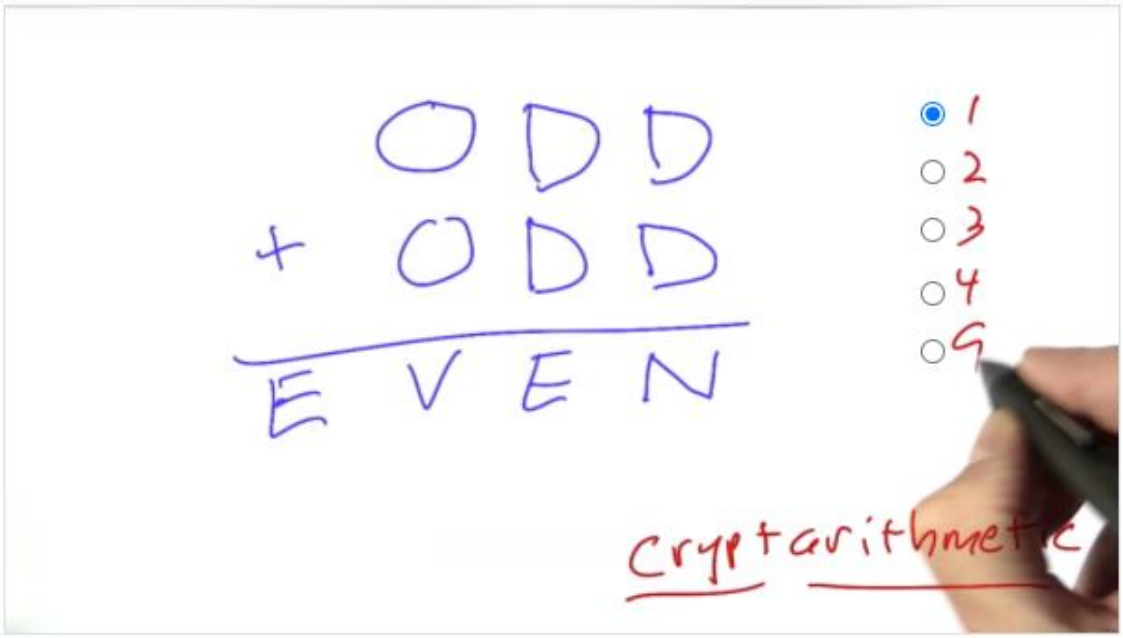
Yeilding Results



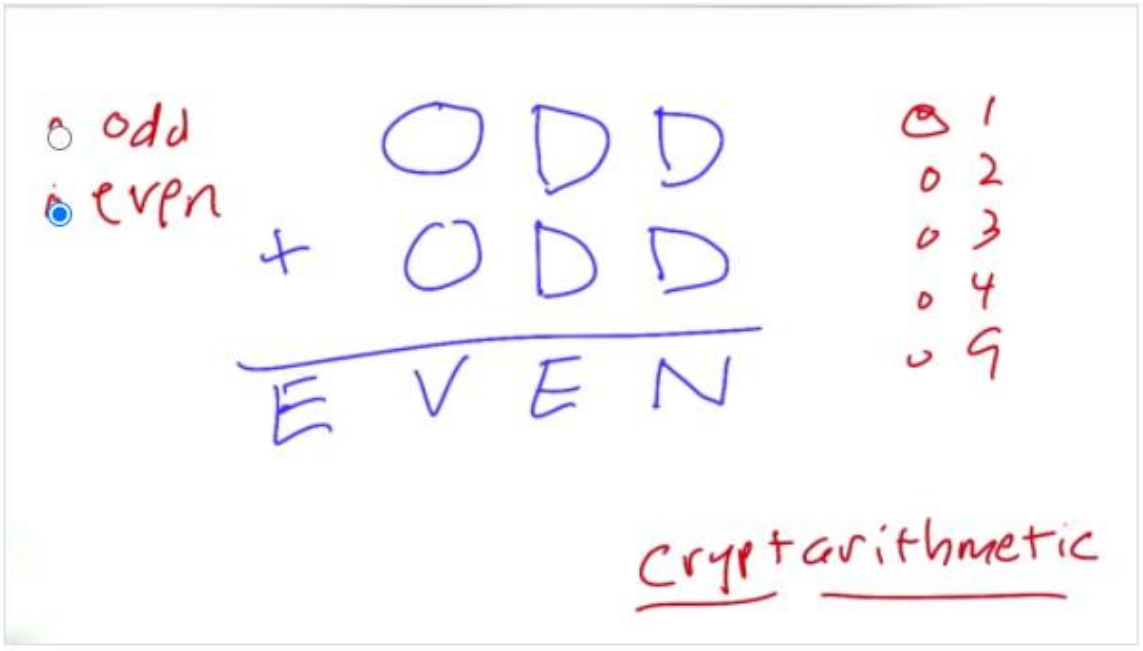
All Ints

```
14 def all_ints():
15     "Generate integers in the order 0, +1, -1, +2, -2, +3, -3, ..."
16     yield 0
17     for i in ints(1):
18         yield +i
19         yield -i
```

Cryptarithmetic



Odd or Even



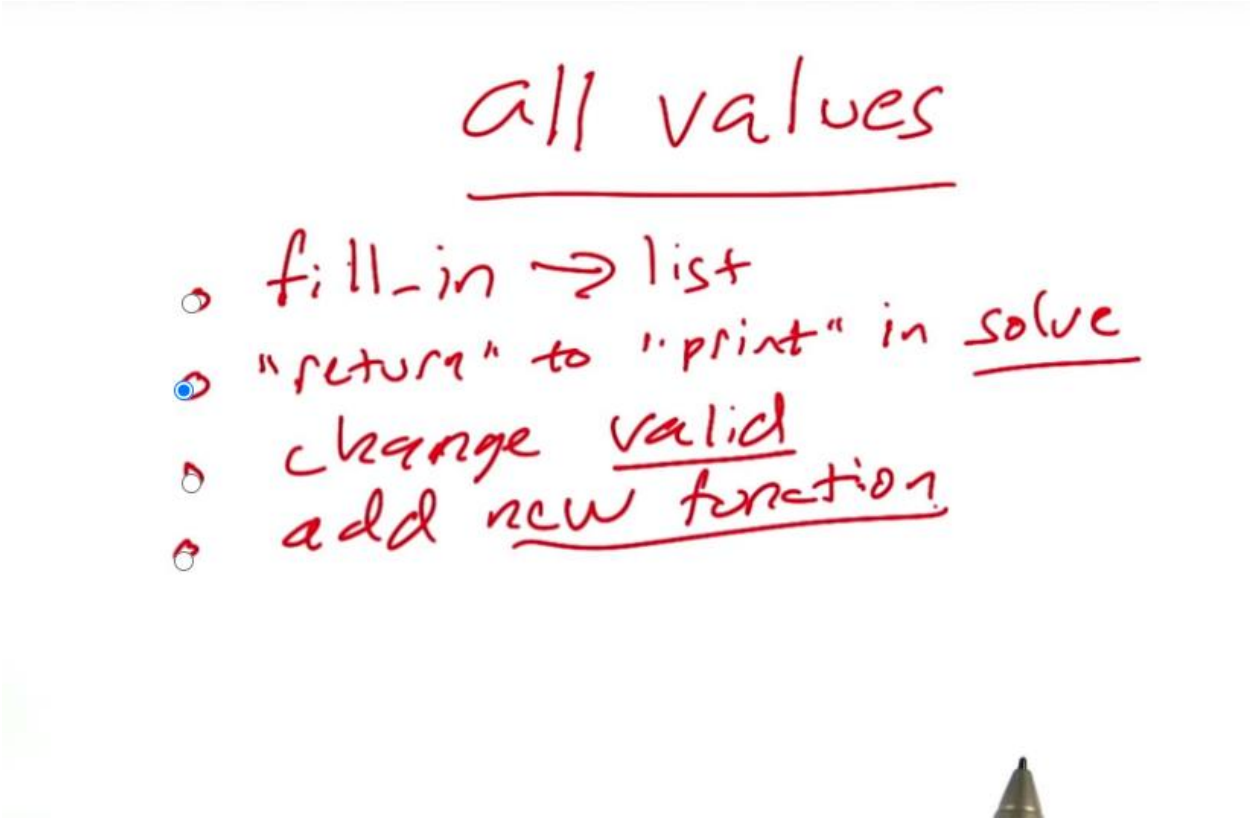
Solving Cryptarithmetic

```
def solve(formula):
    """Given a formula like 'ODD + ODD == EVEN', fill in digits to
    Input formula is a string; output is a digit-filled-in string
    for f in fill_in(formula):
        if valid(f):
            return f
```

