GROKKING ALGORITHMS

BINARY SEARCH

```
def binary_search(list, item):
    low = 0
    high = len(list) - 1
    while low <= high:
        mid = low + high // 2
        guess = list[mid]
        if guess == item:
            return mid
        if guess > item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

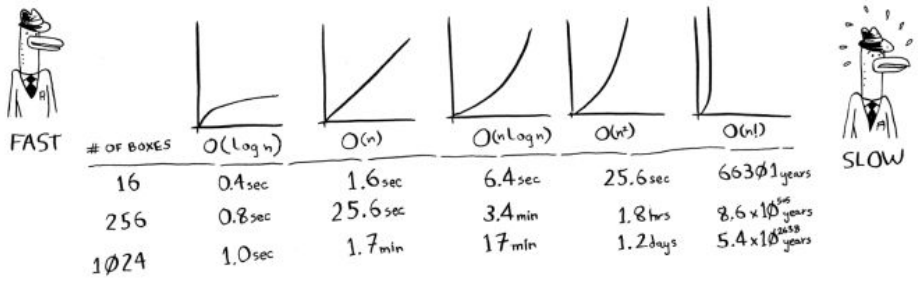
BIG O NOTATION

As the number of items increase, also the time requested to run the algorithm increases.
Big O notation lets you compare the number of operations

$O(n)$
↖ num. of operations

Big O austoscishes a worst case run time
(superior asymtotic limit)

EXAMPLES:
- $O(\log n)$ logarithmic time
- $O(n)$ linear time
- $O(n \log n)$
- $O(n^2)$ quadratic time
- $O(n!)$ fractional time



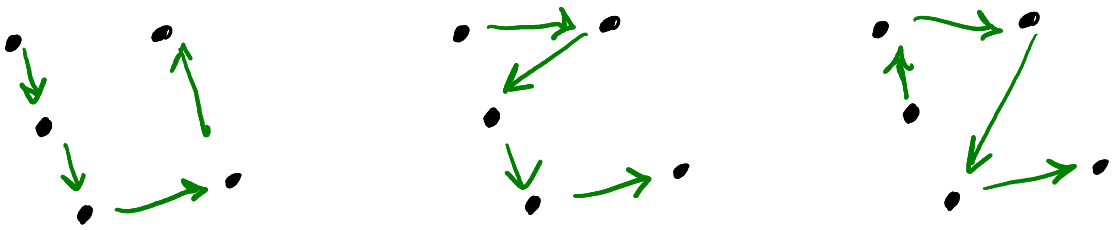
# OF BOXES	O(Log n)	O(n)	O(n Log n)	O(n²)	O(n!)
16	0.4 sec	1.6 sec	6.4 sec	25.6 sec	6630 1 years
256	0.8 sec	25.6 sec	3.4 min	1.8 hrs	8.6×10^{506} years
1024	1.0 sec	1.7 min	17 min	1.2 days	5.4×10^{2638} years

KEY TAKEAWAYS

- ALGO'S SPEED IS MEASURED IN N. OF OPERATIONS
- INPUT SIZE AND ALGO SPEED IS PROPORTIONAL

THE TRAVELLING SALESPERSON PROBLEM

5 CITIES, MIN. DISTANCE TO HIT THEM ALL?



FOR 5 CITIES ⇒ 120 PERMUTATIONS

...

15 CITIES ⇒ 1307 674 368 000 PERMS

FOR N CITIES ⇒ N! PERMUTATIONS

THERE ISN'T A SMART ALGORITHM FOR IT (CHECK GREEDY ALGOS)

ARRAYS AND LINKED LISTS

	ARRAYS	LISTS
READ	O(1)	O(n)
WRITE	O(n)	O(1)
INSERT	O(n)	O(1)
DELETE	O(n)	O(1)

ARRAYS ⇒ RANDOM ACCESS + ADJACENT ELEMENTS + SAME TYPE

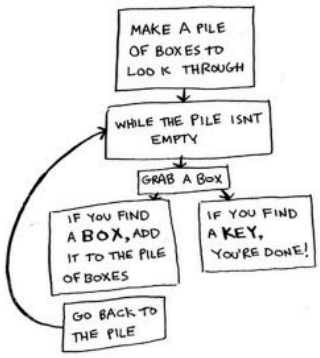
LINKED LISTS ⇒ SEQUENT. ACCESS + POINTER TO NEXT ELEMENT

SELECTION SORT

```
def find_smallest (array):
    smallest = array[0]
    smallest_index = 0
    for i in range (1, len(array)):
        if array[i] < smallest
            smallest = array[i]
            smallest_index = i
    return smallest_index

def selection_sort (array):
    new_array = []
    for i in range (len(array)):
        smallest = find_smallest (array)
        new_array.append (array.pop (smallest))
    return new_array
```

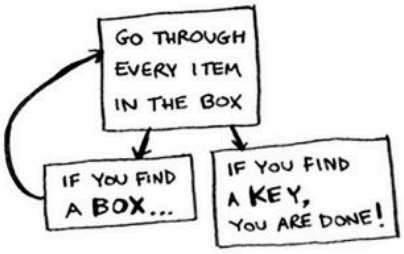
RECURSION

Here's one approach.



1. Make a pile of boxes to look through.
2. Grab a box, and look through it.
3. If you find a box, add it to the pile to look through later.
4. If you find a key, you're done!
5. Repeat.

Here's an alternate approach.



1. Look through the box.
2. If you find a box, go to step 1.
3. If you find a key, you're done!

