# Grokking Algorithms: An Illustrated Guide for Curious Programmers

▼ Binary search

**Search problem:** to find element *x* in a given list *i*, should i start from the beginning or start near the middle and split each step up and half?

This is exactly what binary search is. It's input is a **sort list of elements** with the output being the **index of where that element is located**.

Simple (stupid) search is essentially when we go through every single index in the list up until you find the desired element.

> *If my number was 99, it could take you 99 guesses to get there!*

The better technique for counting? **Start with 50.** If that number's too low, then you just eliminated 50% of all the numbers!

Your next range is between 50-100. What's the middle of that? 75. **Too high? Then cut again?**

63? No. **57? YES!**

You just guessed the number within 7 steps

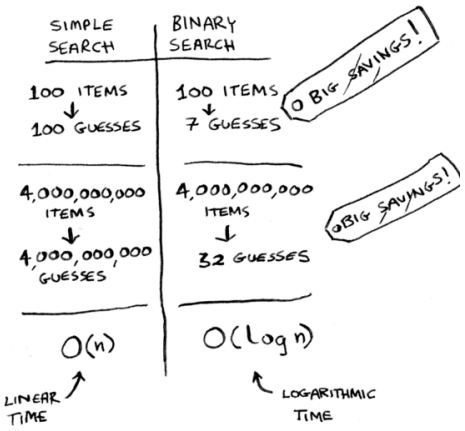
> *Eliminate half the numbers every time with binary search.*

💡 For any list of ***n,*** binary search will take ***log_2_n*** steps to run in the worst case, whereas simple search will take ***n*** steps.

```
def binary_search(list, item):
  low = 0
  high = len(list)-1
  while low <= high:
    mid = int((low + high)/2)
    guess = list[mid]
    if guess == item:
      return mid
    if guess > item:
      high = mid - 1
    else:
      low = mid + 1
  return None
my_list = [1, 3, 5, 7, 9]
print(binary_search(my_list, 3)) # => 1
print(binary_search(my_list, -1)) # => None
```

▼ Running time

Whenever the maximum amount of guesses is equal to the size of the list, we generally define this to be **linear time**




Whenever we want to determine how fast an algorithm is, we generally use something known as **Big O Notation.**

> *That's why it's not enough to know how long an algorithm takes to run—you need to know how the running time increases as the list size increases.*

Big O Notation tells you how fast an algorithm is. For example, let's say that you have a list of size ***n.*** The run time in Big O Notation is ***O(n)***.

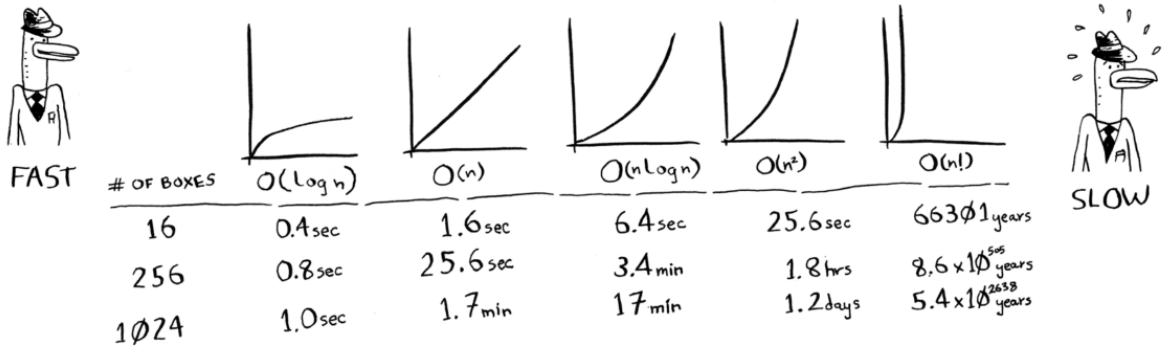
Notice that Big O Notation doesn't ask for the speed in seconds. **Big O notation lets you compare the number of operations. It tells you how fast the algorithm grows.**

What does that look like for Binary search? **O(log_2_n)**

**Note:** Big O Notation establishes a **worst-case run time.** The worst case is that you iterate through every element until you find the desired one, the worst case is that you have to apply binary search indefinitely until you find your desired element.

**Common Big O Run Times**

| Big O Notation | Known as | Examples |
|---|---|---|
| O(log_n) | Log time | Binary search |
| O(n) | Linear time | Simple search |
| O(n*log_n) | Fast sort algorithm | Quicksort |
| O(n^2) | Slow sorting algorithm | Selection sort |
| O(n!) | Really slow sorting algorithm | Travelling salesperson! |



▼ Lists + Arrays

Think of your computer as a giant set of drawers with each drawer having an address.

Every time you want to store an item in memory, you generally will ask the computer for some space (drawers). It then returns an address where you can store your items.

The 2 ways we store multiple items right now are **arrays and lists.**

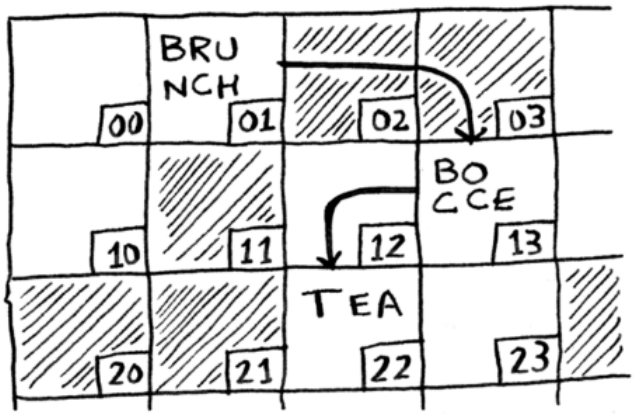
Adding items every single time to an array is very difficult. Think of it like the movies: if you sit with your friends and a friend shows up [but there are no spots for them], you have to move everyone to a new row. And what happens if another one shows up? **You have to move again!**

Adding an item to a computer is very slow. The easiest way we solve this is through "holding seats." **Even if you have only 3 seats, you can ask the computer for 10 slots, just in case.**

The main problem with doing something like this is mainly the fact that you have extra slots that you might've asked for but haven't used. On the other hand, **you might have more than 10 items and would have to move anyway.**

We can solve this through the principle of **Linked Lists.**

Each item stores the address of the next item in the list. A bunch of random random memory addresses are linked together.




Adding an item to this linked list is quite easy; **you just simply stick it anywhere in the memory and store the address with the previous item. This allows you to never have to move your items**

If linked lists are better than arrays (above), what would the purpose of arrays even be?

The problem with linked lsits is that, "***if you're going to keep jumpig around, linked lists are terrible. Why? If I wanted to get to the last item in a linked list, I have to go through EVERY SINGLE ITEM because that's what gives me the address to get to the next item."***

Arrays are different. You actually are given the address for every single item in the array.

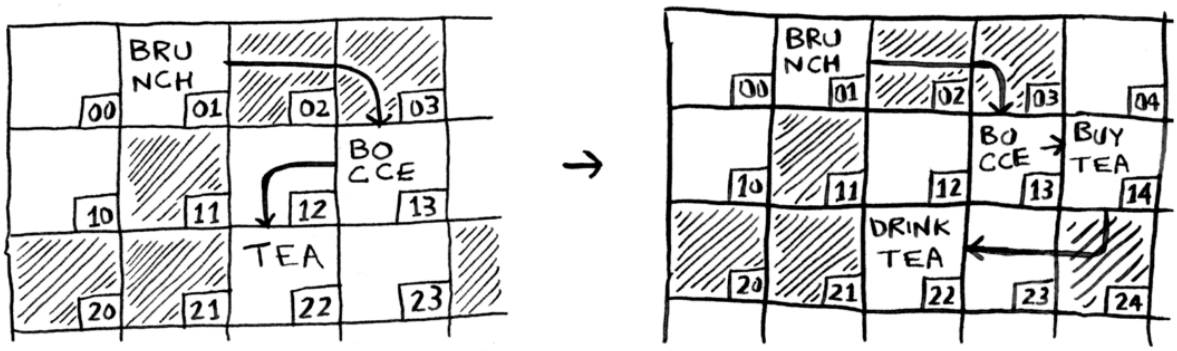
In this case, you'd use an array when you want to **read random elements because you can look up any element in the array.**

| | ARRAYS | LISTS |
|---|---|---|
| READING | O(1) | O(n) |
| INSERTION | O(n) | O(1) |

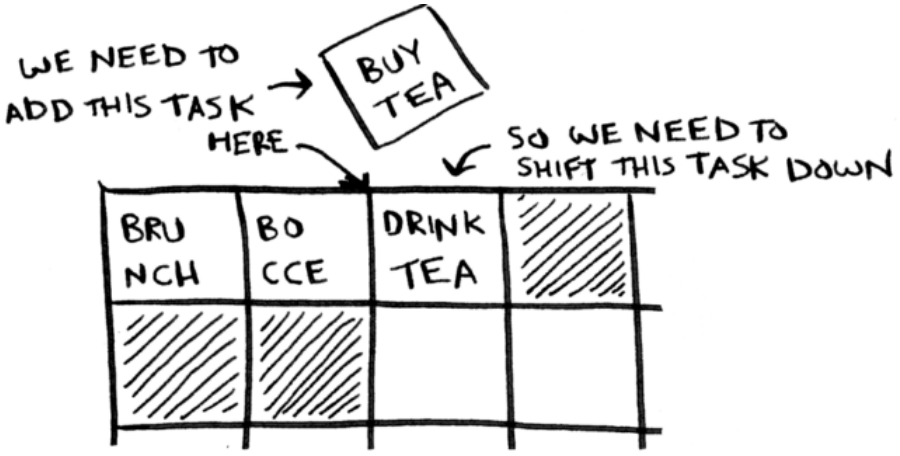
O(n) = LINEAR TIME
O(1) = CONSTANT TIME

Why does it take O(n) time to insert an element into an array?

**Insertion:** generally with lists, insertion is quite easy. You just simply change the address the previous element points to.




But in the case of an array, you need to shift all the elements down.