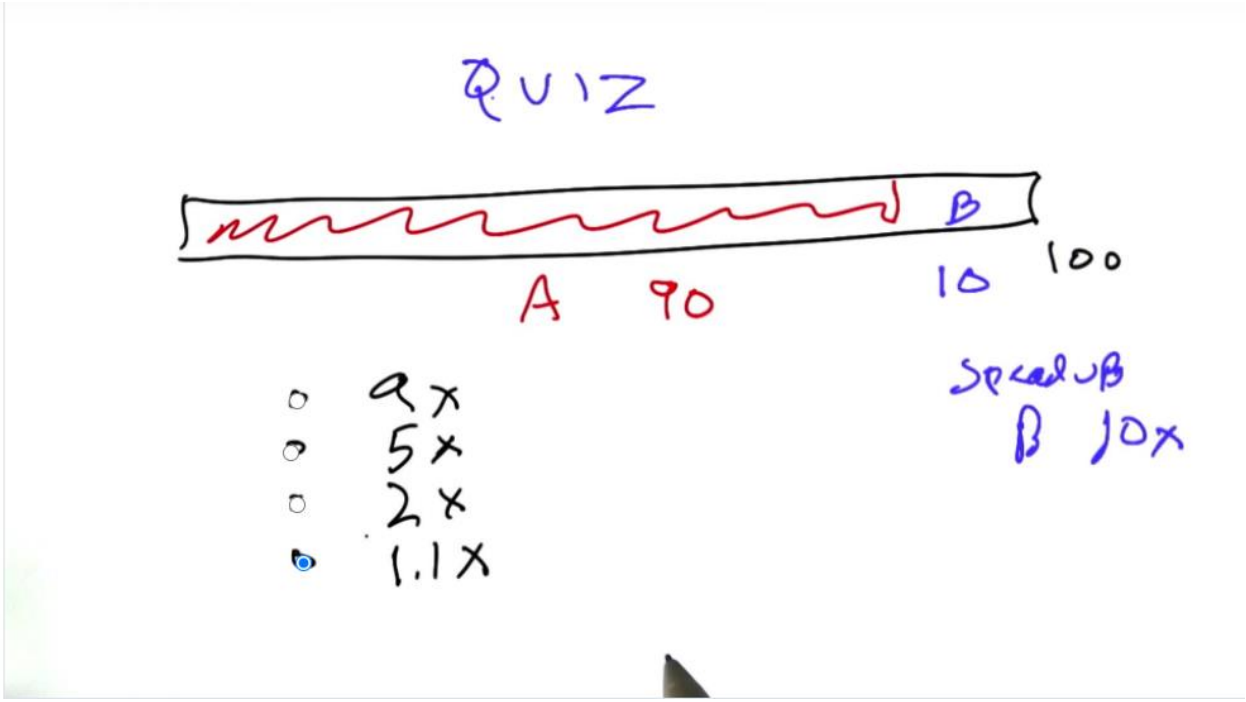
Filling In Fill In

```
def fill_in(formula):
    "Generate all possible fillings-in of letters in formula with digits."
    letters = ''.join(set(re.findall('[A-Z]', formula))) #should be a string
    for digits in itertools.permutations('1234567890', len(letters)):
        table = string.maketrans(letters, ''.join(digits))
        yield formula.translate(table)
```