```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(item)  <-- Recursion!
    elif item.is_a_key():
      print "found the key!"
```

RECURSION HAS TO BE USED WHEN IT MAKES THE SOLUTION CLEANER

MOST OF THE TIMES LOOPS HAVE BETTER PERFORMANCE

(BASE CASE + RECURSIVE CASE)

the function stop calling itself (AVOID ∞ LOOP)

The function calls itself

STACK: { • push • pop LIFO

CALL STACK

EVERYTIME YOU MAKE A FUNCTION CALL, THE OS SAVES THE VALUES FOR ALL THE VARIABLES FOR THAT CALL IN MEMORY.

WHEN YOU CALL A FUNCTION FROM ANOTHER FUNCTION, THE CALLING FUNCTION IS PAUSED IN A PARTIALLY COMPLETED STATE

CALL STACK w/ RECURSION

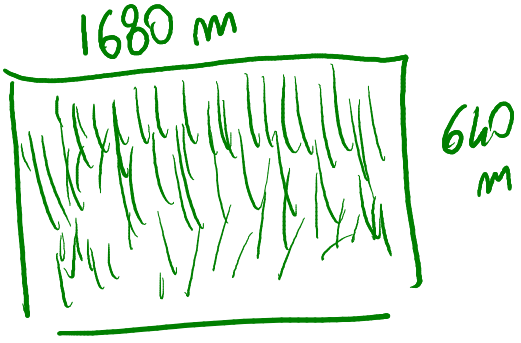
```
def factorial(x):
    if x == 1:
        return 1
    else
        return x * factorial(x-1)
```

EACH CALL TO factorial HAS ITS OWN COPY OF THE VARIABLE X

DIVIDE AND CONQUER

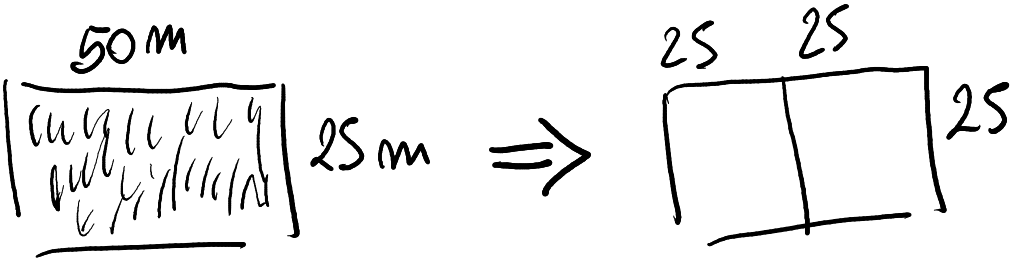
RECURSIVE TECHNIQUE FOR SOLVING PROBLEMS

BASIC EXAMPLE 1.

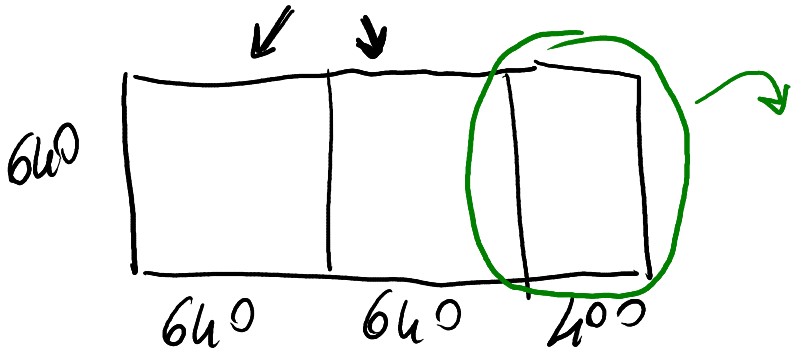


plot of land: dividing the farm evenly into square plots with plots as big as possible

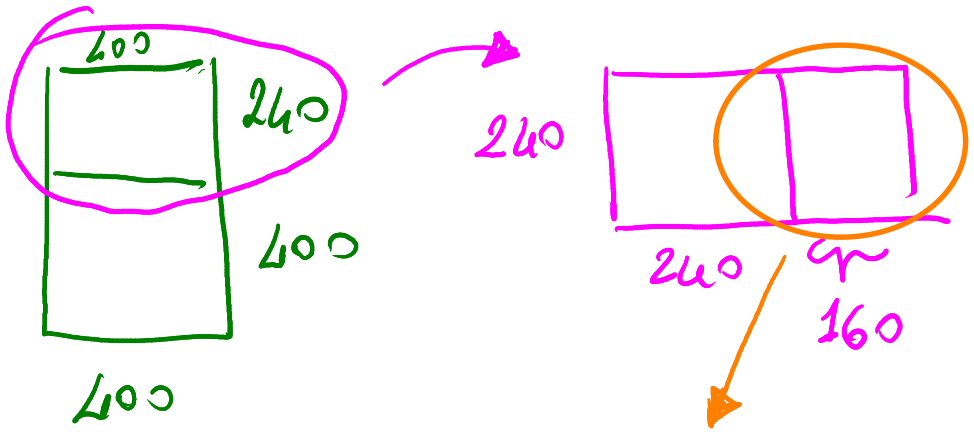
figuring the base case: one side is multiple of the other

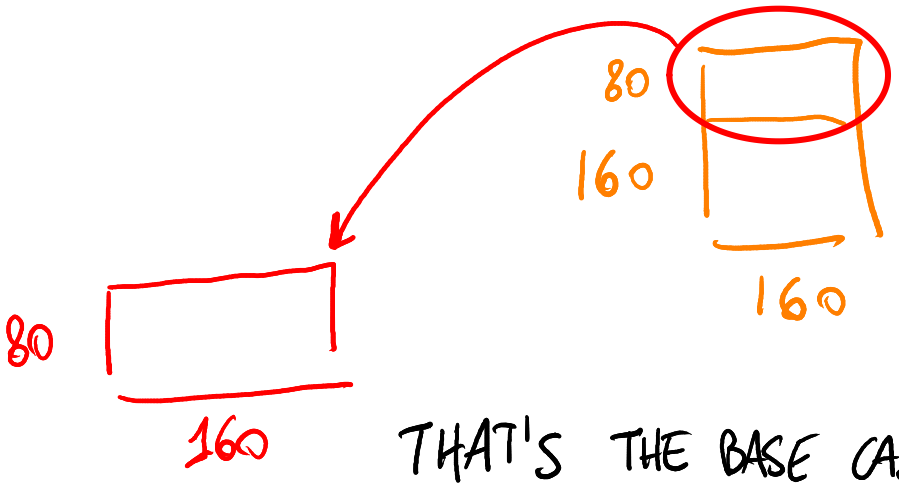


Let's apply the algorithm to the original plot

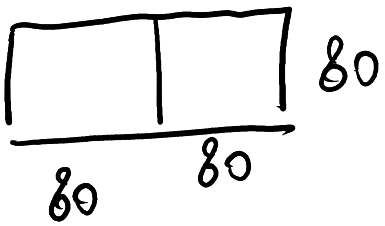


LET'S RE-APPLY THE ALGORITHM TO THIS SEGMENT (RECURSION)





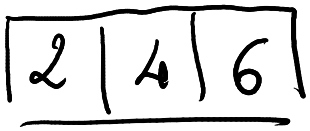
THAT'S THE BASE CASE!



SO, THE LARGEST SQUARE SIZE IS 80 × 80

BASIC EXAMPLE 2.

You're given an array of numbers



we want to add up all the numbers and return the total

BASE CASE

- the array has zero elements
- the array has one element

RECURSIVE CASE

sum (2 | 4 | 6) = 12

⇓

2 + sum (4 | 6) = 2 + 10 = 12

⇓ base case!

2 + 4 + sum (6) = 2 + 4 + 6 = 12

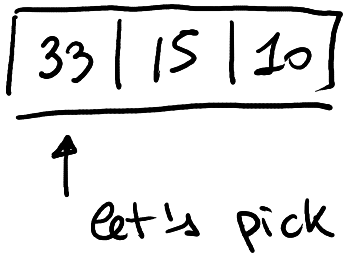
BINARY SEARCH : RECURSIVE APPROACH

```
Algorithm binarySearch(arr, low, high, target):
  if low <= high:
    mid = low + (high - low) // 2

    # Check if the target is present at the middle
    if arr[mid] == target:
      return mid

    # If the target is smaller than the middle element, then it can only be present in the left subarray
    elif arr[mid] > target:
      return binarySearch(arr, low, mid-1, target)

    # Else the target can only be present in the right subarray
    else:
      return binarySearch(arr, mid+1, high, target)