**Verdict: lists are generally better when it comes to inserting elements into the middle.**

What about deletion? **Again, lists are better.** You simple just change what the previous element points to while with arrays, everything needs to be moved up when you delete the element.

| | ARRAYS | LISTS |
|---|---|---|
| READING | O(1) | O(n) |
| INSERTION | O(n) | O(1) |
| DELETION | O(n) | O(1) |

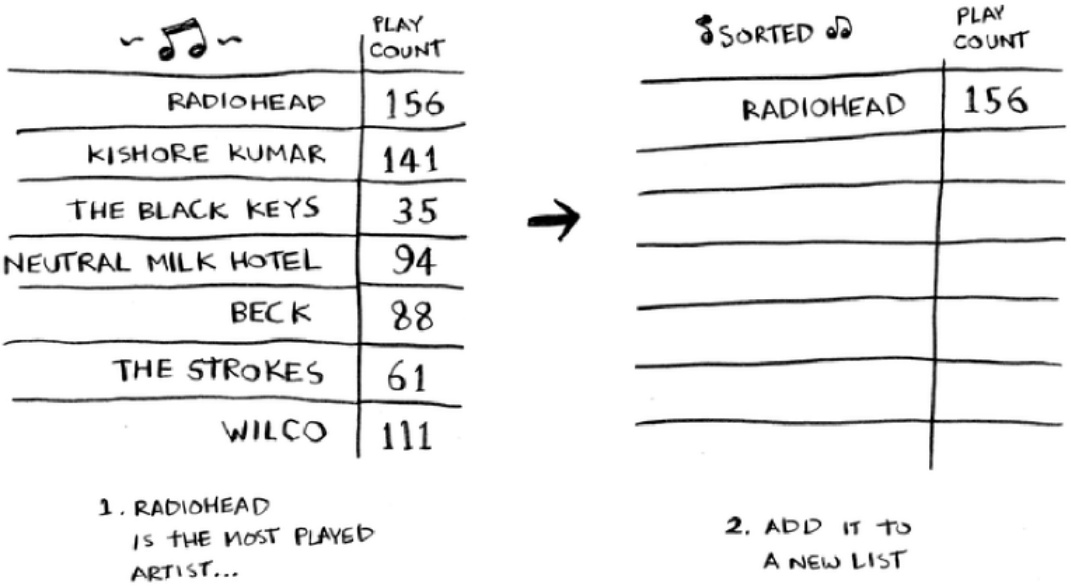
**So, when do you choose between arrays vs. lists?**

Arrays are geenrally used more because they can allow for **random access**.

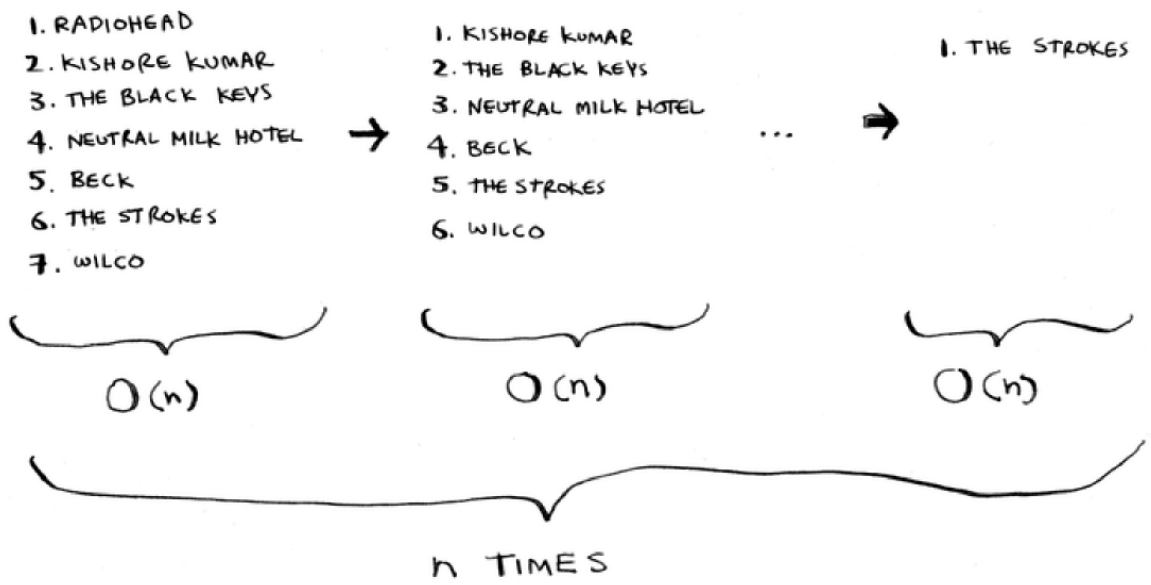
We generally have 2 types of access: **random access** and **sequential** [one by one] **access.** Linked lists can only do sequential access while random allows you to jump directly to your desired element.

▼ Selection sort

Let's say that I'm sorting music




I go from choosing the most played songs all the way down to the least. I start off with my original list and then append all of that (in ordered form) to this new list.




As we do this over and over again, every single time, this overall process ends up taking **O(n*n) or O(n*2) time**

Selection sort is quite neat, but not really that fast. Quicksort is a lot fast, it only takes **O(n log n) time!**

```
def findSmallest(arr):
  smallest = arr[0]
  smallest_index = 0
  for i in range(1, len(arr)):
    if arr[i] < smallest:
      smallest = arr[i]
      smallest_index = i
  return smallest_index
def selectionSort(arr):
  newArr = []
  for i in range(len(arr)):
    smallest = findSmallest(arr)
    newArr.append(arr.pop(smallest))
  return newArr
print selectionSort([5, 3, 6, 2, 10])
```

▼ Recursion

**It's where a function calls itself.**

```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(item)
    elif item.is_a_key():
      print “found the key!”
```

Recursion is generally used when it makes the existing solution a lot clearer and easier to understand.

**There is no performance benefit to using recursion!** In fact, loops may sometimes be better for performance.

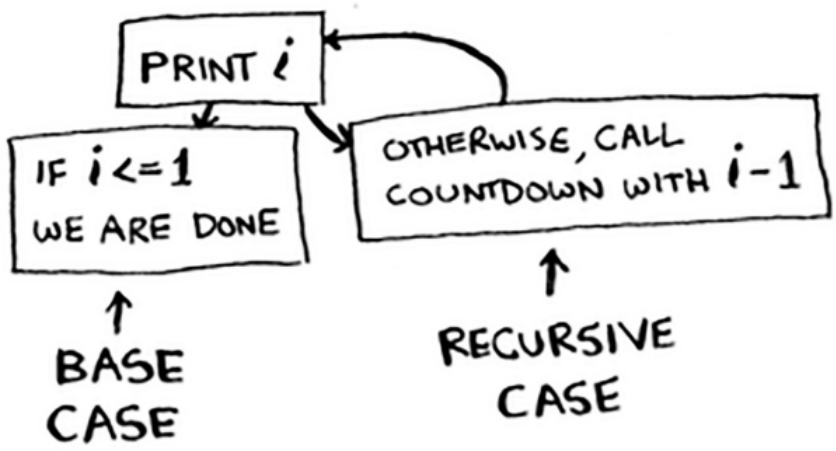
> *“Loops may achieve a performance gain for your program. Recursion may achieve a performance gain for your programmer. Choose which is more important in your situation!”*

What's the main problem with recursion? **You might get stuck in an infinite loop.**

That's why most recursive functions have a **"recursive case"** so that function knows when to stop calling itself.

```
def countdown(i):
  print i
  if i <= 0:  <------ Base case
    return
  else:  <------ Recursive case
    countdown(i-1)
```

Now the function works as expected. It goes something like this.




Computers generally use a stack known as the **call stack.** When you call on a function, your computer allocates a box of memory for that function call with "sub-boxes" for variable names.

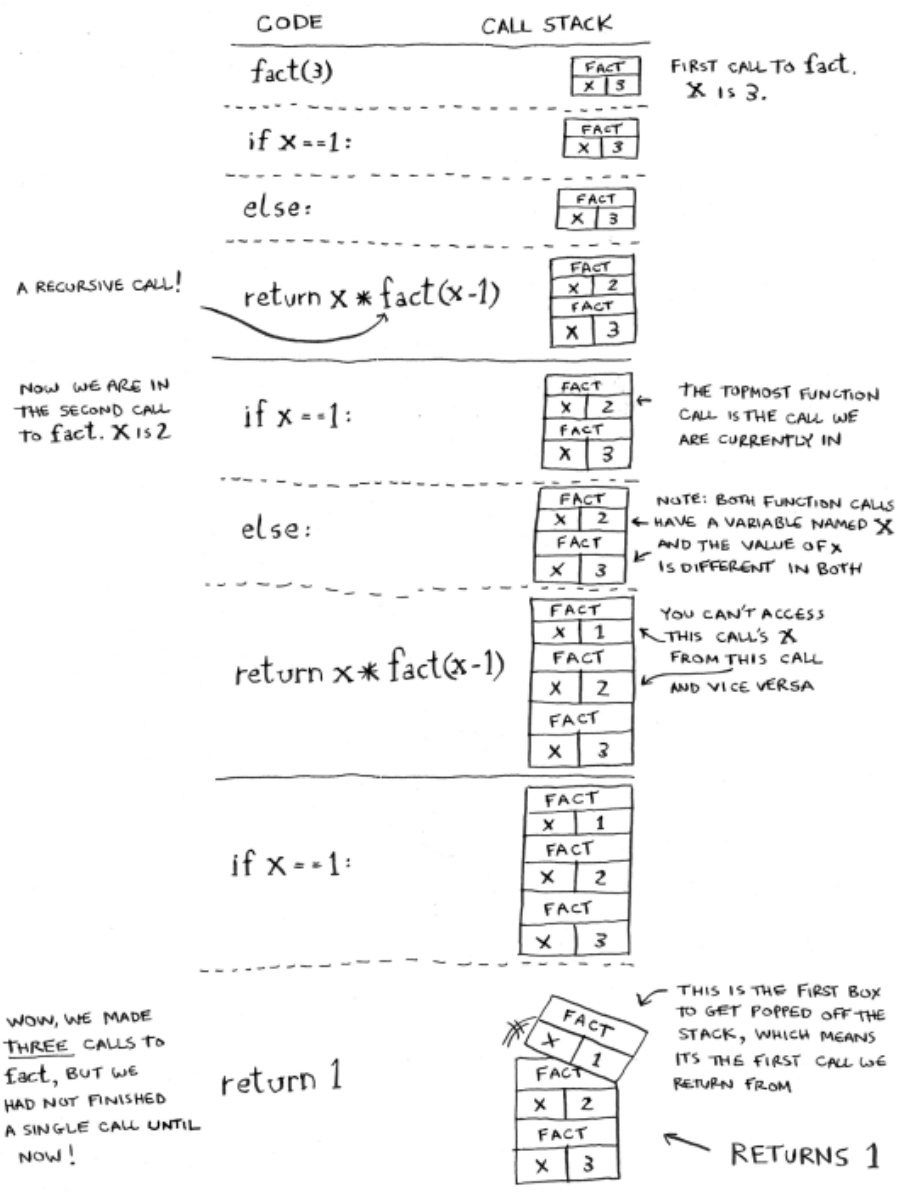
Now, every time you make a function call, your computer saves the values for all the variables in that "box".