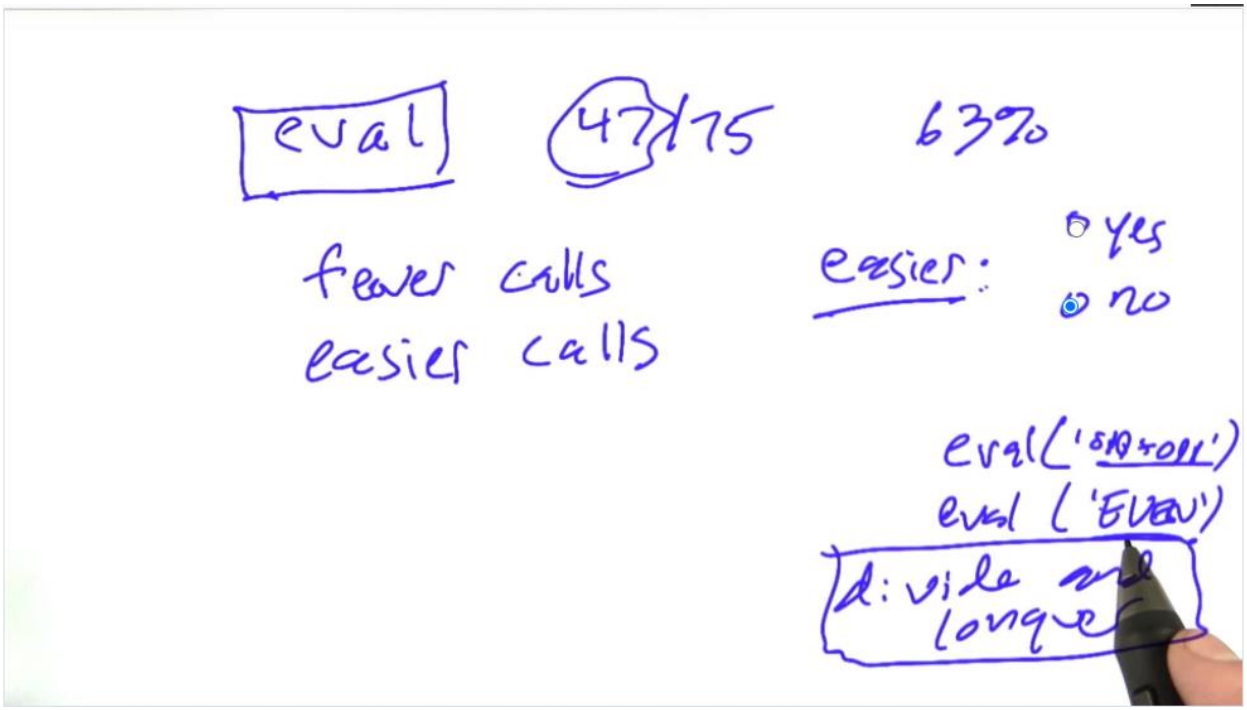
Find All Values



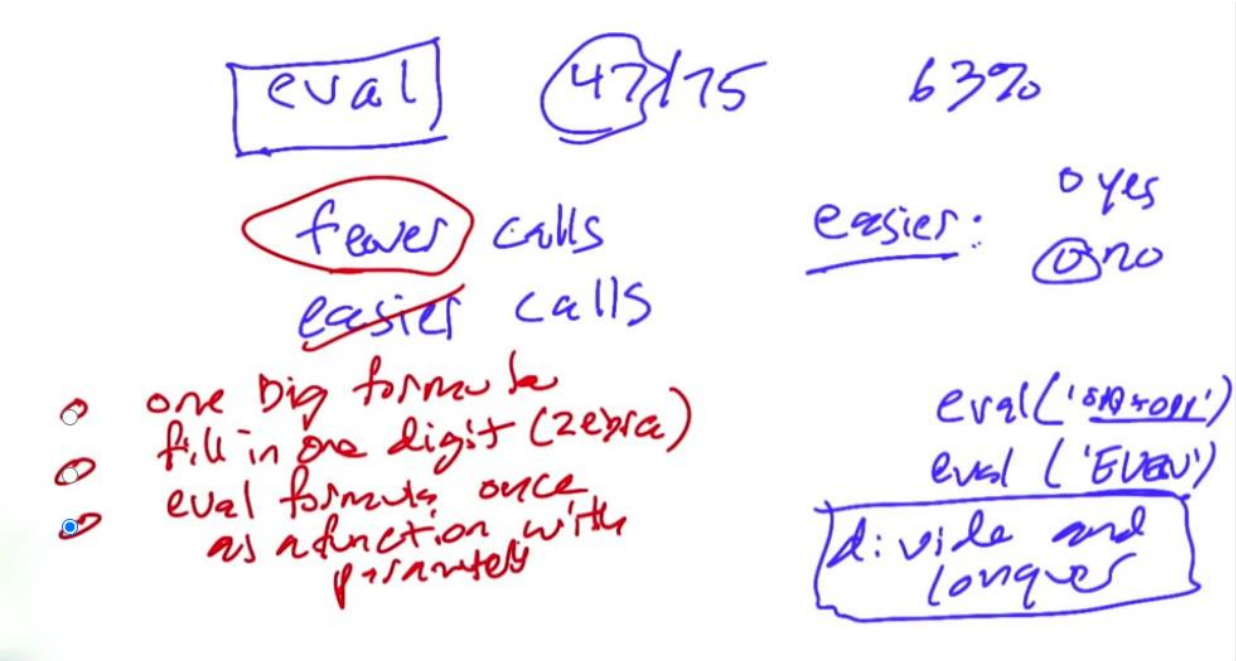
Increasing Speed



Rethinking Eval



Making Fewer Calls

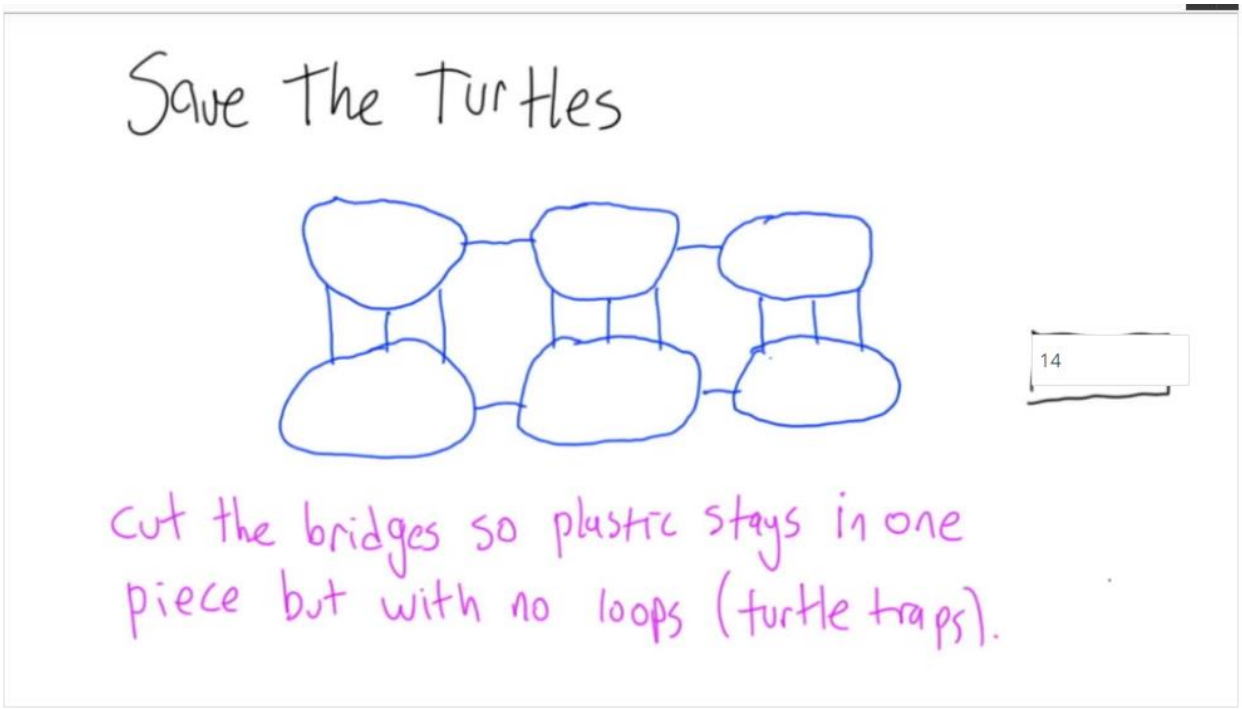


Compile World

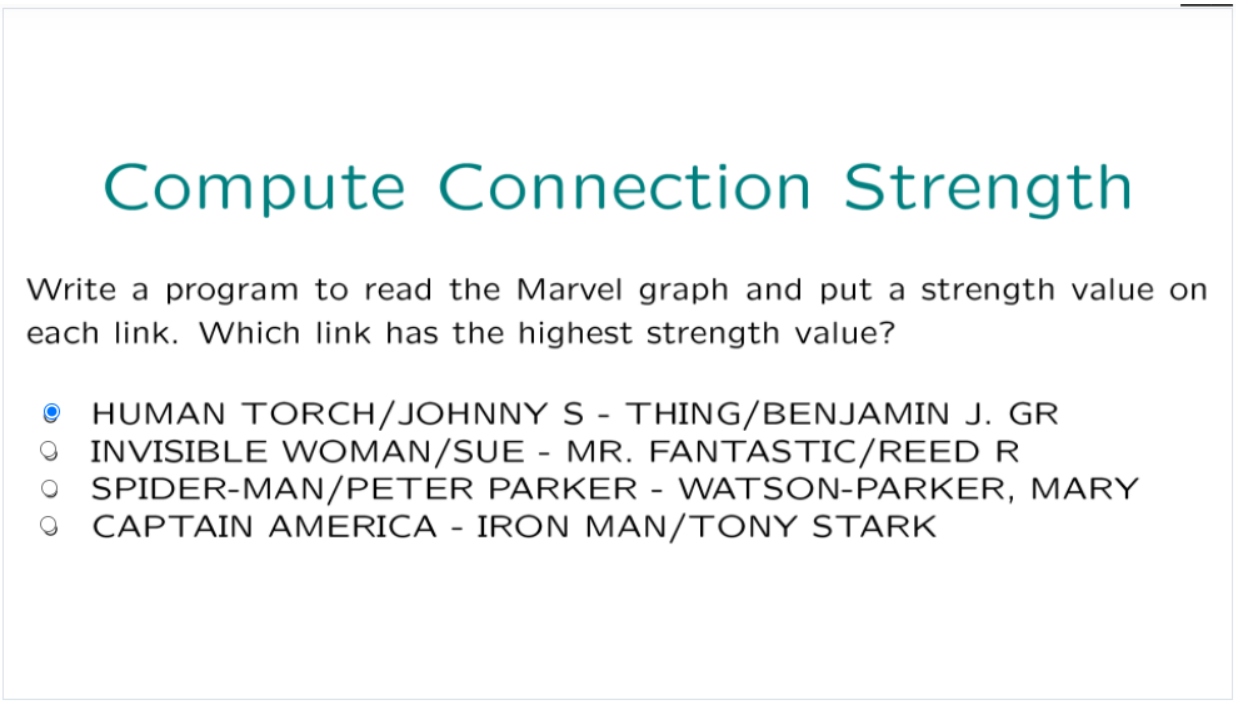
```
def compile_word(word):
    """Compile a word of uppercase letters as numeric digits.
    E.g., compile_word('YOU') => '(1*U+10*O+100*Y)'
    Non-uppercase words unchanged: compile_word('+') => '+'"""
    if word.isupper():
        terms = [('%s*%s' % (10**i, d))
                 for (i, d) in enumerate(word[::-1])]
        return '('+ '+' .join(terms) + ')'
    else:
        return word
```