  # If we reach here, then the element was not present in the array
  return -1
```

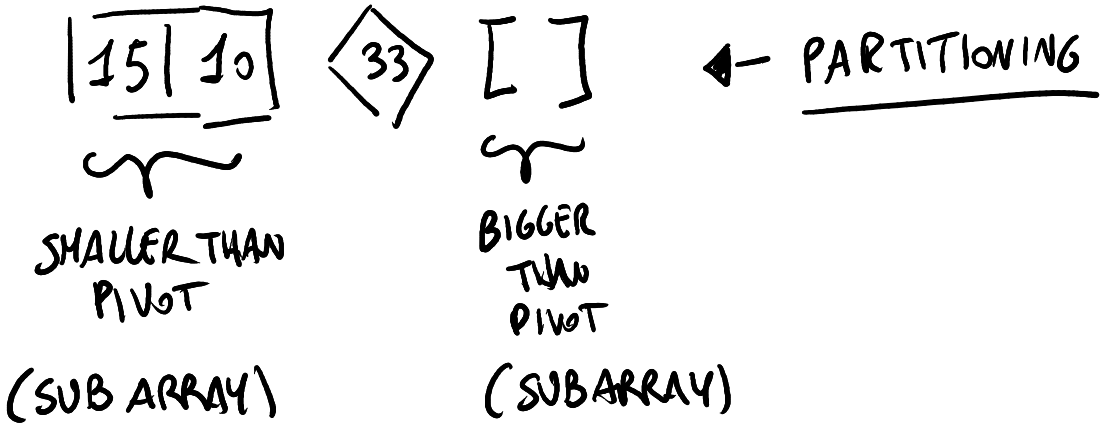
QUICKSORT

BASE CASE: the array has zero or one element
⇒ there's nothing to sort

RECURSIVE CASE:



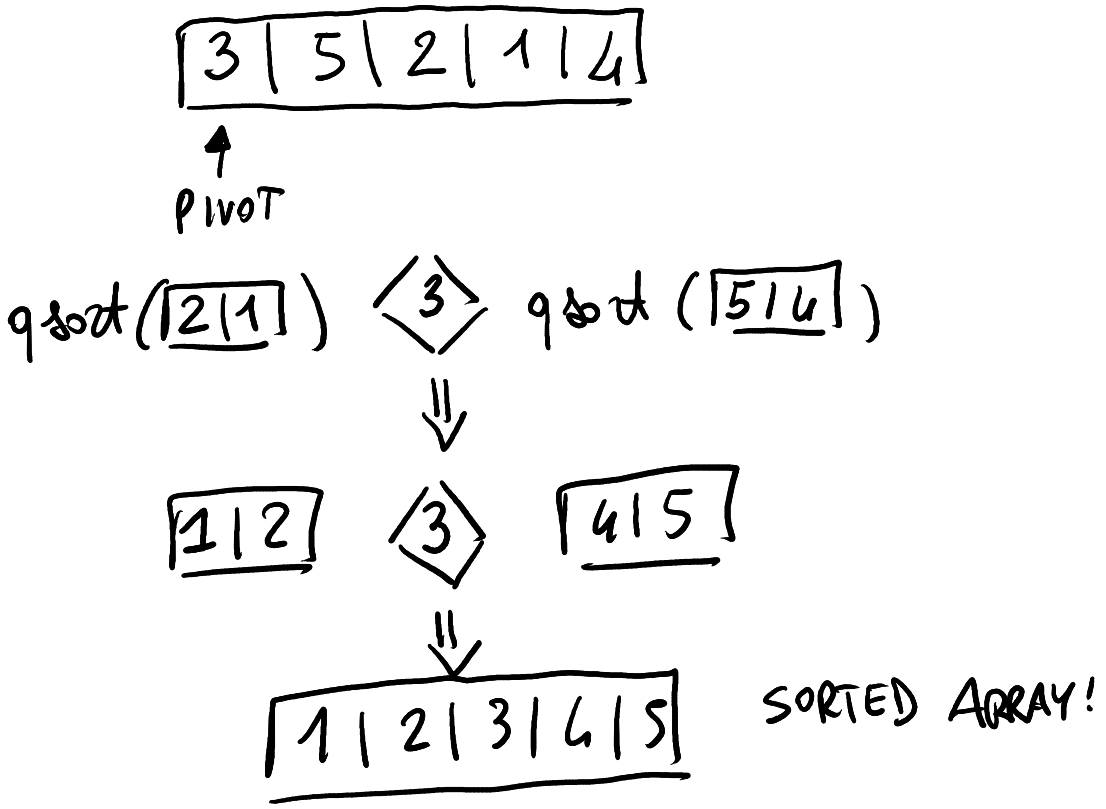
Now let's find the elements smaller / bigger than the pivot



⇒ Now we have to sort the subarrays

quicksort(15 | 10) = 10 | 15

LET'S INCREASE THE DIFFICULTY



FINALLY... QUICKSORT CODE

```
def quicksort(array):
  if len(array) < 2:
    return array          <-- Base case: arrays with 0 or 1 element are already "sorted."
  else:
    pivot = array[0]      <-- Recursive case
    less = [i for i in array[1:] if i <= pivot]   <-- Sub-array of all the elements less than the pivot
    greater = [i for i in array[1:] if i > pivot] <-- Sub-array of all the elements greater than the pivot
    return quicksort(less) + [pivot] + quicksort(greater)
```

QSORT performance depends on the PIVOT

⇒ QUICKSORT RUNS IN $O(n \log n)$ IN THE AVERAGE CASE

WORST CASE: $O(n^2)$ (ARRAY ALREADY SORTED)

BEST WAY TO CHOOSE THE PIVOT: RANDOM ELEMENT

HASH TABLES

HASH FUNCTIONS: taken a string in input, it gives back a number ⇒ it maps strings to numbers

THE MAPPING SHOULD BE

- CONSISTENT
- DIFFERENT FOR EACH STRING

THE HASH FUNCTION KNOWS HOW BIG THE ARRAY IS

HASH FUNCTION + ARRAY = HASH TABLE

OTHER NAMES: hash maps, maps, dictionary, associative arrays

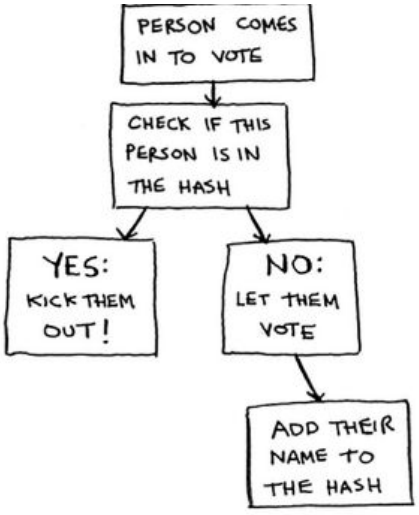
HASH TABLE ⇒ KEYS + VALUES

EG: NAMES / PRICE

HASH TABLES REAL WORLD APPLICATIONS

- PHONEBOOKS
- DNS Resolver

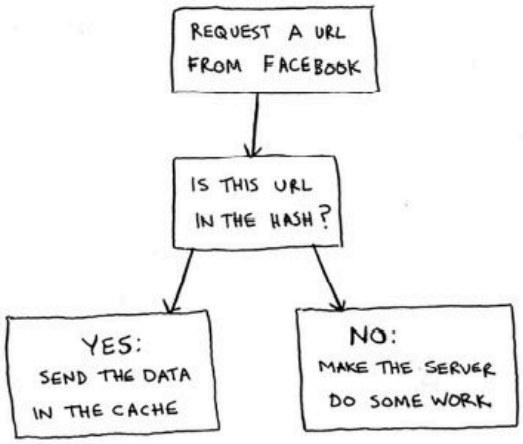
AVOIDING DUPLICATES:



Here's the code:

```
voted = {}

def check_voter(name):
  if voted.get(name):
    print "kick them ou
  else:
    voted[name] = True
    print "let them vote!"
```

OTHER HASH TABLE EXAMPLE : CACHE

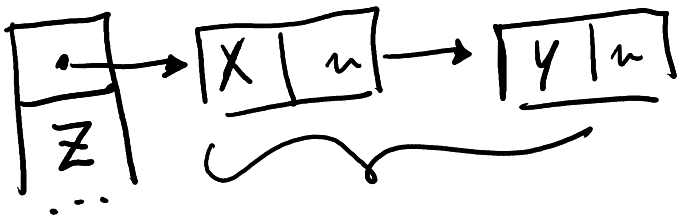


Here it is in code:

```
cache = {}

def get_page(url):
  if cache.get(url):
    return cache[url]  <-- Returns cached data
  else:
    data = get_data_from_server(url)
    cache[url] = data  <-- Saves this data in your cache first
    return data
```

HASH TABLES COLLISIONS: TWO KEYS ASSIGNED TO THE SAME ARRAY SLOT

AVOIDING #1: IF MULTIPLE KEYS MAP TO THE SAME SLOT, START A LINKED LIST AT THAT SLOT



NOW WE HAVE TO SEARCH THROUGH THE LINKED LIST

⇒ HASH FUNCTION IS REALLY IMPORTANT, IT SHOULD MAP KEYS EVENLY ALL OVER THE HASH

HASH TABLES: PERFORMANCE

	AVG CASE	WORST CASE
SEARCH	O(1)	O(n)
INSERT	O(1)	O(n)
DELETE	O(1)	O(n)

TO AVOID COLLISIONS

- GOOD HASH FUNCTION
- LOW LOAD FACTOR

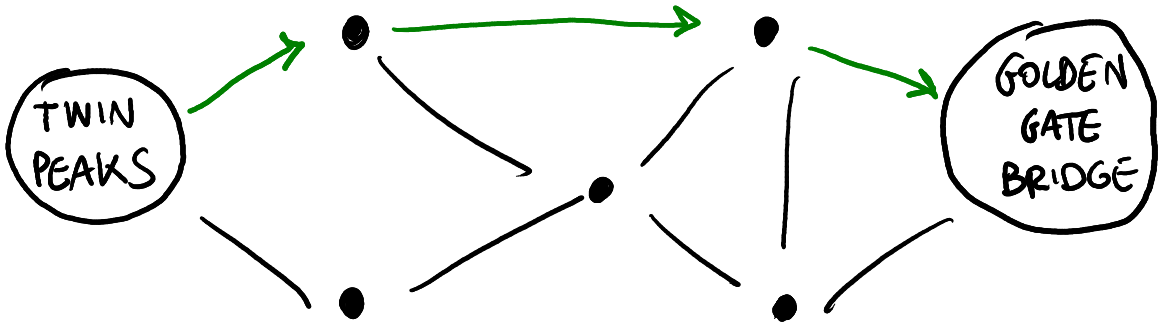
$$\text{LOAD FACTOR} = \frac{\#\text{ OF ITEMS IN HASH TABLE}}{\text{TOTAL NUMBER OF SLOTS}}$$

$$\text{LOAD FACTOR} = \frac{2}{5} = 40\%$$

LOAD FACTOR > 1 $\Rightarrow$ MORE ITEMS THAN SLOTS

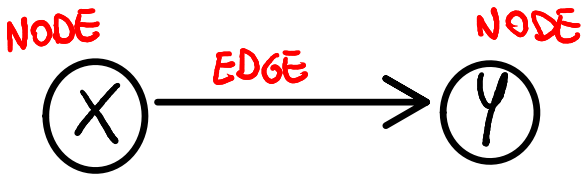
NOTE: LOAD FACTOR SHOULD NEVER BE > 0.07

INTRODUCTION TO GRAPHS



SHORTEST PATH : 3 STEPS

WHAT'S A GRAPH ? IT'S A STRUCTURE THAT MODELS A SET OF CONNECTIONS



DIRECTLY CONNECTED NODES ARE CALLED NEIGHBORS

BREADTH-FIRST SEARCH

IT CAN HELP ANSWER 2 TYPES OF QUESTIONS

1. IS THERE A PATH FROM NODE A TO NODE B ?
2. WHAT'S THE SHORTEST PATH FROM A TO B ?

⇒ WE NEED TO SEARCH IN THE ORDER THE NODES ARE ADDED, WE NEED A QUEUE (FIFO)

TWO OPERATIONS

- QUEUE (ADDING)
- DEQUEUE (TAKING OUT)

BFS: IDEA AND CODE IMPLEMENTATION

1. KEEP A QUEUE CONTAINING THE ELEMENTS TO CHECK
2. POP A ELEMENT OFF THE QUEUE
3. CHECK IF THAT ELEMENT IS THE ONE WE'RE LOOKING FOR
4. 1 IF IT IS IT WE'RE DONE
   4.2 IF IT'S NOT ADD ALL THEIR NEIGHBOR TO THE QUEUE
5. LOOP

```python
from collections import deque

def bfs(graph, start):
    # Create a deque for the queue and a set to keep track of visited nodes
    queue = deque([start])
    visited = set([start])

    while queue:
        # Pop the first node from the queue
        node = queue.popleft()