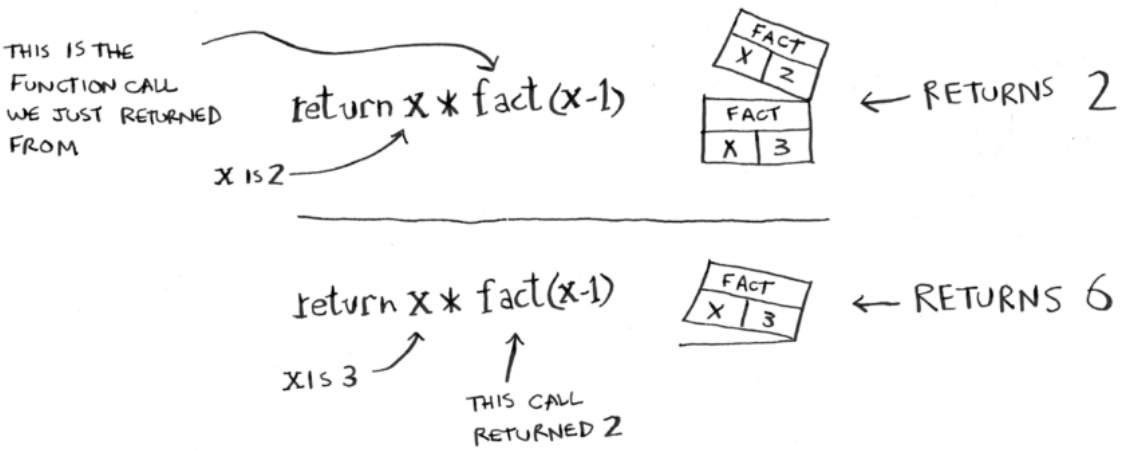
> "W*hen you call a function from another function, the calling function is paused in a partially completed state."*

The values of that function are stored in memory but they're instead **put on pause.**

Recursive functions can also use this call stack as well. A perfect example of this is in **factorials.**

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


CODE
CALL STACK
fact(3)
FACT
X 3
FIRST CALL TO fact. X IS 3.
if X==1:
else:
A RECURSIVE CALL!
return X * fact(X-1)
FACT
X 2
NOW WE ARE IN THE SECOND CALL TO fact. X IS 2
THE TOPMOST FUNCTION CALL IS THE CALL WE ARE CURRENTLY IN
NOTE: BOTH FUNCTION CALLS HAVE A VARIABLE NAMED X AND THE VALUE OF X IS DIFFERENT IN BOTH
FACT
X 1
YOU CAN'T ACCESS THIS CALL'S X FROM THIS CALL AND VICE VERSA
WOW, WE MADE THREE CALLS TO fact, BUT WE HAD NOT FINISHED A SINGLE CALL UNTIL NOW!
return 1
THIS IS THE FIRST BOX TO GET POPPED OFF THE STACK, WHICH MEANS IT'S THE FIRST CALL WE RETURN FROM
RETURNS 1


THIS IS THE FUNCTION CALL WE JUST RETURNED FROM
return X * fact(X-1)
X IS 2
FACT
X 2
FACT
X 3
RETURNS 2
return X * fact(X-1)
X IS 3
THIS CALL RETURNED 2
FACT
X 3
RETURNS 6

What's the main problem with saving everything on the call stack? **It takes up A LOT of memory.** When your stick is too tall, that means your computer is saving information for many function cells.

2 things you can do:

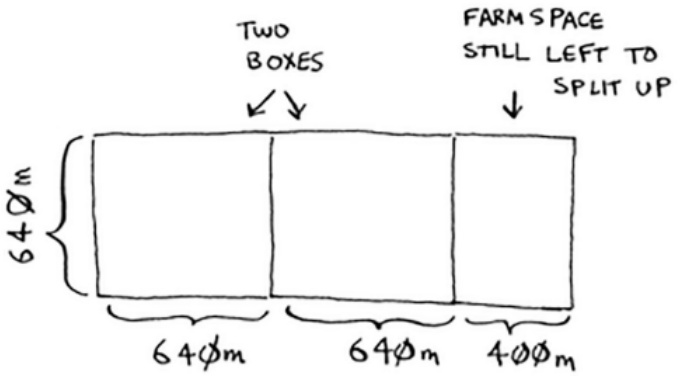
1. Rewrite your code to use a loop isntead
2. Use something known as **tail recursion. (only supports some languages)**

▼ Divide and Conquer

Divide and Conquer is generally used in situations where you need to solve a problem that **can't be solved with the existing algorithms you've learned.**

It can also be known as a **recursive technique** used to solve these problems. Think of D&C like this whole new framework used for solving problems.

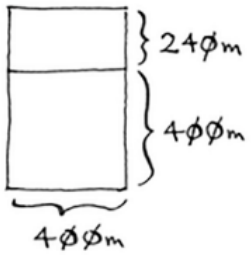
Let's say that we have this farm:

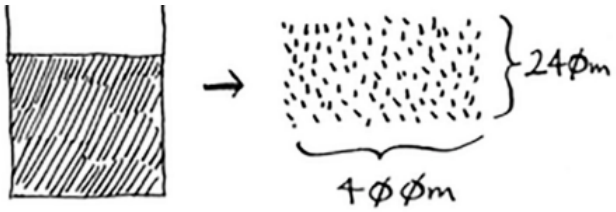



You cut the 1680x640 farm into 2 squares, each of length 640. **Now your problem is that you need to split up the smaller segment (640x400).**

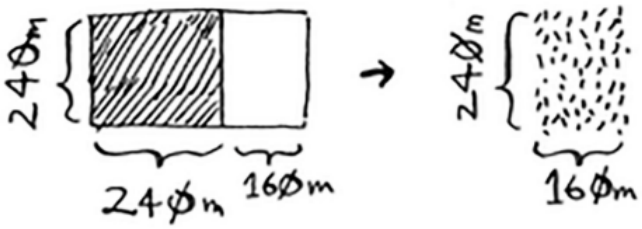
If you find the biggest box for that size (640x400), **then that'll be the biggest box which will work for the entire farm!** Notice how you reduced the problem from a 1680x640 farm to a 640x400 farm.

And now, you simply apply the algorithm again and again until you find the perfect case.

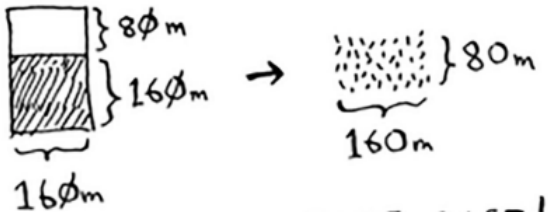





And you can draw a box on that to get an even smaller segment, 240 × 160 m.




And then you draw a box on that to get an even *smaller* segment.



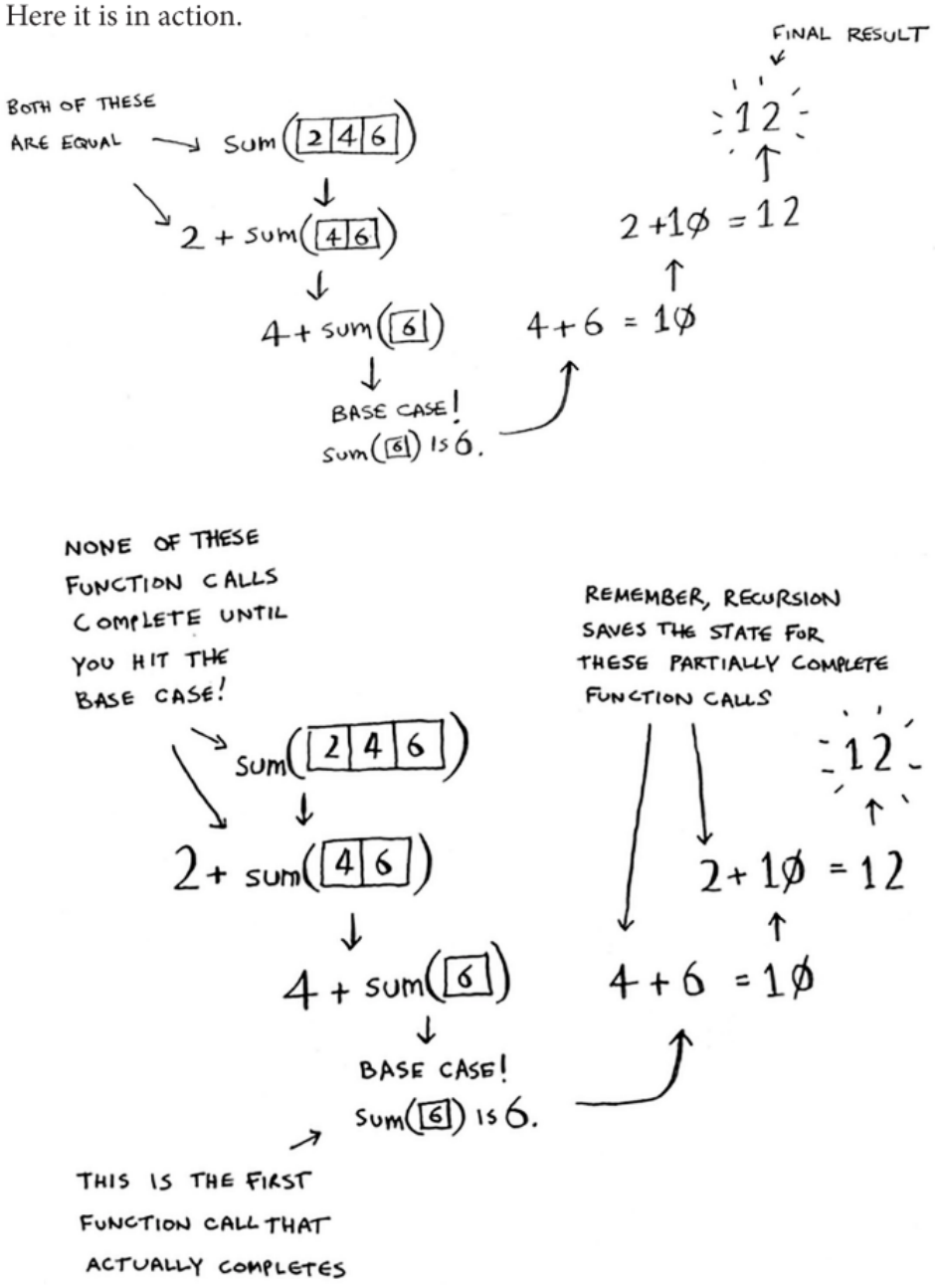

So for recap, D&C can be broken down into 2 main steps:

1. Figure out a simple case as the base case
2. Figure out how to reduce your problem and get to the base case.

**Remember that D&C isn't a simple algorithm that you can apply to a problem. Instead, it's a framework that you can use to think about a problem.**

Here's array summing in recursion:

Here it is in action.




This is known as **functional programming** (Haskell).

▼ Quicksort

Quicksort is generally known as a **sorting algorithm.** It's a lot **faster than selection sort** and is generally used. It uses a recursive algorithm structure for fast + proper sorting.

Here's what the process looks like:

1. Pick an element from the array. **This is generally known as a pivot.**
2. From there, you **bin the smaller + larger elements into 2 separate arrays.**
    a. This overall process is known as **partitioning**

3. You then run this **recursively until you get sorted arrays** and then you can simply **concatenate everything** and you now have your output.

```
def quicksort(array):
  if len(array) < 2:
    return array
  else:
    pivot = array[0]
    less = [i for i in array[1:] if i <= pivot]
    greater = [i for i in array[1:] if i > pivot]
    print(pivot)
    print(less)
    print(greater)
    print("-"*20)
  return quicksort(less) + [pivot] + quicksort(greater)
print(quicksort([10, 5, 2, 3]))
```

The average runtime generally for quicksort is **O(n log n)** but the worst case is generally **O(n^2) time.**

Generally remember that when we call something like **O(n)**, it's generally more of like **O(c*n)** where *c* is a constant multiplied by n but we usually don't look at that for large numbers.

But sometimes, **the constant can make the difference.** Quicksort generally has a smaller constant than merge sort. So if both of them are in O(n log n) time, then **quicksort is faster.**

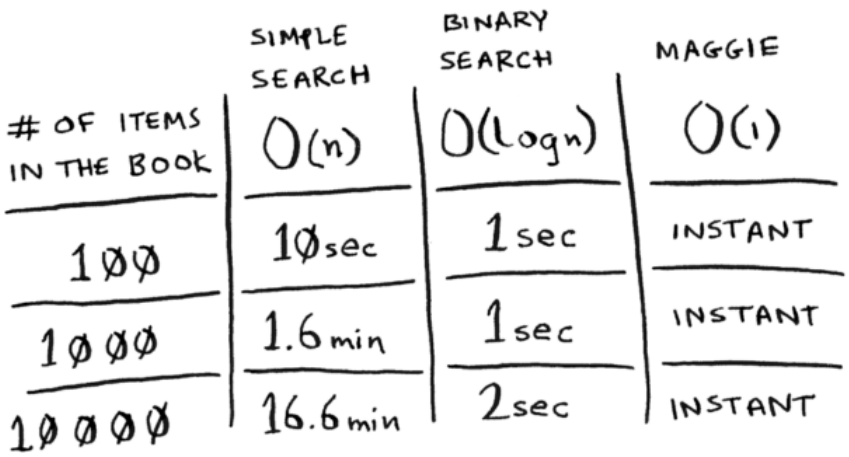
Quicksort is also faster in practice mainly because it actually meets the average case substantially more than the worst case.

**The pivot matters.** You want to have both arrays be as small as they can. **That's why you'd always choose the middle [or a random] number.**

> "In this example, there are O(log n) levels (the technical way to say that is, “The height of the call stack is O(log n)”). And each level takes O(n) time. *The entire algorithm will take O(n) * O(log n) = O(n log n) time. This is the best-case scenario."*

▼ Hash tables

The goal with hash tables mainly is to be able to **retrieve information about an item instantaneously.** Something that's running time is **O(1)**.




Hash functions are essentially when you **input a string** and are returned **a number.** Think of it as a way to map from strings to numbers.

Why does this work?

1. Hash functions consistently **map a name to the same index (value)**
2. Hash functions **map different strings to different indexes.**
3. Hash functions **know how big your array already is** and can only return valid indexes.

Everything, as described above, is known as a **hash table.**

Other common names include: **hash maps, maps, dictionaries, and associative arrays.**

Common use cases generally include: **lookups, translation, correlation, preventing duplicate entries, and in cache.**

Sometimes, what happens is that whenever we bin values together (alphabet words), we end up with 2 or more words that fall under the same index. **This is known as collision.** (2 keys have been assigned the same slot)

The simplest solution used right now is to use **linked lists**. But there's a problem here; if you're data is not evenly distributed i.e. only one node has the majority of lists, **you're going to slow down your table.**

What do you do? **Create a proper hash function that maps the keys evenly throughout the hash.**

Hash tables run in **O(1) time aka constant time** (which means instantaneously)**.** Your worst case is O(n) time.

| | HASH TABLES (AVERAGE) | HASH TABLES (WORST) | ARRAYS | LINKED LISTS |
|---|---|---|---|---|
| SEARCH | O(1) | O(n) | O(1) | O(n) |
| INSERT | O(1) | O(n) | O(n) | O(1) |
| DELETE | O(1) | O(n) | O(n) | O(1) |

It's important you don't hit worst-case scenarios when working with hash tables. To do that, you need to **avoid collisions.** To do that, you need: **a low load factor and a good hash function.**

The load factor is essentially the **Number of items in hash table / total number of slots.**

Ex. if I had a hash table with length 5 and I've filled in 2 slots, then my load factor is: **2/5 = 0.4**

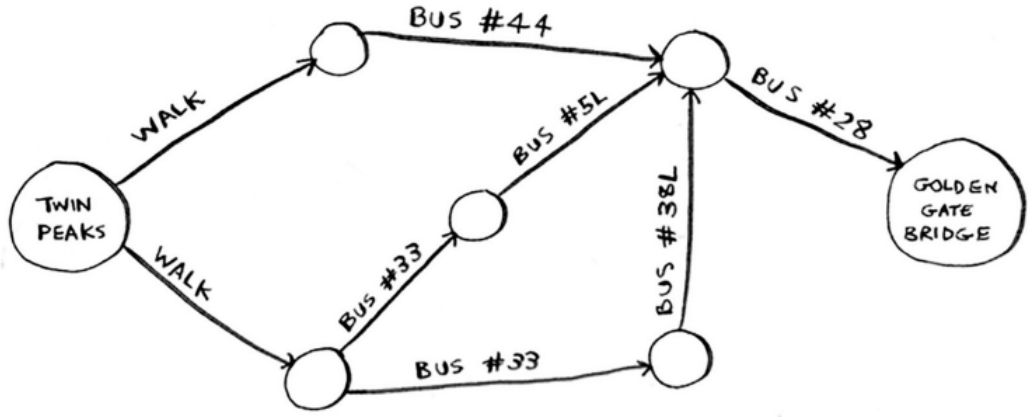
When your load factor starts to grow, you need to add more slots to your hash table. This is generally known as **resizing.**

**Quick note:** lower load factors lead to fewer collisions + results in increased performance. A rule of thumb mentioned in the book is to resize whenever the **load factor ≥ 0.7.**

Yes, averaged out, hash tables will still take only O(1) for resizing

▼ Graphs + Breadth-first search

Breadth-first search allows you to find **the shortest distance between 2 values.**




**Note:** the problem indicated above is known as a **shortest-path problem** (how can I get from x to y in the fastest way?)**.** The algorithm that we use to solve this? This is **breadth-first search.**

To figure out how we solve the above diagram in the fastest way, there are 2 main steps:

1. Model the problem as a graph
2. Solve the problem using breadth-first search

What even is a graph? A graph is a way to **set connections.** They're made up of **nodes** and **edges**. A node can be **directly connected** to other nodes. These nodes that are connected are known as **neighbors.**

The 2 questions breadth-first answers?

1. **Is there a path** from node A to node B?
2. **What is the shortest path** from node A to node B?

▼ Linear programming

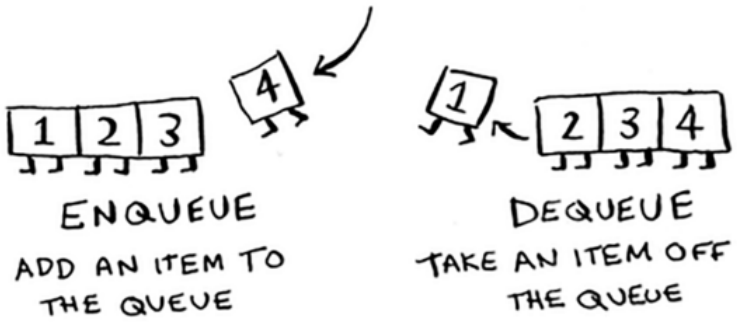
Linear programming can be used to maximize something given the time constraints.