LECCIÓN 12

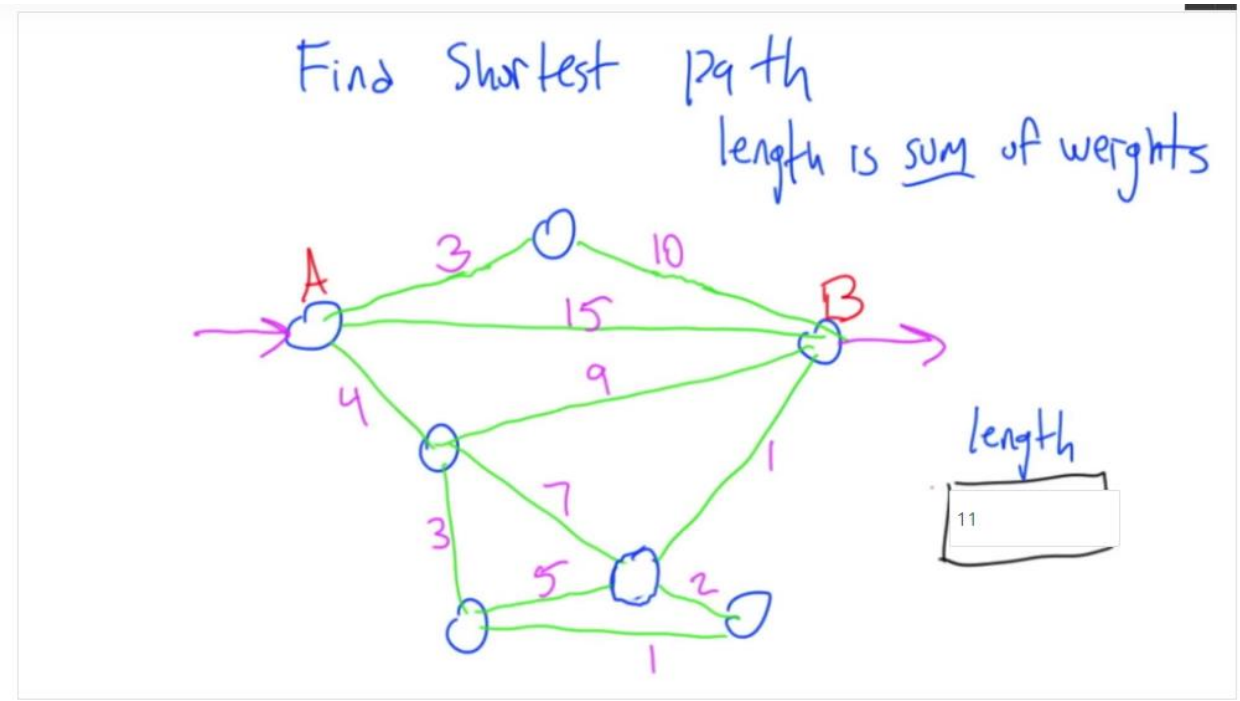
Save the Turtles



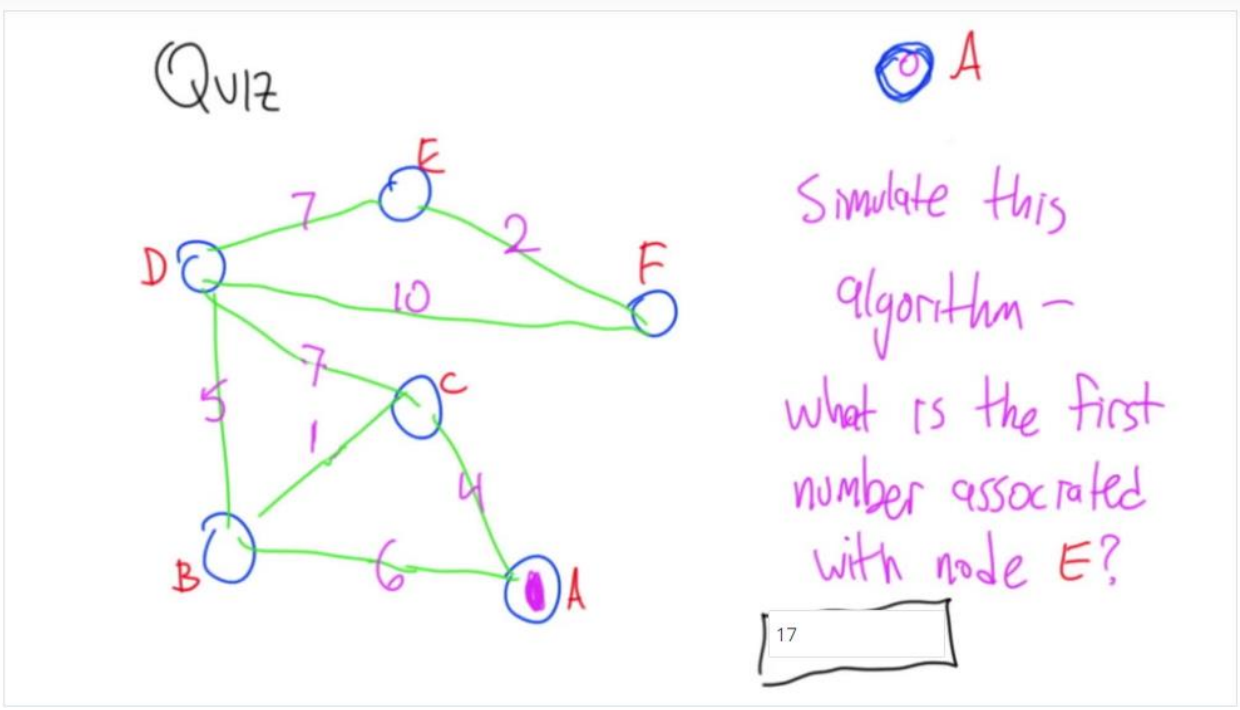
Strength of Connections



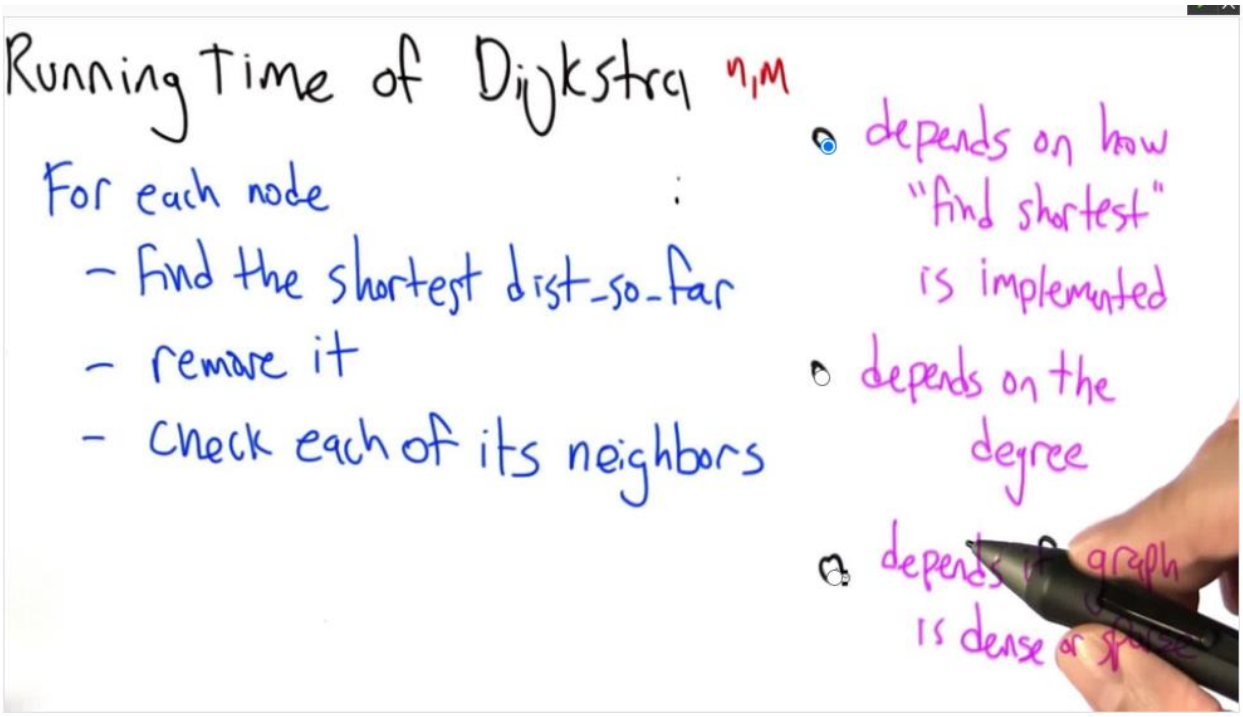
Find the Shortest Path



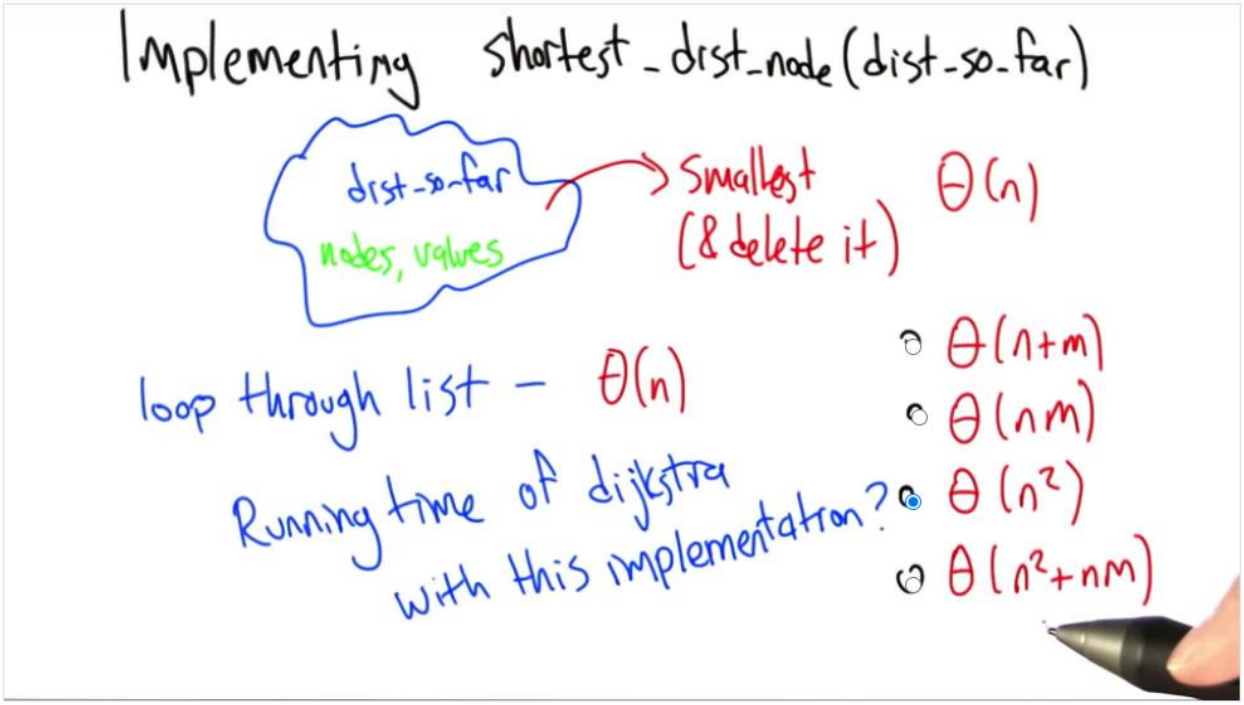
Simulate this Algorithm



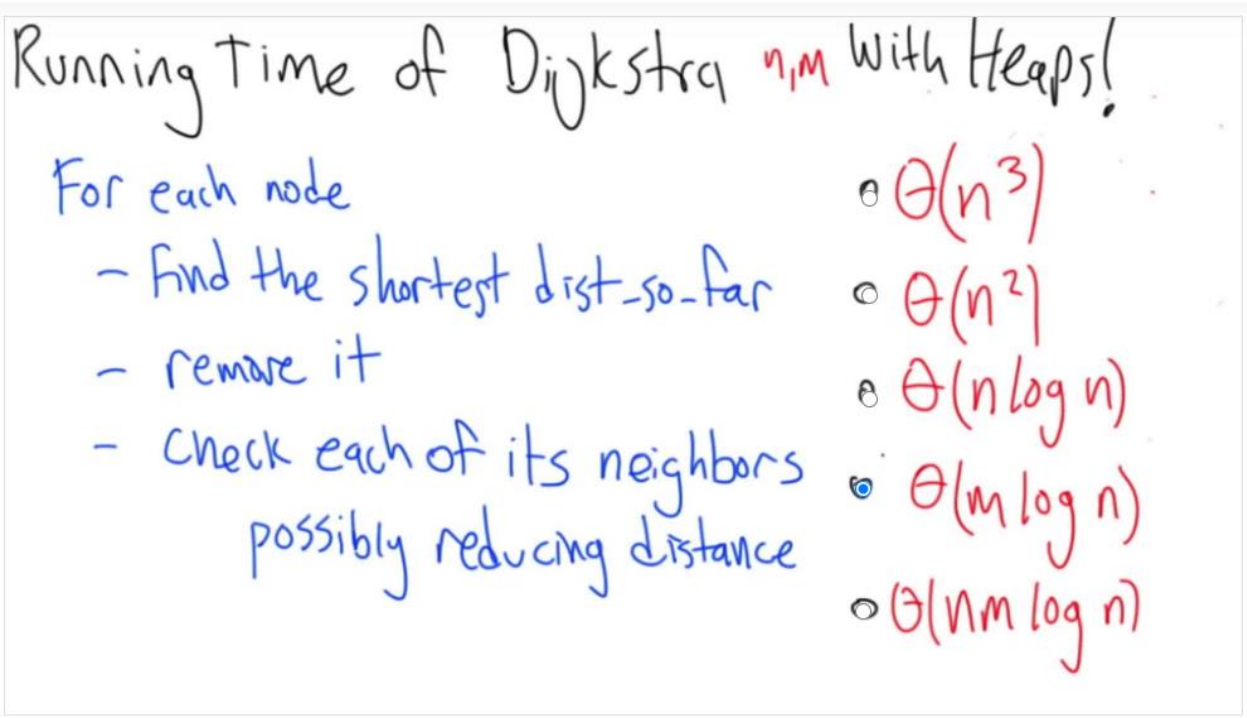
What is Missing?



Implementing Shortest Distance



Using Heaps



Randomizing Clustering Coefficient

```
def expected_C(G,v):
    # G[v].keys() is the set of neighbors of v
    neighbors = G[v].keys()
    degree = len(neighbors)
    # x in G[w][x] if x and w are connected in the graph (C[w,x])
    return clustering_coefficient(G, v)
```

LECCIÓN 13

Implementing Dijkstra with Heaps

```
9
10  import heapq
11
12  def dijkstra(HG, v):
13      dist_so_far = {v: 0}
14      final_dist = {}
15      heap = [(0, v)]
16      while dist_so_far:
17          (w, k) = heapq.heappop(heap)
18          if k in final_dist or (k in dist_so_far and w > dist_so_far[k]):
19              continue
20          else:
21              del dist_so_far[k]
22              final_dist[k] = w
23          for neighbor in [nb for nb in HG[k] if nb not in final_dist]:
24              nw = final_dist[k]+ HG[k][neighbor]
25              if neighbor not in dist_so_far or nw < dist_so_far[neighbor]:
26                  dist_so_far[neighbor] = nw
27                  heapq.heappush(heap, (nw, neighbor))
28      return final_dist
29
```