        # Process the node
        print(node)

        # Add all unvisited neighbors to the queue and mark them as visited
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
```

ADD HERE THE FUNCTION TO CHECK THE NODE

BREADTH-FIRST SEARCH RUNS IN $O(V+E)$ TIME

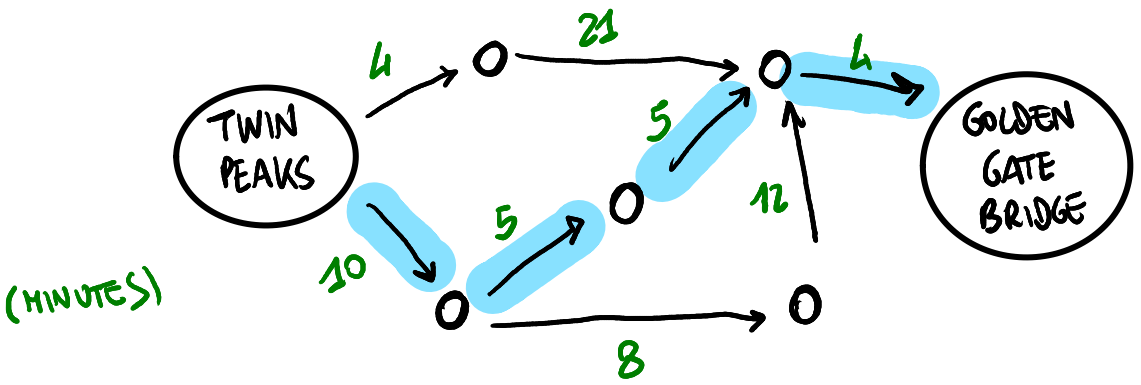
V: # OF VERTICES

E: # OF EDGES

DIJKSTRA'S ALGORITHM

Works for WEIGHTED GRAPHS and it's made for finding the FASTEST PATH.

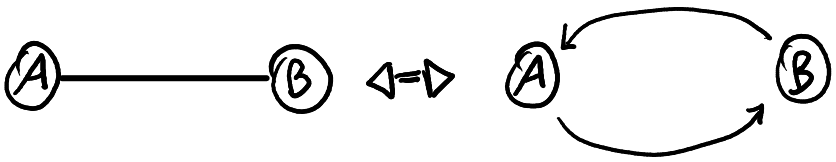
(DOES NOT WORK WITH NEGATIVE WEIGHTS)



STEPS FOR THE ALGORITHM

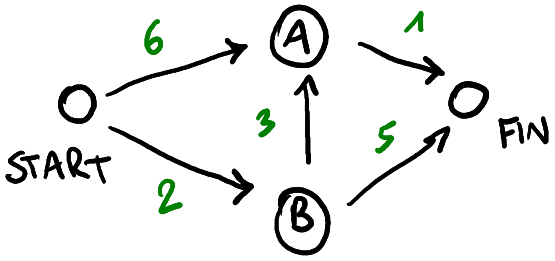
1. FIND THE CHEAPEST NODE
2. CHECK IF THERE'S A CHEAPER PATH TO THE NEIGHBORS OF THIS NODE. IF SO UPDATE THEIR COSTS
3. REPEAT FOR EVERY NODE
4. CALCULATE THE FINAL PATH

Note: UNDIRECTED GRAPH ⇒ CYCLE

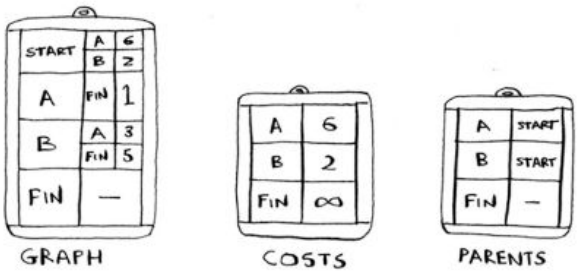


Note: FOR NEGATIVE WEIGHT EDGES USE THE BELLMAN-FORD ALGORITHM

DIJKSTRA'S ALGORITHM CODE IMPLEMENTATION



WE'RE GONNA NEED 3 HASH TABLES



Representing the weights:

```
> graph = {}
> graph['start'] = {}
> graph['start']['A'] = 6
> graph['start']['B'] = 2
[rest of the nodes / neighbors]
...
> graph['fin'] = {}
```

Storing the costs of each node

```
> infinity = float("inf")
> costs = {}
> costs['A'] = 6
> costs['B'] = 2
> costs['fin'] = infinity
```

Representing the parents

```
> parents = {}
> parents['A'] = 'start'
> parents['B'] = 'start'
> parents['fin'] = None
```

We're gonna also need an array of already processed

```
> processed = []
```

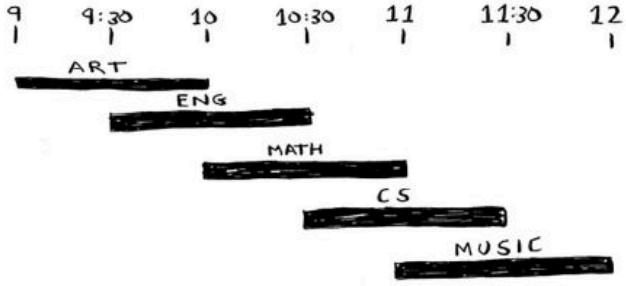
```
node = find_lowest_cost_node(costs)   <-- Find the lowest-cost node that you haven't processed yet.
while node is not None:               <-- If you've processed all the nodes, this while loop is done.
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys():        <-- Go through all the neighbors of this node.
        new_cost = cost + neighbors[n]    If it's cheaper to get to this neighbor
        if costs[n] > new_cost:           by going through this node ...
            costs[n] = new_cost       <-- ... update the cost for this node.
            parents[n] = node         <-- This node becomes the new parent for this neighbor.
    processed.append(node)            <-- Mark the node as processed.
    node = find_lowest_cost_node(costs)   <-- Find the next node to process, and loop.