You're goal could be: ***given 2 meters of fabric + cotton, what's the maximum amount of shirts and pants that I can make to maximize profit?***

These aren't direct answers but rather a **framework for thinking.**

This is quite similar to graph algorithms (actually linear programming is the greater subset for graphs)

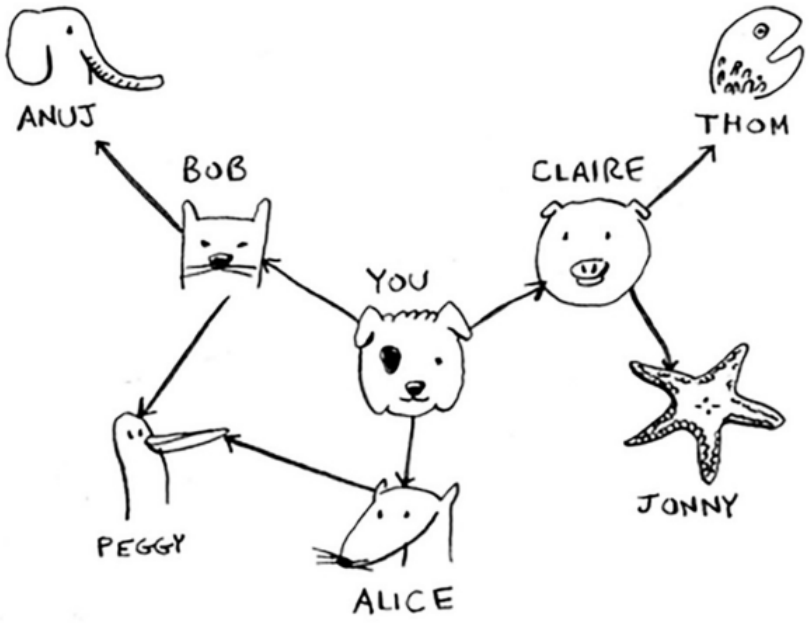
The way we'd organize through different hierarchies (levels) of nodes are through **queues.** They're similar to stacks but you can only perform 2 operations: **enqueue** and **dequeue.**




It works in **FIFO structure;** first in, first out. Notice how as the 4 is added, the 1 is automatically removed.

The main concern people face is, *How do you express relationships (nodes)?* We can do this through **hash tables.**

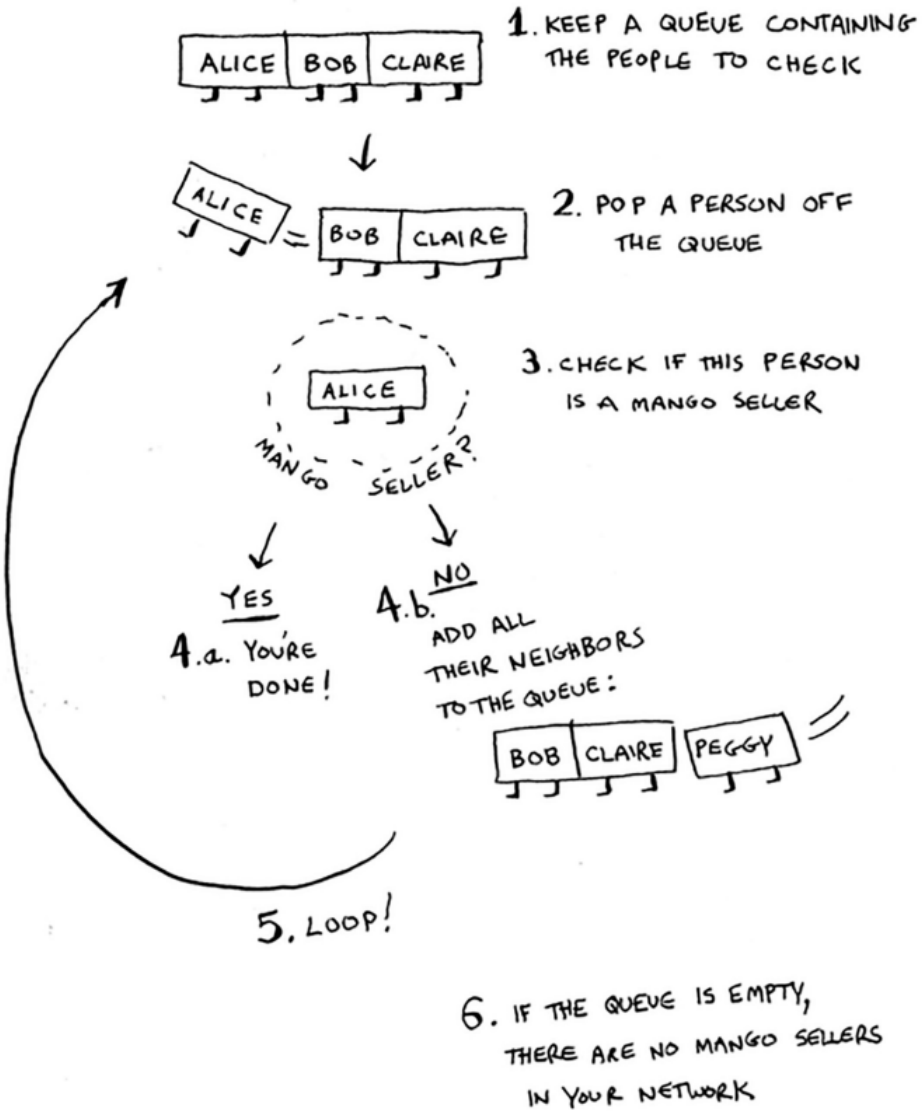
For example, with the diagram below,




```
graph = {}
graph[“you”] = [“alice”, “bob”, “claire”]
graph[“bob”] = [“anuj”, “peggy”]
graph[“alice”] = [“peggy”]
graph[“claire”] = [“thom”, “jonny”]
graph[“anuj”] = []
graph[“peggy”] = []
graph[“thom”] = []
graph[“jonny”] = []
```

We also have something known as **directed graphs.** This is essentially when one of the nodes don't connect to anything else (Anuj, Peggy, Jonny, Thom).

So to recap, here's what the overall implementation looks like:

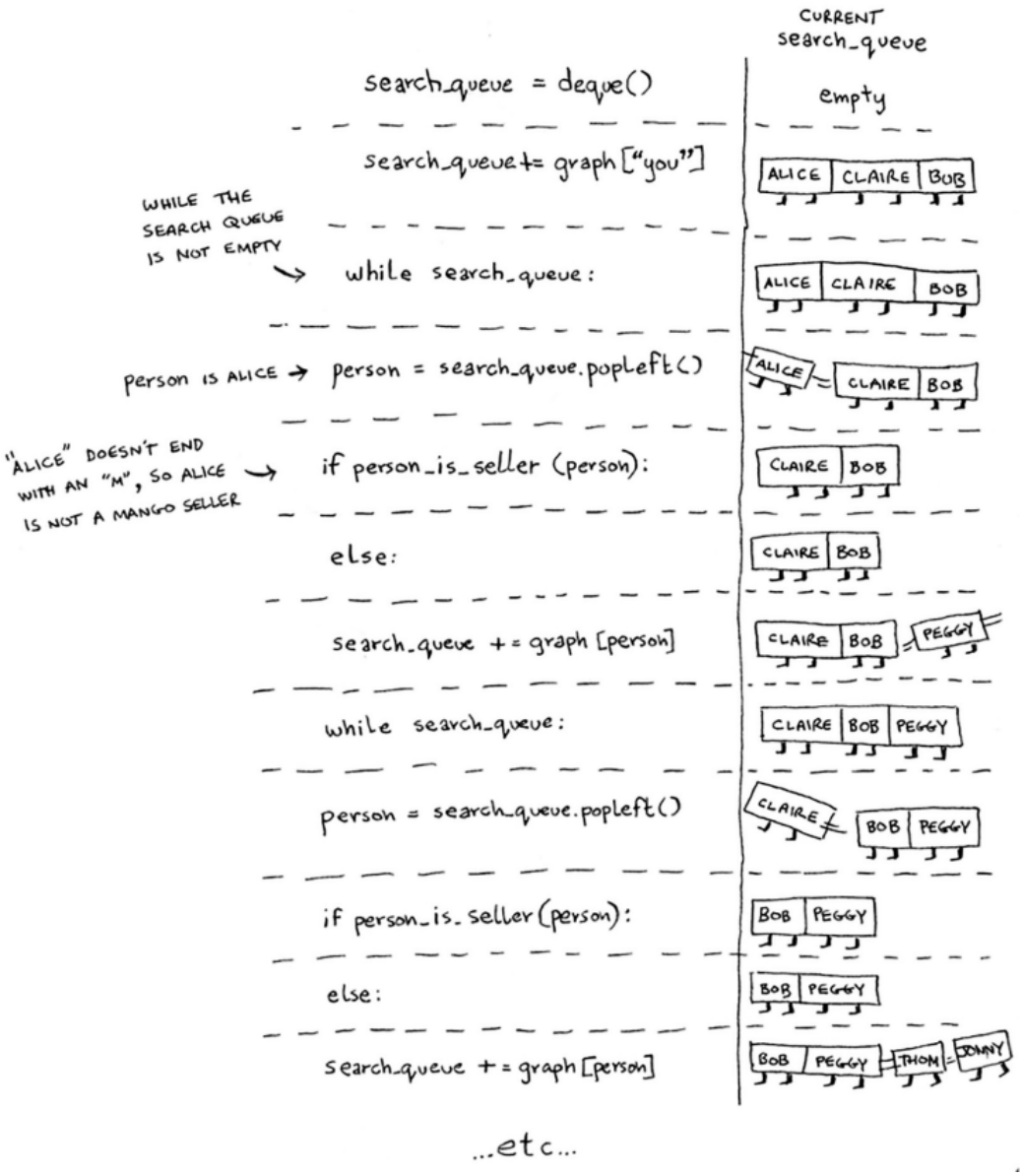



```
# A use case of this for "mango" sellers

from collections import deque
search_queue = deque()
search_queue += graph[“you”]

def person_is_seller(name):
  return name[-1] == ‘m’

while search_queue:
  person = search_queue.popleft()
  if person_is_seller(person):
    print person + “ is a mango seller!”
    return True
  else:
    search_queue += graph[person]
return False
```




```
def search(name):
  search_queue = deque()
  search_queue += graph[name]
  searched = []
  while search_queue:
    person = search_queue.popleft()
    if not person in searched:
      if person_is_seller(person):
        print person + " is a mango seller!"
        return True
    else:
      search_queue += graph[person]
      searched.append(person)
  return False
```

Note that the running time for this algorithm will be **atleast O(n)** because we search through and follow each edge.