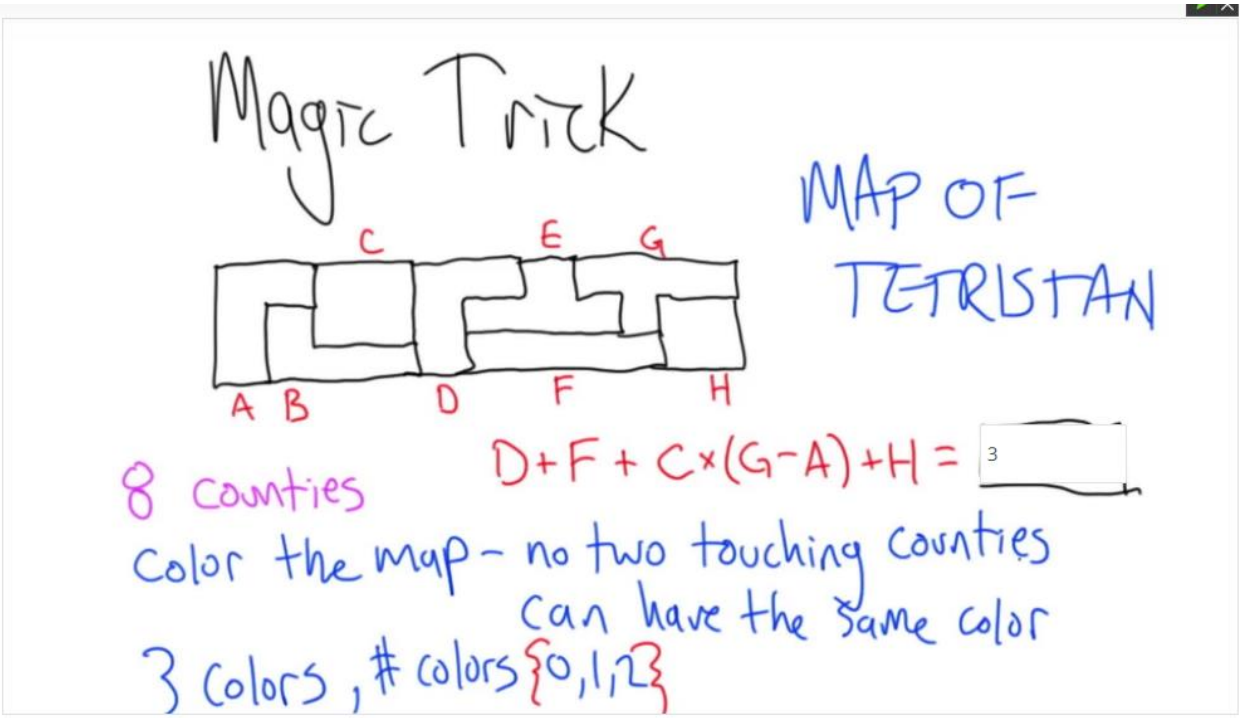
Weighted Marvel Graph

Least Obscure Path

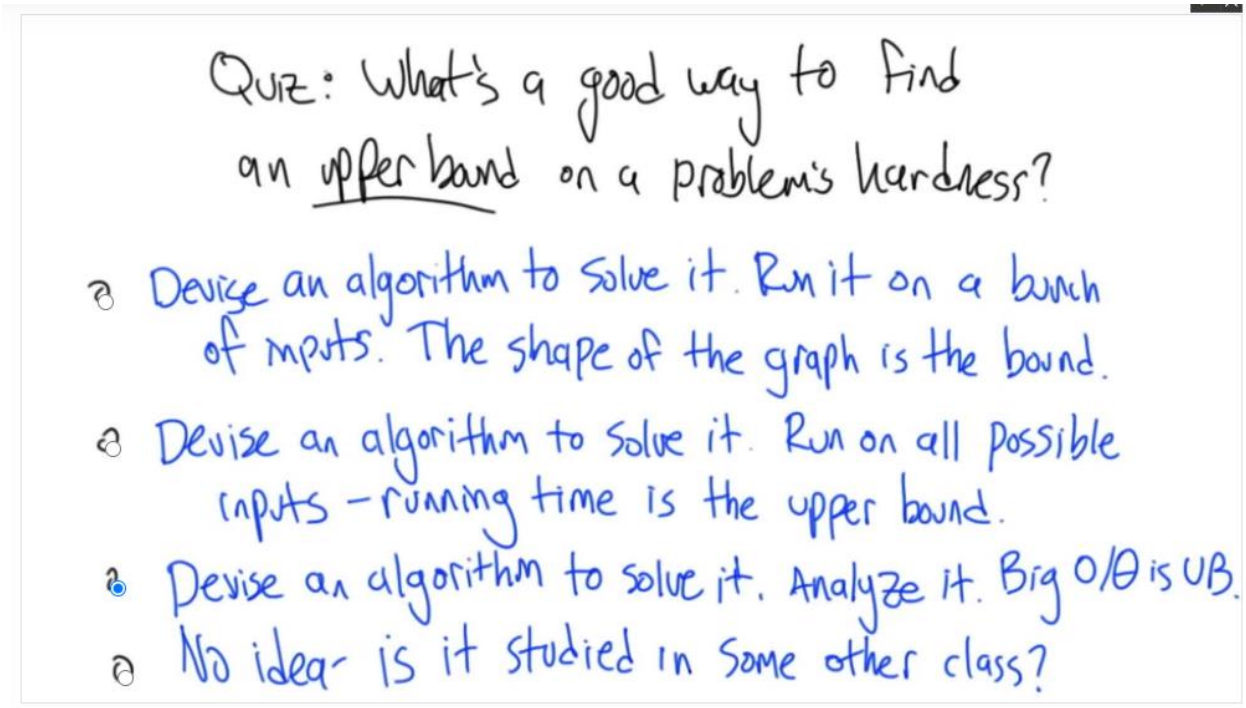
```
121     return final_dist
122
123 def solve(graph, actor_0, actor_1):
124
125     return amended_dijkstra(graph, actor_0)[actor_1]
126
127 if __name__ == '__main__':
128     total_graph = read_graph("imdb-1.tsv")
129     movie_obscurity = read_obscurity("imdb-weights.tsv")
130
131     HG = make_hop_graph(total_graph, movie_obscurity)
132
133     print "answer = {"
134     for ch1, ch2 in answer:
135         #ch1, ch2 = t[0], t[1]
136         #routes = amended_dijkstra(HG, ch1)
137         answer[ch1, ch2] = solve(HG, ch1, ch2)
138
139         print '\t(\"' + ch1 + '\", \"'+ ch2 +'\"):', answer[ch1, ch2], ","
140     print "}"
```