def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None
    for node in costs:                <-- Go through each node.
        cost = costs[node]                If it's the lowest cost so far and hasn't been processed yet ...
        if cost < lowest_cost and node not in processed:
            lowest_cost = cost        <-- ... set it as the new lowest-cost node.
            lowest_cost_node = node
    return lowest_cost_node
```

GREEDY ALGORITHMS AND NP-COMPLETE PROBLEMS

The classroom scheduling problem

CLASS	START	END
ART	9 AM	10 AM
ENG	9:30 AM	10:30 AM
MATH	10 AM	11 AM
CS	10:30 AM	11:30 AM
MUSIC	11 AM	12 PM

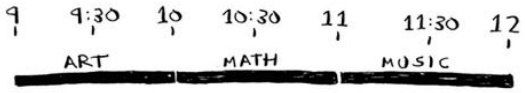
You can't hold *all* of these classes in there, because some of them overlap.



ALGORITHM SOLUTION

1. Pick the class that ends the soonest (that will be the first class you'll hold)
2. Again, pick the class that ends the soonest
3. REPEAT

RESULT:



GREEDY ALGORITHM: AT EACH STEP YOU PICK THE <u>LOCALLY OPTIMAL SOLUTION</u>

THE SET COVERING PROBLEM

Suppose you're starting a radio show and you want to reach listeners in all the states.
You have a list of stations, each station covers a region and there's overlap.
Question: What's the smallest set of stations you can play to cover all the states?

1. There are 2^n possible subsets
2. From these, pick the smallest number of stations that covers all the states

$O(2^n) \Rightarrow n = 32$, we would need 13.6 years !!

THE SET COVERING PROBLEM: APPROXIMATION SOLUTIONS

STEPS:

1. Pick the station that covers the most states that haven't been covered yet. It's OK if the station covers some states already covered
2. Repeat until all the states are covered

THIS SOLUTION RUNS IN $O(n^2)$

APPROXIMATION ALGORITHMS

- HOW FAST THEY ARE
- HOW CLOSE THEY ARE TO THE SOLUTION

NP COMPLETE PROBLEMS AND TRAVELING SALESPERSON

LET'S START SMALL: SUPPOSE WE HAVE 2 CITIES

① STARTING AT MARIN:

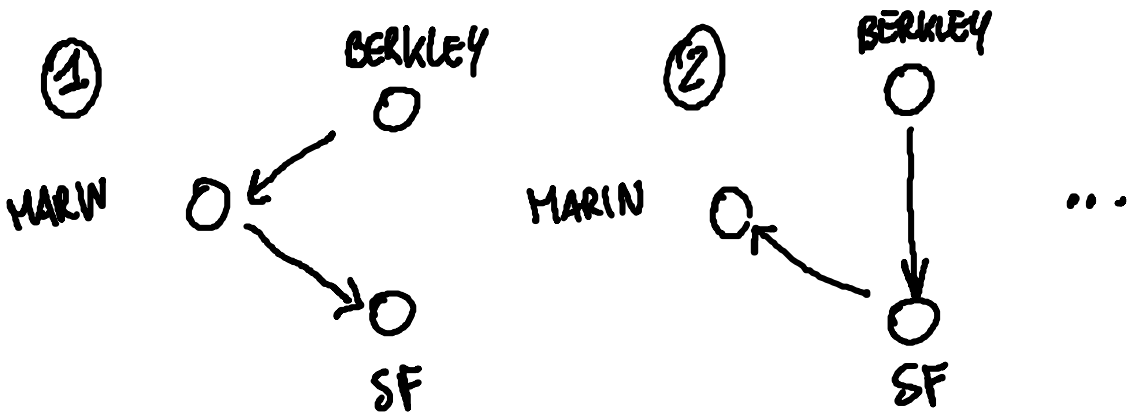
MARIN TO S. FRANCISCO

② STARTING AT S. FRANCISCO

SAN FRANCISCO TO MARIN

2 CITIES ⇓ 2 POSSIBLE ROUTES

NOW SUPPOSE WE HAVE 3 CITIES



2 ROUTES FOR EACH CITY ⇒ 6 POSSIBLE ROUTES

NOW 4 CITIES : SIX ROUTES FOR EACH CITY

$6 \cdot 4 = 24$ ROUTES

...

N CITIES ⇒ N! POSSIBLE ROUTES

GOOD APROXIMATION :

- ARBITRARY PICK A START CITY
- EACH TIME THE T.S HAS TO PICK THE NEXT CITY, PICK THE CLOSEST UNVISITED CITY

THE TRAVELING SALESPERSON, LIKE THE SET COVERING PROBLEM, IS NP-COMPLETE

We should calculate every possible solution and pick the smallest / shortest one

giovedì 23 febbraio 2023 11:20

HOW TO TELL IF A PROBLEM IS NP-COMPLETE

THERE IS NO EASY WAY TO TELL, BUT THERE ARE SOME GUEAWAYS:

1. THE ALGORITHM RUNS WELL WITH FEW ELEMENTS, BUT SLOWS DOWN WITH MORE
2. 'ALL COMBINATIONS OF X' ⇒ NP-PROBLEM USUALLY
3. SEQUENCE OF ITEMS + HARD TO SOLVE ⇒ NP-PROBLEM
   OR
   SET
4. IF YOU CAN RESTATE THE PROBLEM AS THE SET COVERING OR THE TRAVELING SALESPERSON PROBLEM

DYNAMIC PROGRAMMING

LET'S REVISIT THE KNAPSACK PROBLEM:
"You're a thief with a knapsack that can carry only 4lb of goods"

⇒ SIMPLE SOLUTION: TRY EVERY SET OF GOODS AND FIND THE SET THAT GIVES THE MOST VALUE

[32 ITEMS ⇓ 4 BILLION SETS]

⇓

IT WORKS BUT IT'S REALLY SLOW!
IT RUNS IN $O(2^m)$

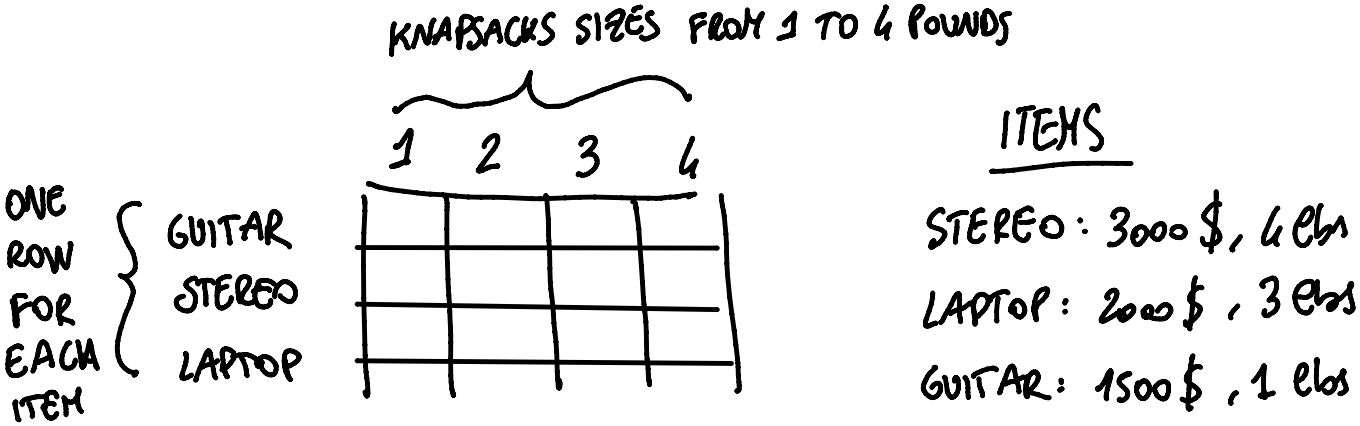
⇒ FASTER SOLUTION: DYNAMIC PROGRAMMING

⇓

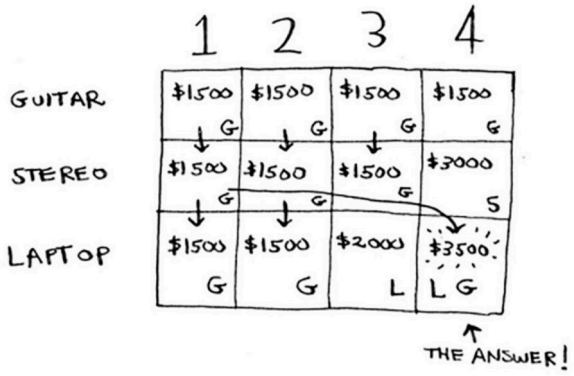
- START BY SOLVING SUBPROBLEMS (SMALLER KNAPSACKS)
- BUILD UP TO SOLVING THE BIG PROBLEM

LET'S TRY SOLVING THE KNAPSACK PROBLEM WITH D.P

EVERY D.P ALGO STARTS WITH A GRID



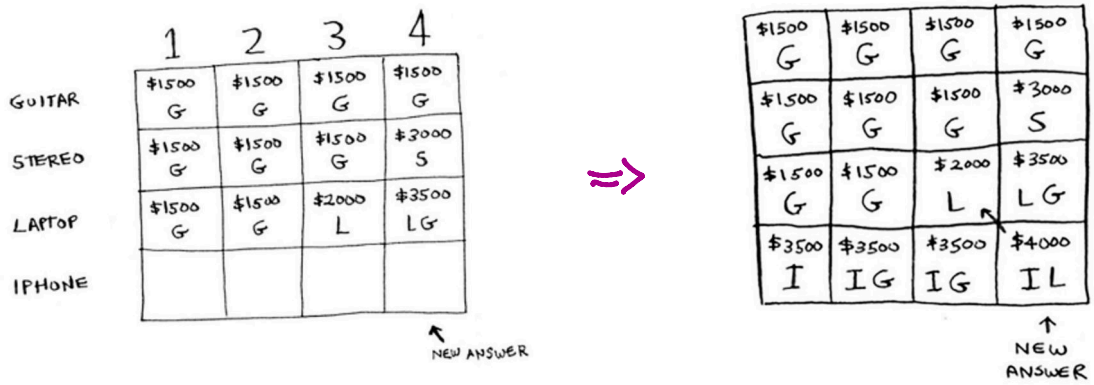
LET'S FILL THAT GRID:



THE FORMULA:

$$CELL[i][j] = \max \begin{cases} \text{THE PREVIOUS MAX}: & CELL[i-1][j] \\ \text{CURRENT ITEM} + \text{REMAINING SPACE}: & CELL[i-1][j - \text{ITEM WEIGHT}] \end{cases}$$

WHAT IF YOU ADDED A NEW ITEM TO STEAL? IPHONE: $2000, 1 lb



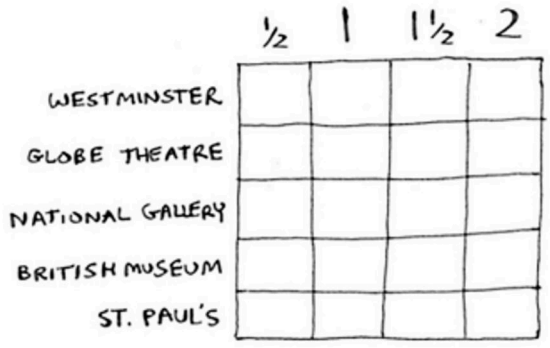
FAQ

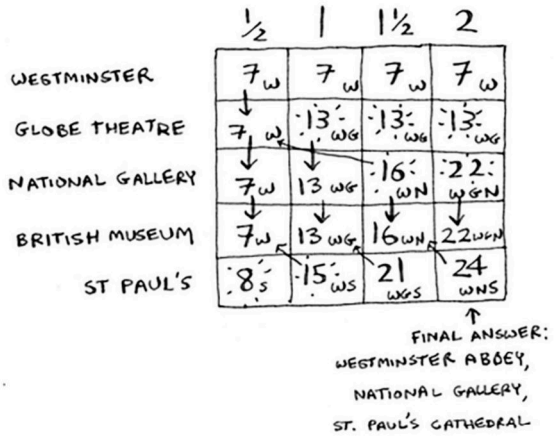
- The answer doesn't change if you change order of rows
- Filling the grid column-wise instead of row wise could make a difference

- Filling the grid column-wise instead of row wise could make a difference
- If we add a smaller item (say 0.5 lbs), we have to account for finer granularity

ANOTHER EXAMPLE: TRAVEL ITINERARY

ATTRACTION	TIME	RATING
Westminster Abbey	1/2 DAY	7
Globe Theater	1/2 DAY	6
National Gallery	1 DAY	9
British Museum	2 DAY	9
St. Paul	1/2 DAY	8





A BIGGER PROBLEM: LONGEST COMMON SUBSTRING

LET'S REVIEW DYNAMIC PROGRAMMING'S TAKEAWAYS

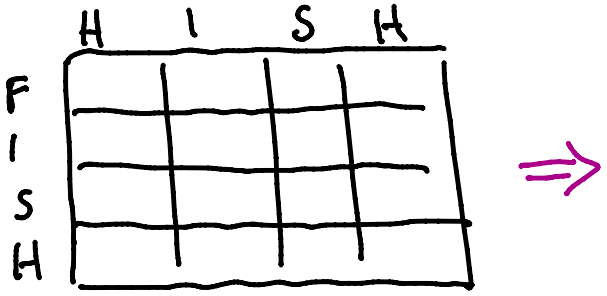
- Useful when you're trying to optimize something