> "*Breadth-first search takes O(number of people + number of edges), and it's more commonly written as O(V+E) (V for number of vertices, E for number of edges)."*

**Trees** are essentially special graphs where you don't have an edge pointing up (like a family tree!)

▼ Dijkstra's Algorithm

4 main steps:

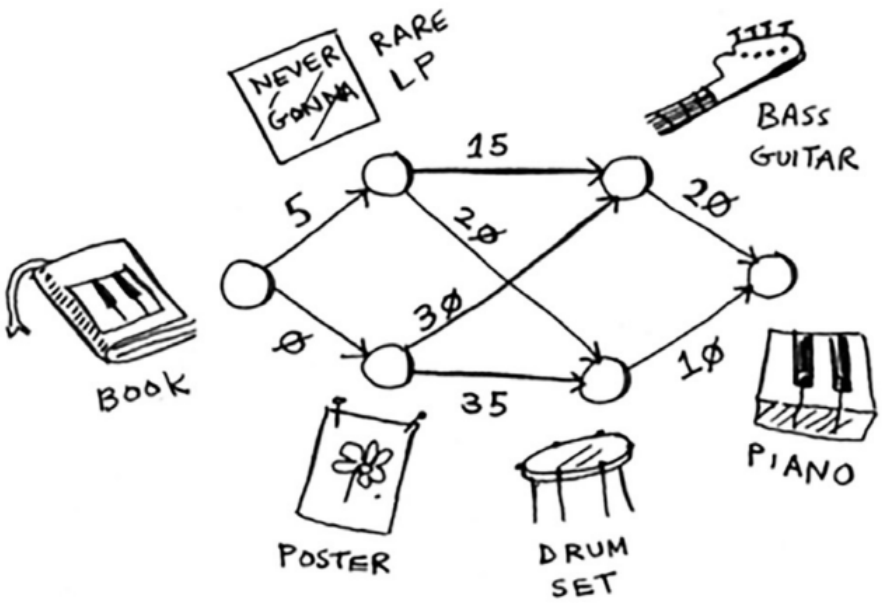
1. Find the "cheapest" node. This is generally the node that you can get to with the shortest cost
2. Update the costs of the neighbours of the node.
3. Repeat this until you've done this for every node in the graph.
4. Calculate the final path

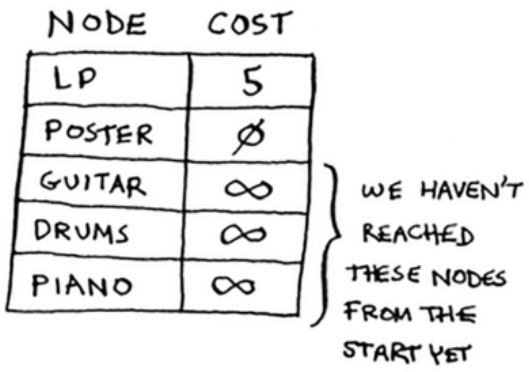
> *"In the last chapter, you used breadth-first search to find the shortest path between two points. Back then, "shortest path" meant the path with the fewest segments. But in Dijkstra's algorithm, you assign a number or weight to each segment. Then Dijkstra's algorithm finds the path with the smallest total weight."*

Each edge in the graph has a number associated it, **known as weights (**graphs with weights are known as weighted graphs, unweighted for no weights).

Whenever dealing with unweighted graphs, **breadth-first** works. For weighted ones, **Dijkstra's algorithm.**

Here's an example of trading a **piano book → actual piano (screenshots):**

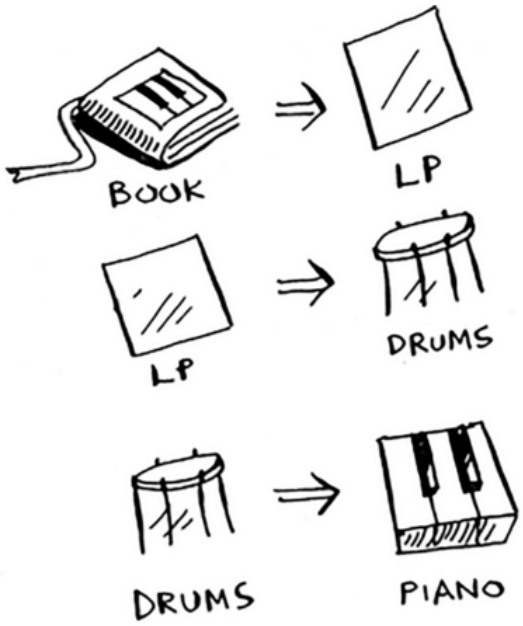





Before you start, you need some setup. Make a table of the cost for each node. The cost of a node is how expensive it is to get to.

We work in **hierarchies.** That means the first section **(parent column)** would be the the book as **that's the parent for the poster + rare LP.**

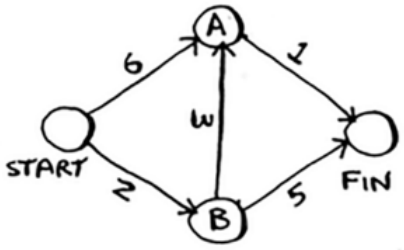
What are the trades he needs to make?



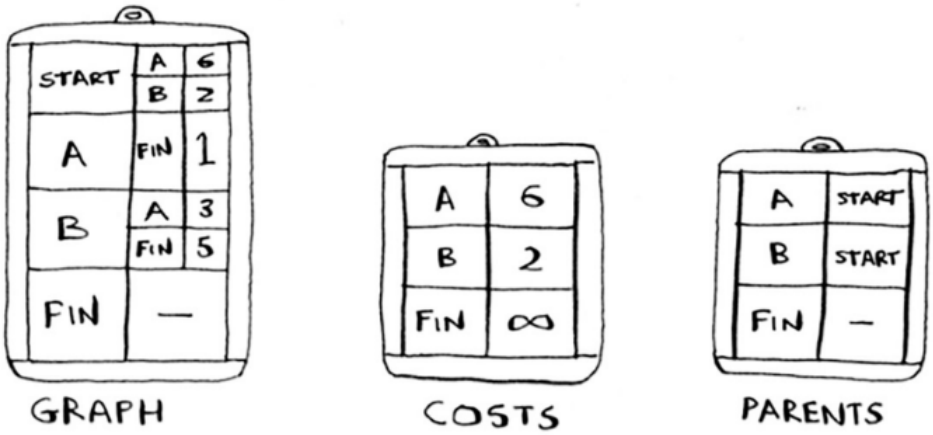

We can also use **negative signs** when we want to show situations **where we get something back** (ex. I get paid back $7 for getting some item).

The only concern? **Dijkstra's algorithm doesn't support negative weights.**

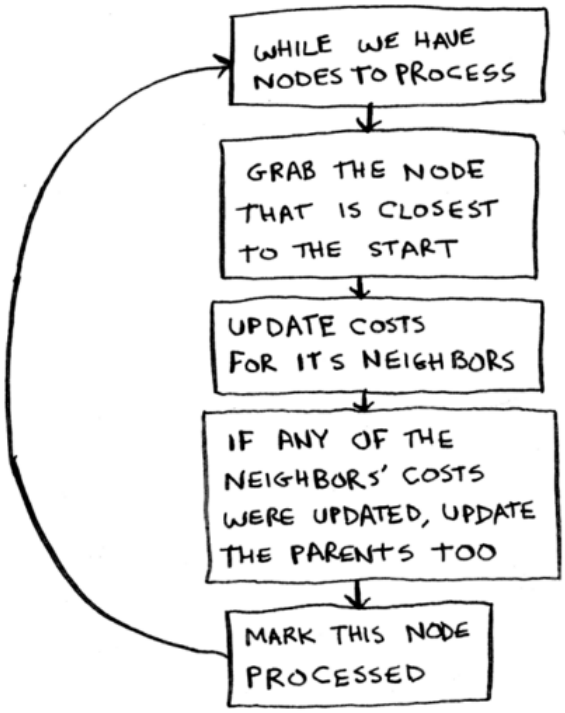
What do we do? **Use the Bellman-Ford algorithm!**




To code this example, you'll need three hash tables.




What the algorithm looks like?



```
def find_lowest_cost_node(costs):
  lowest_cost = float(“inf”)
  lowest_cost_node = None
  for node in costs:
    cost = costs[node]
    if cost < lowest_cost and node not in processed:
      lowest_cost = cost
      lowest_cost_node = node
  return lowest_cost_node

node = find_lowest_cost_node(costs)
while node is not None:
  cost = costs[node]
  neighbors = graph[node]
  for n in neighbors.keys():
    new_cost = cost + neighbors[n]
    if costs[n] > new_cost: #we want to reduce + lower the cost
      costs[n] = new_cost
      parents[n] = node
  processed.append(node)
node = find_lowest_cost_node(costs)
```

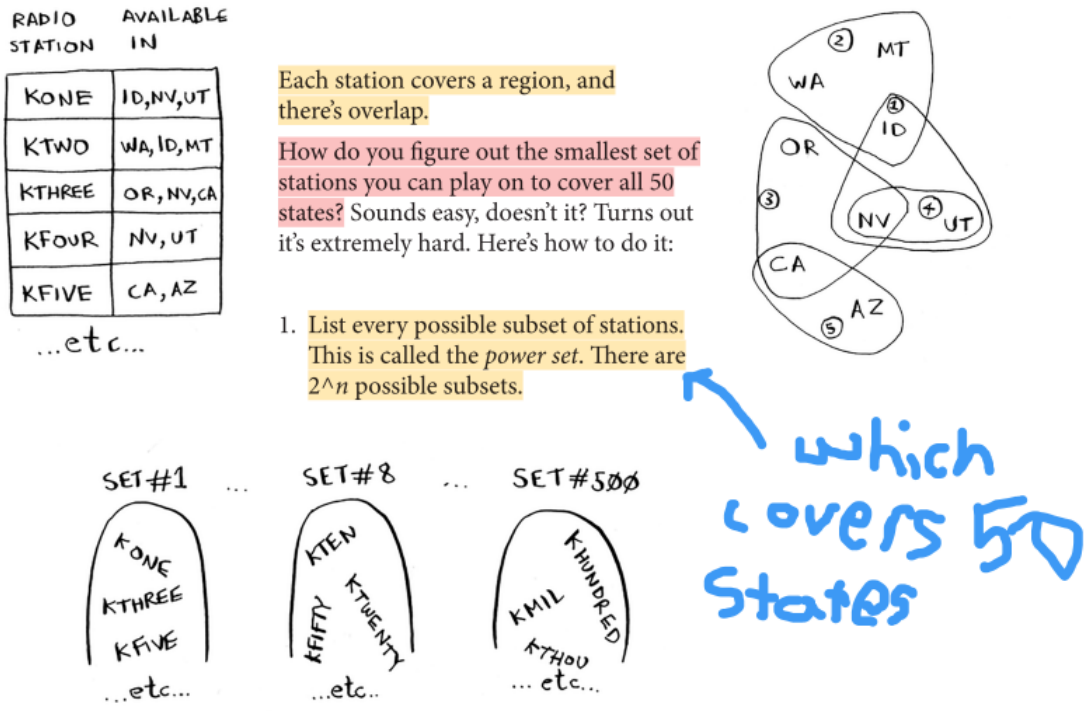
▼ Greedy algorithms

It's pretty straightforward: **at each step you pick the locally optimal solution, and in the end you’re left with the globally optimal solution.**

They're also super simple to write and usually hit the solution.