LECCIÓN 14

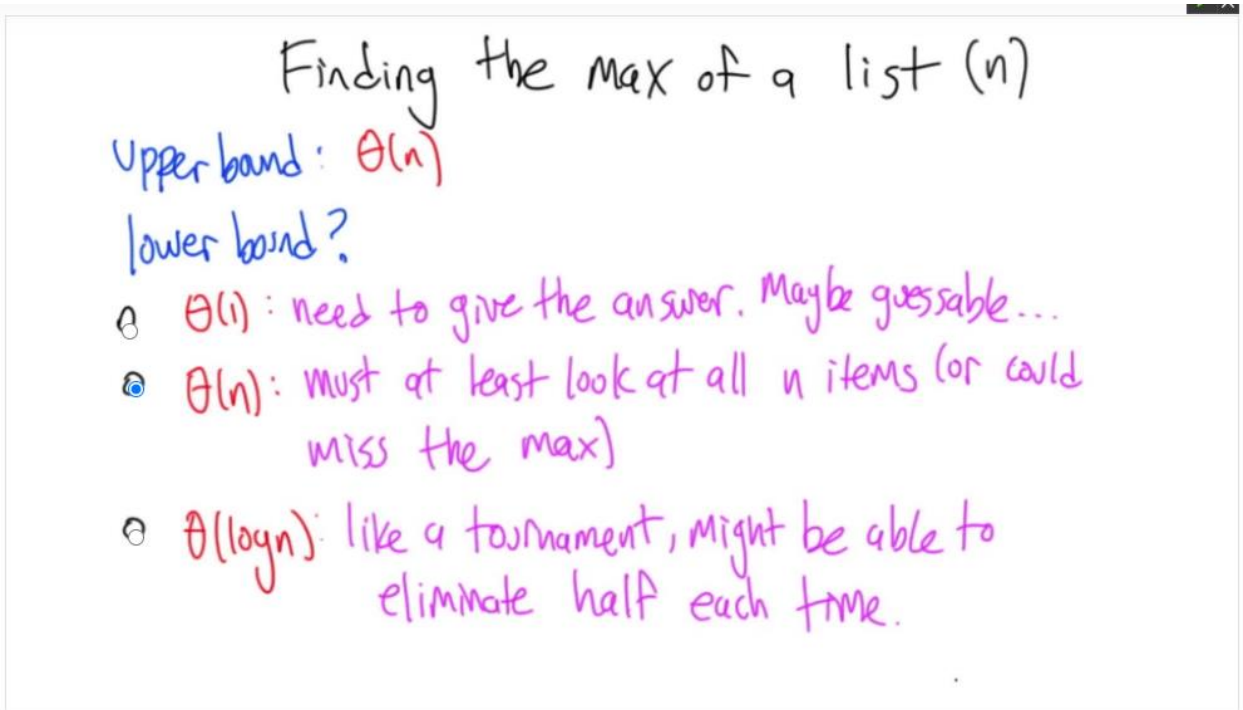
Tetristan



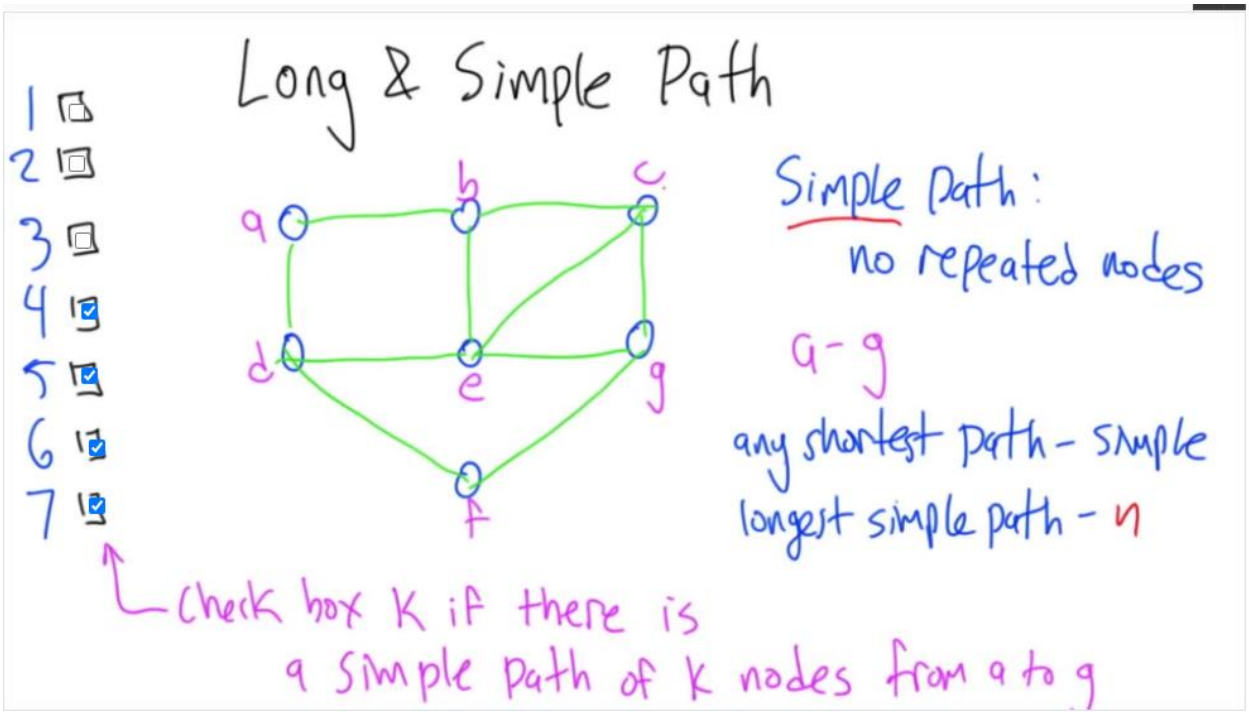
Degrees of Hardness



Lower Bound on Complexity



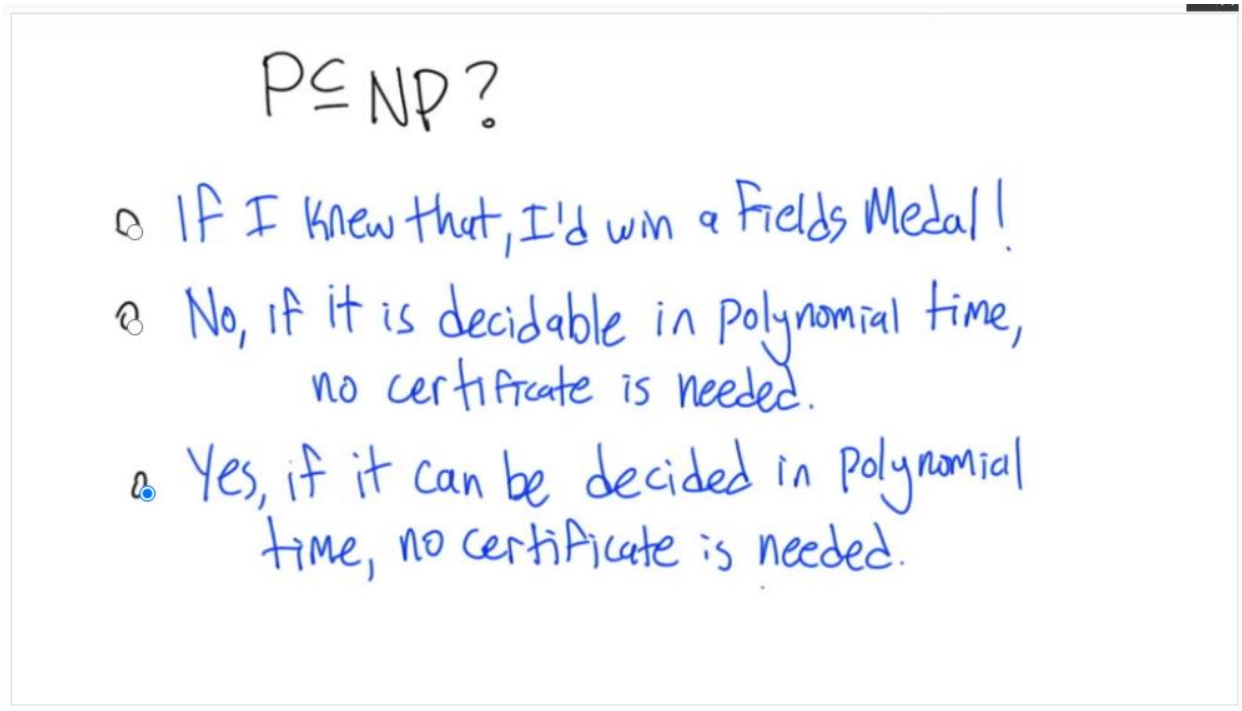
Longest Simple Path



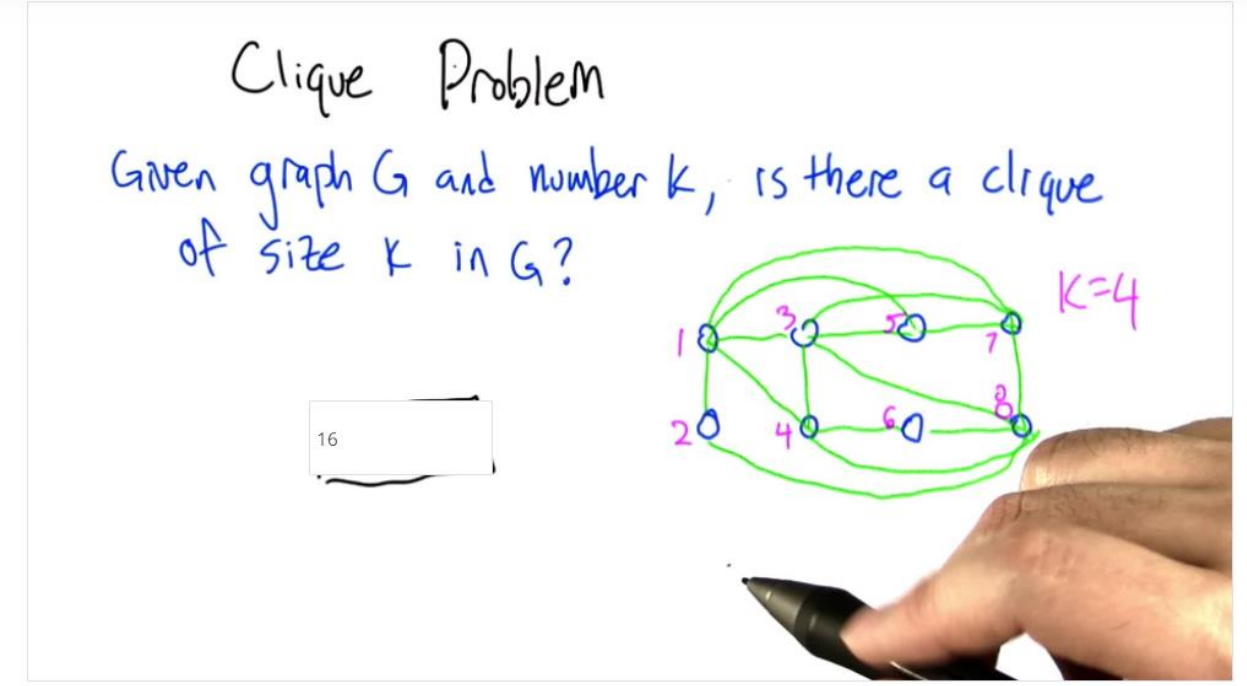
Reduction: Long and Simple Path

```
def long_and_simple_path(G,u,v,l):
    """
    G: Graph
    u: starting node
    v: ending node
    l: minimum length of path
    """
    if not long_and_simple_decision(G,u,v,l):
        return False
    for node1 in G:
        neighbors = G[node1].keys()
        for node2 in neighbors:
            G = break_link(G, node1, node2)
            if not long_and_simple_decision(G,u,v,l):
                G = make_link(G, node1, node2)
    path = [u]
    node = u
    next_node = (G[node].keys())[0]
    while next_node != v:
        path.append(next_node)
        next_next_0 = (G[next_node].keys())[0]
        next_next_1 = (G[next_node].keys())[1]
        if next_next_0 == node:
            node, next_node = next_node, next_next_1
        else:
            node, next_node = next_node, next_next_0
    return path + [v]
```