- Useful for problems that can be broken into discrete subproblems that don't depend on each other

SOME GENERAL TIPS

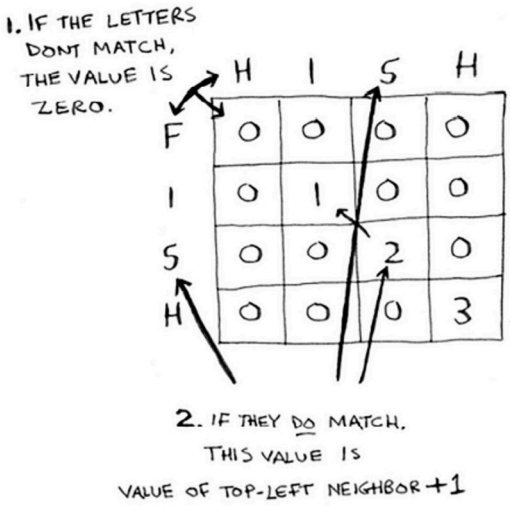
- Every DP solution involves a grid
- The values in the cell are usually what we're optimizing
- Each cell is a subproblem

The problem: you search for 'HISH' in a online dictionary
Which word should it display
FISH or VISTA ?

The grid could look like this



THE SOLUTION IS THE LARGEST NUMBER IN THE GRID

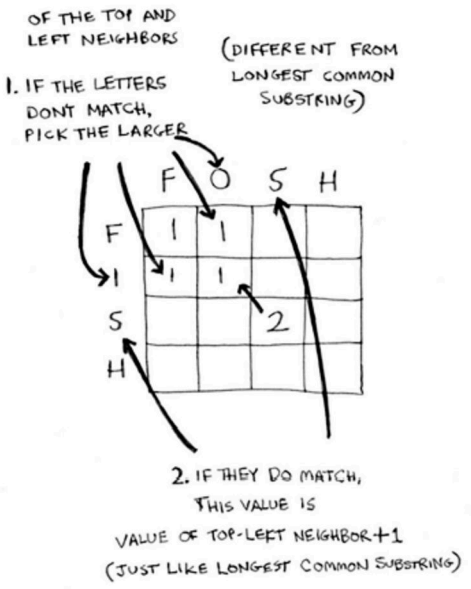


LONGEST COMMON SUBSEQUENCE

Suppose you've typed 'FOSH'
What did you mean, FISH or FORT?

Using the precedent algorithm, both return VALUE: 2

THE SUBSEQUENCE ALGORITHM:



{SOME REAL WORLD EXAMPLES}

- Similarities in DNA
- Differences between two files
- Similarities between two strings (LEVENSHTEW DISTANCE)

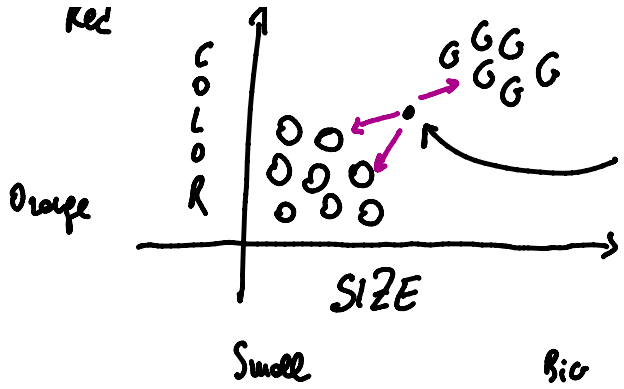
K-NEAREST NEIGHBORS

Algorithm used to build a classification system

EXAMPLE: ORANGES VS GRAPEFRUITS

Red

PARENTS

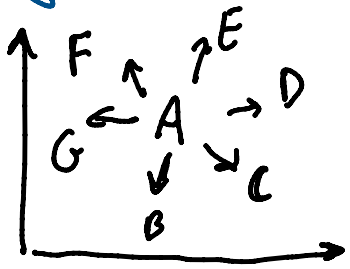


The mysterious fruit has more ORANGE-percents, so probably it is an ORANGE

The algorithm is pretty simple

1. You get a new fruit to classify
2. You look at its nearest neighbors
3. Final conclusions

Building a Recommendations System



The recommendation works with the reviews of each user

	A	B	C
COMEDY	3	4	2
ACTION	4	3	5
DRAMA	4	5	1
HORROR	1	1	3
ROMANCE	4	5	1

$A \rightarrow (3, 4, 4, 1, 4)$

We calculate the distance between the points in 5 DIMENSIONS

$$\Rightarrow \sqrt{(a_1-a_2)^2+(b_1-b_2)^2+(c_1-c_2)^2+(d_1-d_2)^2+(e_1-e_2)^2}$$

$$\overline{AB} = \sqrt{1+1+1+0+1} = 2$$

They're pretty close, so they could like the same things

REGRESSION

We want to guess how F will like a certain movie.
Take the five people closest to A.

A : 5
B : 4
C : 4
D : 5
E : 3

AVG. RATING : 4.2 ← PREDICTED VALUE

TWO BASIC THINGS TO DO WITH KNN

- CLASSIFICATION : categorization into a group
- REGRESSION : predicting a response

ANOTHER EXAMPLE: Running a Bakery

How to predict how many loaves to make for today ?
Set of features
- Weather on a scale of 1 to 5 (1 = bad, 5 = great)
- Weekend or holiday (1), otherwise (0)
- Is there a game on (1), otherwise (0)

We have a <u>past history</u>

A. (5, 1, 0) = 300 loaves
B. (3, 1, 1) = 225 loaves
C. (1, 1, 0) = 75 "
D. (4, 0, 1) = 200 "
E. (4, 0, 0) = 150 "
F. (2, 0, 0) = 50 "

Today is a weekend day with good weather

=> (4, 1, 0) = ? loaves

Let's find the closest neighbors

A. 1 ←
B. 2 ←
C. 9
D. 2 ←
E. 1 ←
F. 5

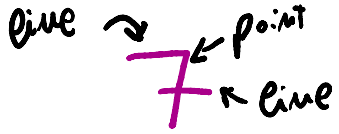
TAKING THE AVERAGE 218.75

Distance formula is a good one, but the best / common formula used is COSINE SIMILARITY, that composes the angles of two vectors

When working with KNN, it's really important to pick the right features to compare against (FEATURES THAT DIRECTLY CORRELATES AND FEATURES THAT DON'T HAVE A BIAS)

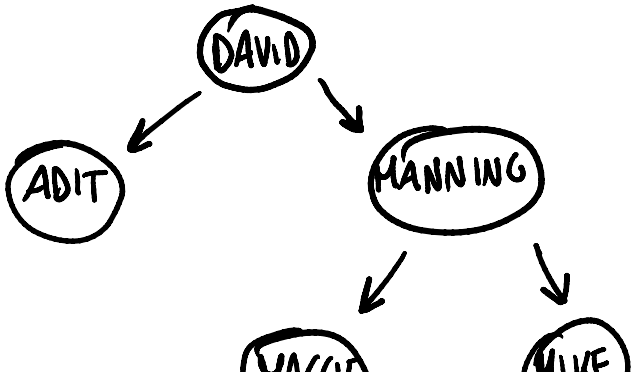
INTRODUCTION TO MACHINE LEARNING FROM KNN

Optical Character Recognition



1. Go through a lot of images of numbers and extract features of those numbers (TRAINING)
2. When getting a new image extract the features of that image and see what its nearest neighbors are

BINARY SEARCH TREES



FOR EVERY NODE, THE NODES TO ITS LEFT ARE SMALLER, AND THOSE TO ITS RIGHT ARE LARGER

BST: Performance

	ARRAY	BST
SEARCH	O(log n)	O(log n)
INSERT	O(n)	O(log n)
DELETE	O(n)	O(log n)

DOWNSIDES:
- Don't get random access
- The tree has to be balanced

AUTO-BALANCING BST ARE THE RED-BLACK TREES

INVERTED INDEXES

How a search engine works: Suppose you have 3 web pages

Let's build a hash table for this content

KEYS: WORDS

VALUES: THE PAGE EACH WORD APPEARS ON

{
HI : A, B,
THERE : A, C,
ADIT : B,
WE : C,
GO : C,
}

HASH TABLE THAT MAPS WORDS TO PLACES THEY APPEAR

⇓

INVERTED INDEX

THE FOURIER TRANSFORM

"Given a song, the transform can separate it into individual frequencies"

The transform tells you exactly how much each note contributes to the overall song ⇒ we can get rid of useless notes (MP3)

PARALLEL ALGORITHMS

Transform algorithms to make them run on MULTIPLE PROCESSORS

For example: parallel quicksort ~ $O(n)$
instead of quicksort ~ $(O \; n \log n)$

Parallel algorithms are hard to design and it's also hard to make sure they work correctly.
Also, twice the cores $\not\Rightarrow$ twice the performance.
Reasons can be OVERHEAD IN MANAGEMENT and LOAD BALANCING

MAP REDUCE

It's a distributed algorithm that runs across different machines

It's built up from two simple ideas: MAP and REDUCE functions

MAP FUNCTION: it takes an array and applies the same function to each item in the array
(from one array to another)

EX. DOUBLING ITEMS