Let's use the set-covering problem (from textbook):

> "Suppose you’re starting a radio show. You want to reach listeners in all 50 states. You have to decide what stations to play on to reach all those listeners. It costs money to be on each station, so you’re trying to minimize the number of stations you play on. You have a list of stations."



What's the problem? **It takes an extremely long time to calculate every single possible subset.** The running time is **O(2^n)**, it's WAY TOO SLOW. That would be at least **15 years to figure this out** (assuming that you calculate 10 subsets/second)**!**

How can we solve this with greedy algorithms?

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

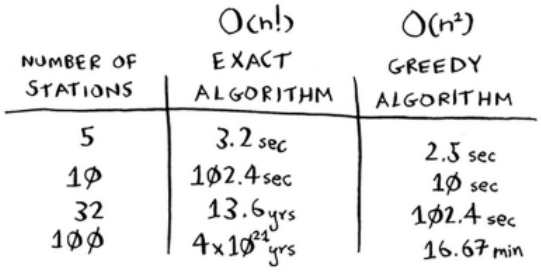
What we're using is something known as an **approximation algorithm.** Calculating the exact solution takes way too long.

How do we judge the algorithm?

1. How fast they are
2. How close they are to the optimal solution

What makes greedy a good choice? It has an **O(n^2) running time** while being able to have high running speeds.

```
states_needed = set(["mt", "wa", "or", "id", "nv", "ut", "ca", "az"])
arr = [1, 2, 2, 3, 3, 3]
stations = {}
stations["kone"] = set(["id", "nv", "ut"])
stations["ktwo"] = set(["wa", "id", "mt"])
stations["kthree"] = set(["or", "nv", "ca"])
stations["kfour"] = set(["nv", "ut"])
stations["kfive"] = set(["ca", "az"])
final_stations = set()
while states_needed:
  best_station = None
  states_covered = set()
  for station, states_for_station in stations.items():
    covered = states_needed & states_for_station # this is a set intersection
    if len(covered) > len(states_covered): #check whether this station covers more states then the current station, best_station
      best_station = station #update if condition is met
      states_covered = covered
states_needed -= states_covered #update because the station has covered those states
final_stations.add(best_station)
print(final_stations)
```




In this example, the travelling salesman is an **NP-complete problem**; a computational problems for which **no efficient solution algorithm has been found.**

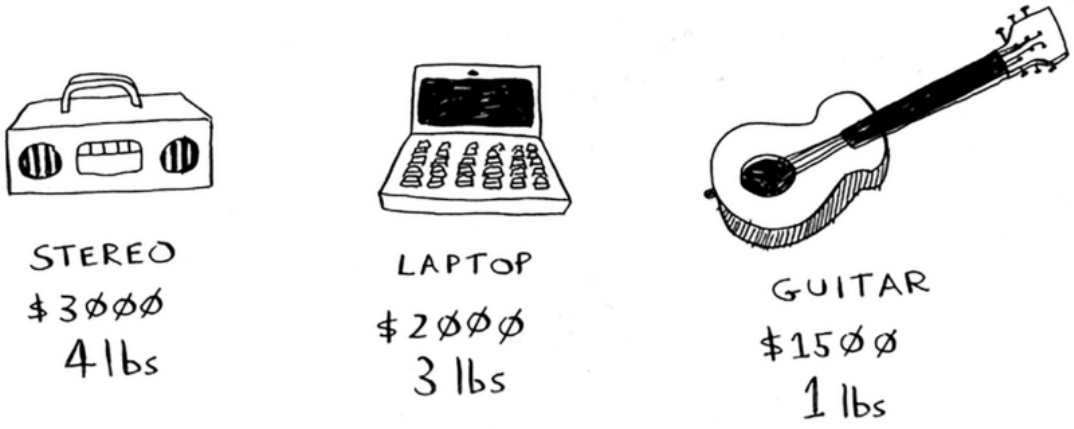
▼ Dynamic programming

We talked about in the previous chapter (greedy algorithms) that sometimes, we can only come up with an **approximate solution.** But is it even possible to find + calculate the optimal solution? That's where dynamic programming comes in.

Dynamic programming is essentially this method to be able to **solve subproblems that build up to solving the big problem.**

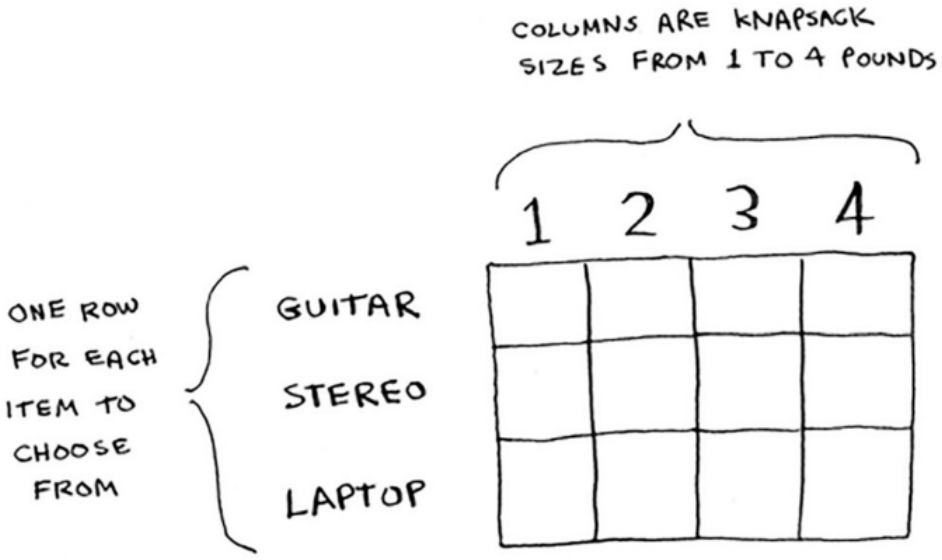
Let's look at the knapsack problem:

You have three items that you can put into the knapsack.



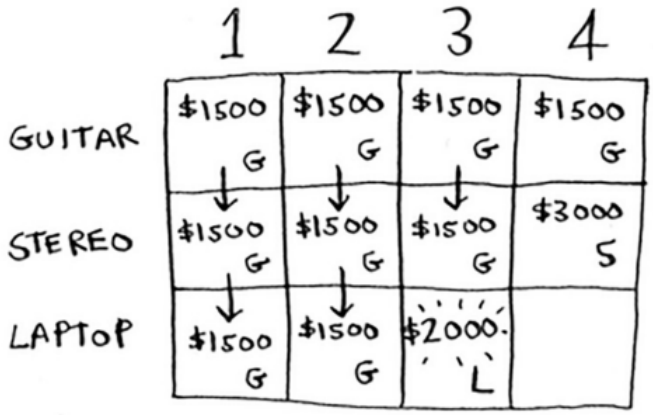

What items should you steal so that you steal the maximum money's worth of goods?

**Every dynamic problem starts with some sort of grid**, here's a grid for the knapsack problem:



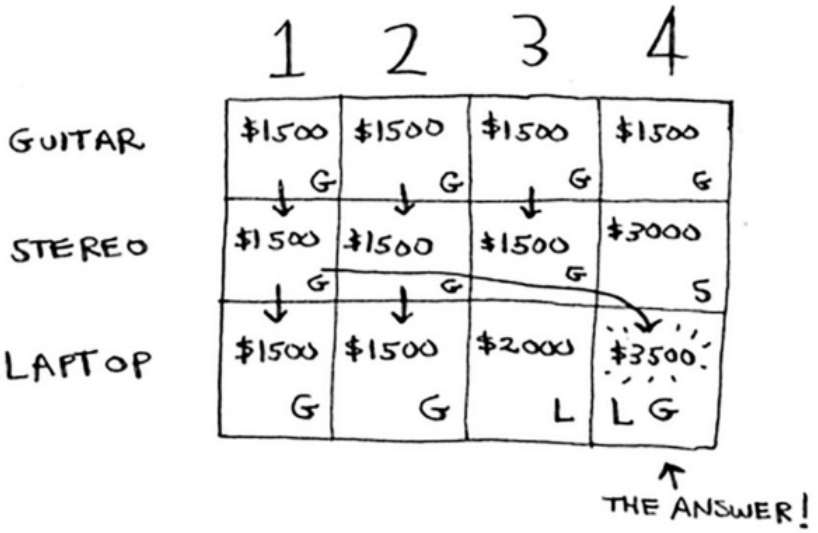

The question that the grid is trying to answer is **if your knapsack had capacity *x* lb, what is the maximum value that you can put in that bag?**

As you fill through the table, this is what you end up becoming at:

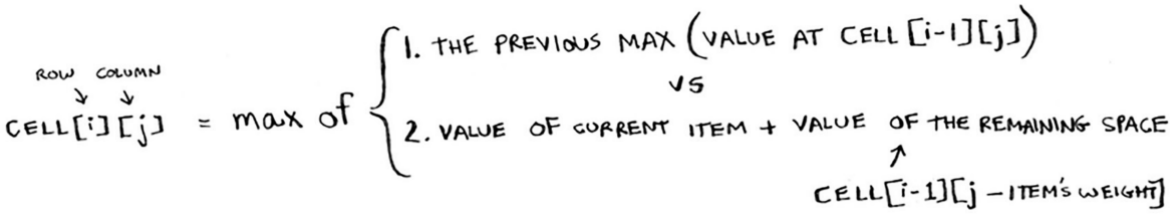




According to your last best estimate, the best item you can put in there for that 1lb space is the guitar. This is what your now final grid looks like!