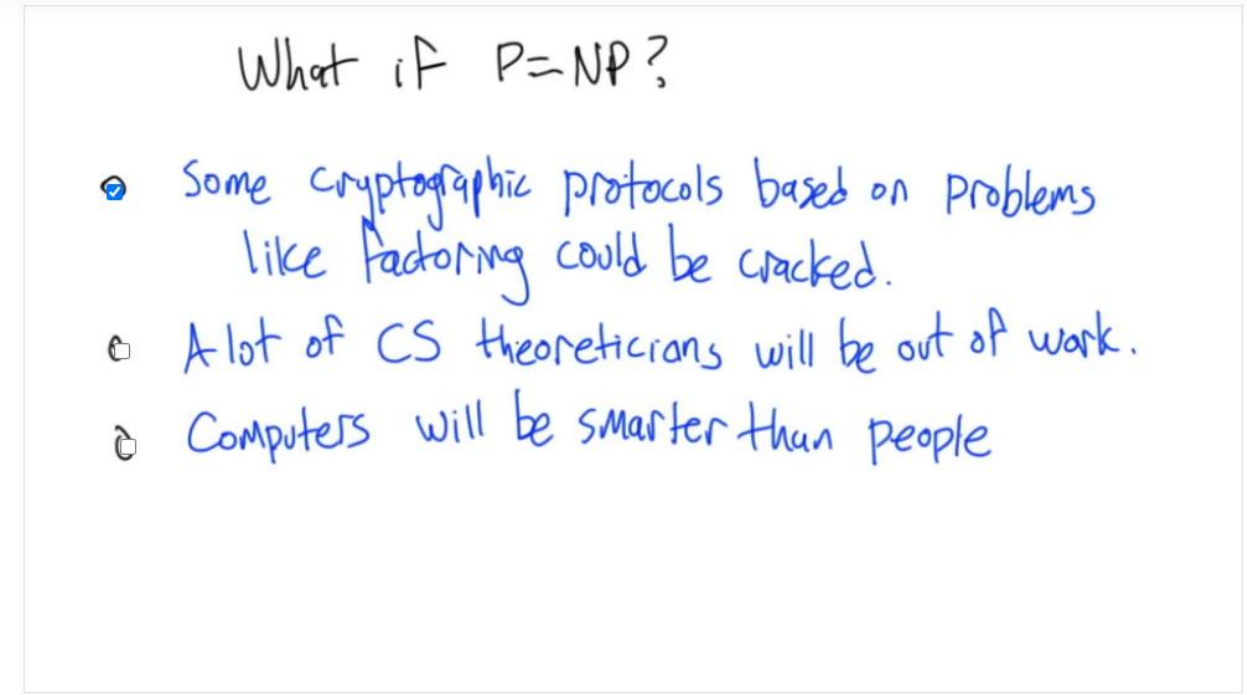
Accepting Certificate



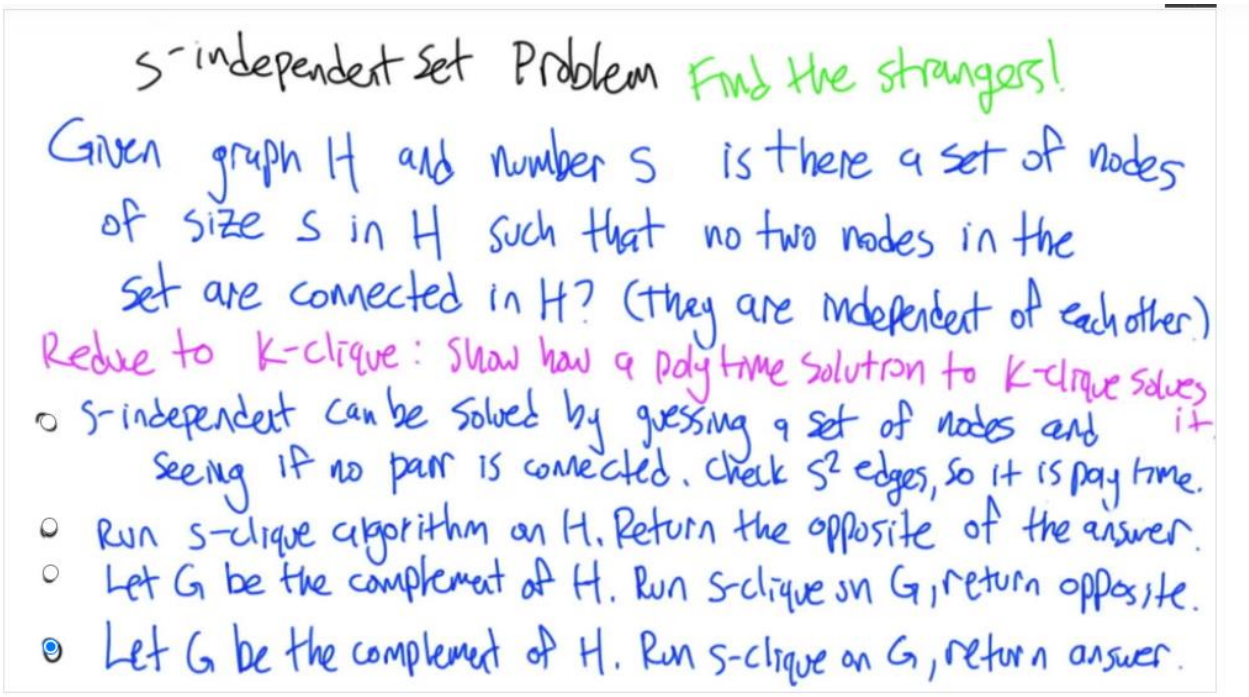
Clique Problem



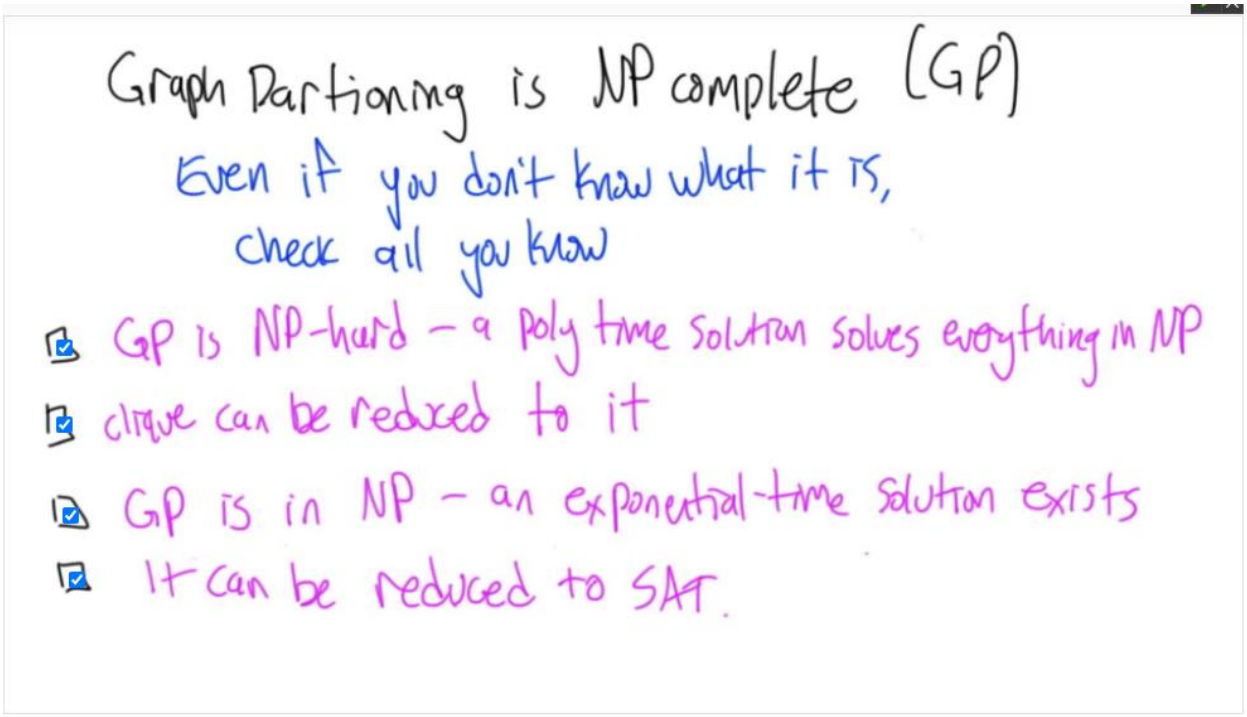
P=NP?