```
> a221 = [1,2,3,4,5]
> a222 = map(lambda x: 2*x, a221)
```

REDUCE FUNCTION: it reduces a whole list of items to one item
(from one array to a single item)

EX: SUM UP ELEMENTS

```
> a221 = [1,2,3,4,5]
> reduce(lambda x,y: x+y, a221)
```

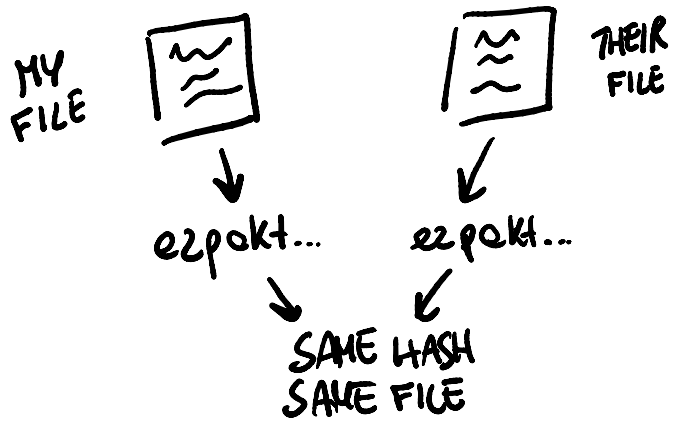
BLOOM FILTERS

They are PROBABILISTIC DATA STRUCTURES. They give you an answer that could be wrong, but it's probably correct.
Bloom Filters take up very little space, differently from Hash Tables that would have to memorize every item

Note: false positives are possible
false negatives aren't possible

The Secure Hash Algorithm (SHA)

Given a String, it gives you a hash for that string.
SHA generates a different hash for every string
You can use SHA to tell if two files are the same:



SHA is also useful when you want to compare strings without revealing what the original string was.

Another SHA interesting feature is its LOCAL INSENSITIVITY: if you change just one character of a string and regenerate the hash, it's totally different.

DIFFIE-HELLMAN KEY EXCHANGE

It answers the question "How do you encrypt a message so it can only be read by the person you sent the message to?"
This algorithm solves two problems
1. Both parties don't need to know the cipher
2. Encrypted messages are extremely hard to decode

There are TWO KEYS: the PUBLIC and the PRIVATE

You can do anything with it
Used to encrypt

Used to decrypt