Find the Strangers



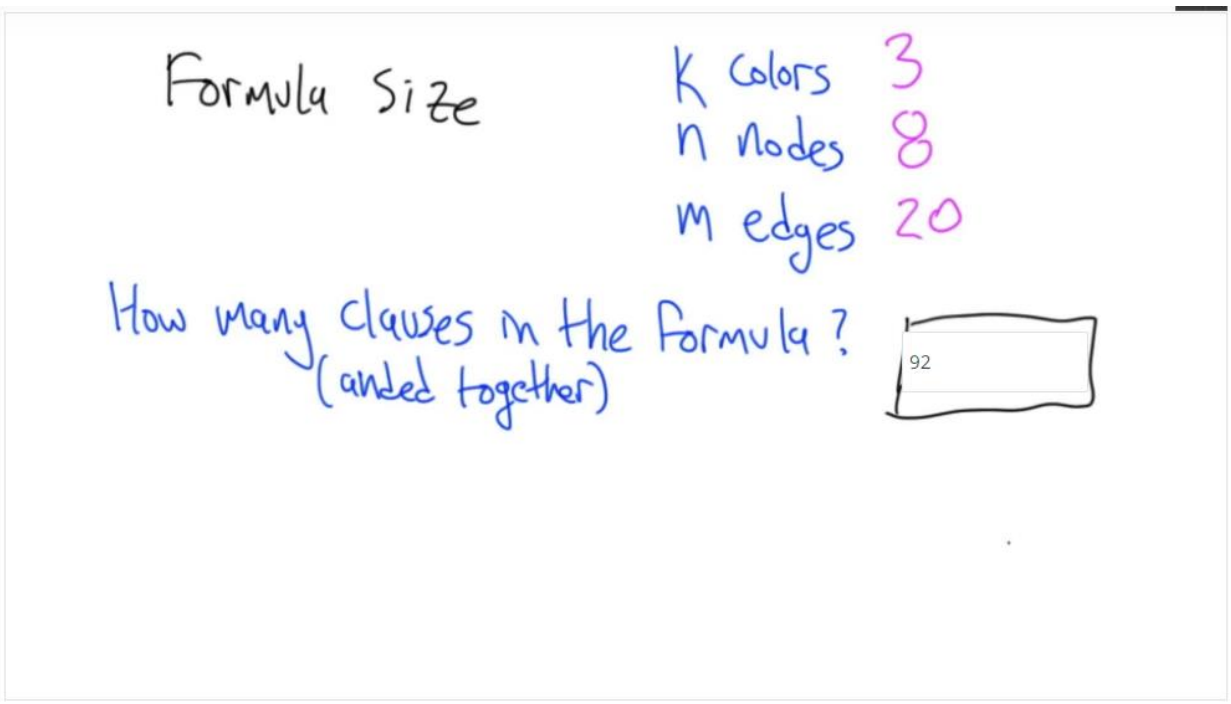
NP-Completeness



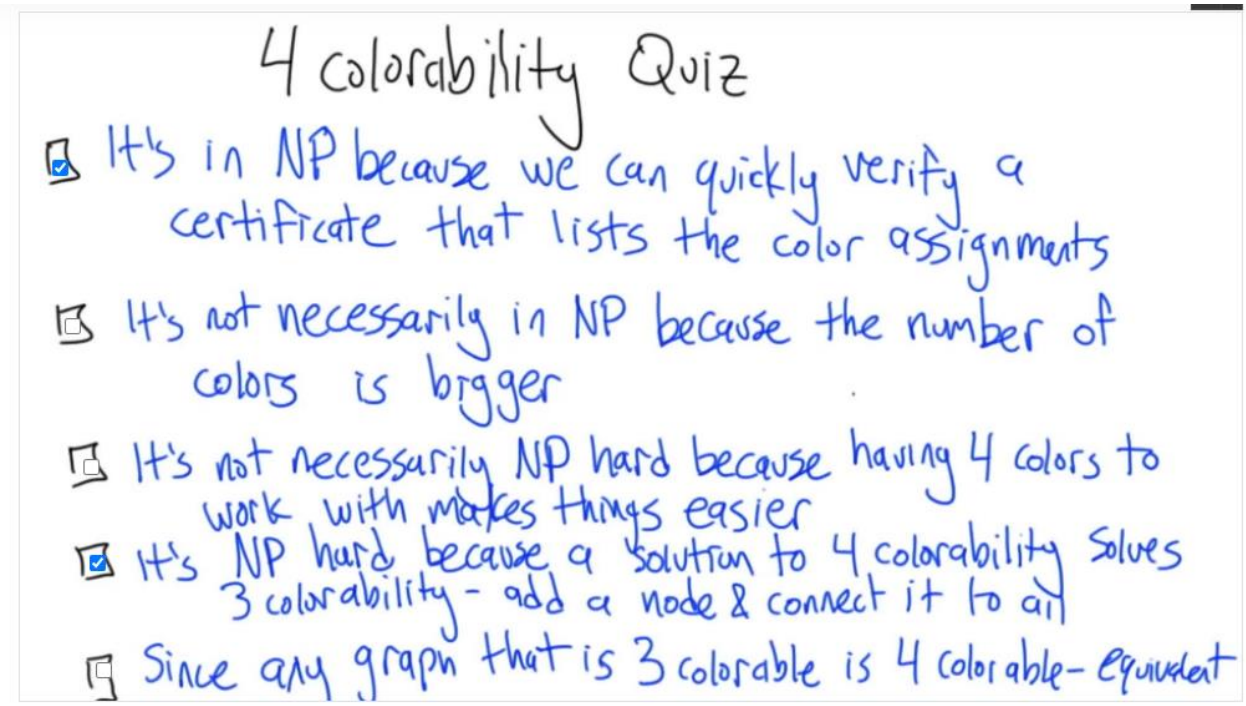
Solving 3-Colorability

```python
def verify(G, cert, k):
    if len(cert) != len(G):
        return False
    for node in cert:
        if cert[node] not in range(k):
            return False
        for neighbor in G[node]:
            if cert[node] == cert[neighbor]:
                return False
    return True
```

Generating a Formula



4-Colorabiity



LECICÓN 15

Programming a Reduction

```
69
70  # This function should use the k_clique_decision function
71  # to solve the independent set decision problem
72  def independent_set_decision(H, s):
73      return k_clique_decision(reform_graph(H), s)
74
75  if __name__ == '__main__':
76      flights = [(1,2),(1,3),(2,3),(2,6),(2,4),(2,5),(3,6),(4,5)]
77      G = {}
78      for (x,y) in flights:
79          make_link(G,x,y)
80
81      for node in G:
82          print node, "--",
83          for x in G[node]:
84              print x,
85          print
86
87      print
88      print "independent set:"
89      for i in range(1, len(flights)):
90          print i, "-" , independent_set_decision(G, i)
```

Reduction: k-Clique to Decision

```
def k_clique(G, k):
    if not k_clique_decision(G, k):
        return False
    if len(G) is k:
        return G.keys()

    for edge in get_all_edges(G):
        break_link(G, edge[0], edge[1])
        if k_clique_decision(G, k):
            return k_clique(G, k)
        make_link(G, edge[0], edge[1])

if __name__ == '__main__':
    edges = [(1,2),(1,3),(1,4),(2,4)]
    G = {}
    for (x,y) in edges:
        make_link(G,x,y)

    for x in G:
        print x, G[x].keys()
    print

    k = 3
    print "k =", k
    print k_clique(G, k)
```

Polynomial vs. Exponential

Polynomial vs. Exponential

Theoreticians would say that an algorithm with a running time of n^{100} is efficient, but one with a running time of 1.1^n is not. Assuming these running times are exact, what's the smallest n for which the efficient algorithm is faster?

9623

From Clauses to Colors

From Clauses to Colors

In the reduction from 3-SAT to 3-COLORABILITY, we talked about a way of converting a 3-SAT problem with x variables and y clauses into a graph with n nodes and m edges. Give a formula for n and m. (Fill in the boxes to complete the equation. See the example given below.)

$$n = \boxed{0}x + \boxed{0}y + \boxed{0}$$
$$m = \boxed{0}x + \boxed{0}y + \boxed{0}$$

(ex. $n = 4x + 10y + 8$)

NP or Not NP?

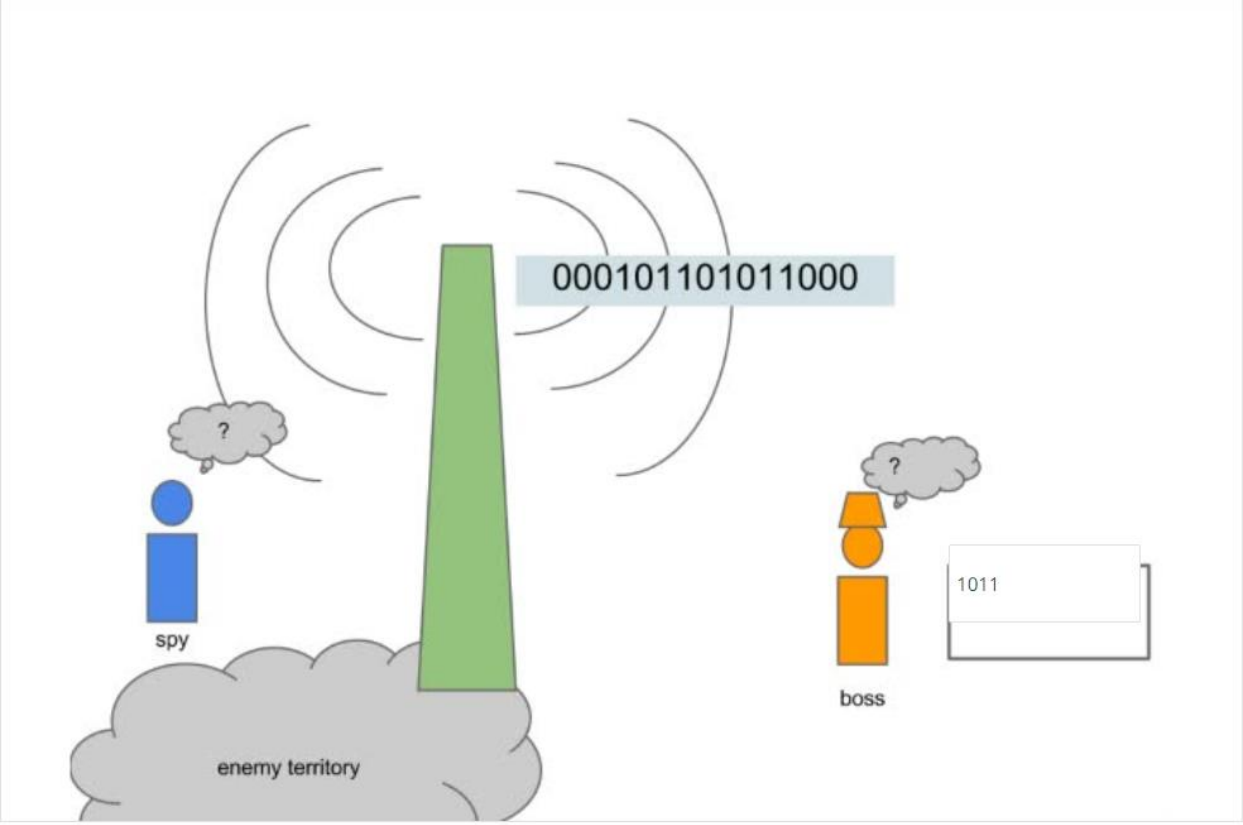
NP or Not NP? That is the Question

Select all the problems below that are in NP. Hint: Think about whether or not each one has a short accepting certificate.

- [ ] **Connectivity**: Is there a path from x to y in G?
- [ ] **Short path**: Is there a path from x to y in G that is no more than k steps long?
- [x] **Fewest colors**: Is k the absolute minimum number of colors with which G can be colored?
- [x] **Near Clique**: Is there a group of k nodes in G that has at least s pairs that are connected?
- [x] **Partitioning**: Can we group the nodes of G into two groups of size $n/2$ so that there are no more than k edges between the two groups.
- [x] **Exact coloring count**: Are there exactly s ways to color graph G with k colors?

LECCIÓN 16

Spy Control



Homophily

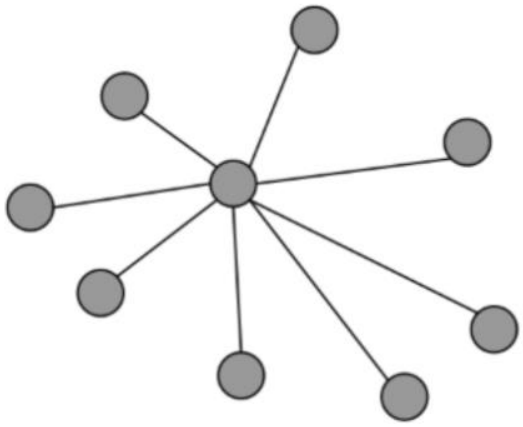
Homophily is:

- [ ] the observation that two words tend to be pronounced similarly if they are in a social network.
- [ ] the fear that two people connected in a social network may develop a blood disorder that interferes with clotting.
- [ ] the political movement that suggests that people should be allowed to marry people of the same sex.
- [x] the idea that nodes that are connected in a social network are also more likely to share other characteristics.
- [ ] the term used to refer to individuals who reside in the largest city in Pennsylvania.

Networks

How many hubs do we need to list in each node of a star network to satisfy the property that we can compute shortest path lengths efficiently?

1



LECCIÓN 18

Top Two

```
def top_two(graph, start, topX=2):
    heap = [((0, [start]), start)]
    best = defaultdict(list)
    best[start] = [(0, [start])]
    while heap:
        ((cost, path), node) = heapq.heappop(heap)
        for (other, weight) in graph[node].items():
            if other in path:
                continue

            new = (cost + weight, path + [other])
            best[other] = sorted(best[other] + [new])[:topX]
            if new in best[other]:
                heapq.heappush(heap, (new, other))

    return best
```