What does that look like as a formula?




The biggest question that pops up in your mind is **can the value of a column ever go down?**

**NO.** At every iteration, you're storing the current maximum estimate. That estimate that you get will never be worse than what it was before. Same thing with the order of the table, **it makes no impact.**

What about dealing with fractions of items? **That wouldn't work, dynamic programming allows you to only take the item or not.** Instead, you can solve this with a **greedy algorithm!** Take as much as you can of one item, when that runs out, take as much as you can of the next most valuable item, and so on.

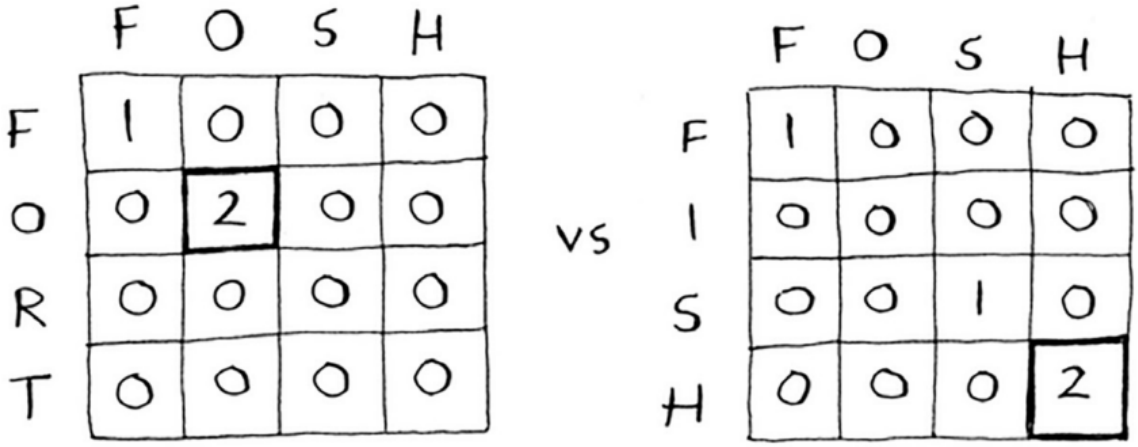
> "*Dynamic programming is useful when you're trying to optimize something given a constraint. In the knapsack problem, you had to maximize the value of the goods you stole, constrained by the size of the knapsack."*

General tips for dynamic solutions:

1. Every dynamic-programming solution involves a grid.
2. The values in the cells are usually what you're trying to optimize. For the knapsack problem, the values were the value of the goods.
3. Each cell is a subproblem, so think about how you can divide your problem into subproblems. That will help you figure out what the axes are.

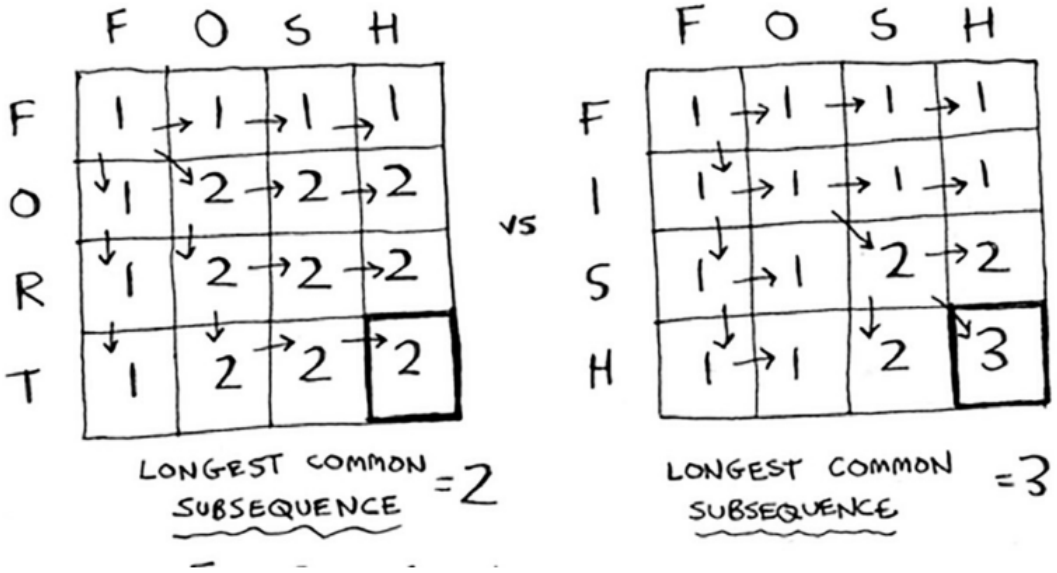
Let's look at the longest common subsequence: **Suppose Alex accidentally searched for fosh. Which word did he mean: fish or fort?**

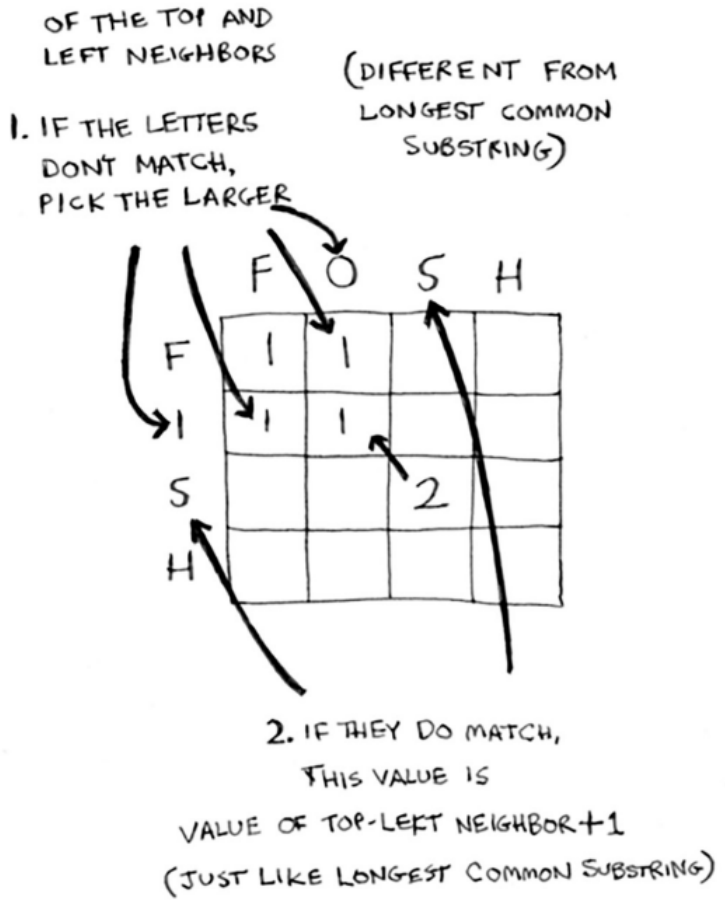
Let's compare them using the longest-common-substring formula.




That isn't gonna work. Let's redo the algorithm:

Here's the final grid.






Notice how it works in a diagram "way". When a new match is found, that row + column is filled with that max value until another match is found.

What are some applications of dynamic programming (taken from book):

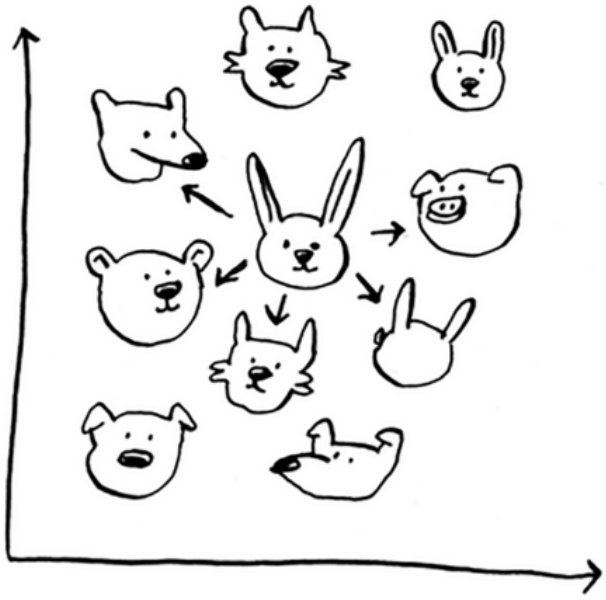
1. Biologists use the longest common subsequence to find similarities in DNA strands. They can use this to tell how similar two animals or two diseases are. The longest common subsequence is being used to find a cure for multiple sclerosis.
2. Have you ever used diff (like git diff)? Diff tells you the differences between two files, and it uses dynamic programming to do so.
3. We talked about string similarity. Levenshtein distance measures how similar two strings are, and it uses dynamic programming. Levenshtein distance is used for everything from spell-check to figuring out whether a user is uploading copyrighted data.

▼ k-nearest neighbors

Generally, the main intuition behind KNNs are finding the **closest neighbor** of a point when graphed out.

On a high level understanding, Netflix's algorithm also works like this!

Generally, users are plotted out based on similarity so whenever you get recommended movies, Netflix **looks at the people who share similar taste and choose a movie that'll appeal to you.**



The way you figure out the similarity will be through the **features you choose to filter by.** The way we find the distances between two [or more] points can be through the **Euclidian distance** (aka Pythagorean formula):

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

With KNNs, there are 2 main things you can do: **classification, and regression.** In this case, above we did classification i.e. classifying genres of similar movies, but regression is super useful as well.

The main question right now is **"how do I chose the right features to compare against?"** 2 things:

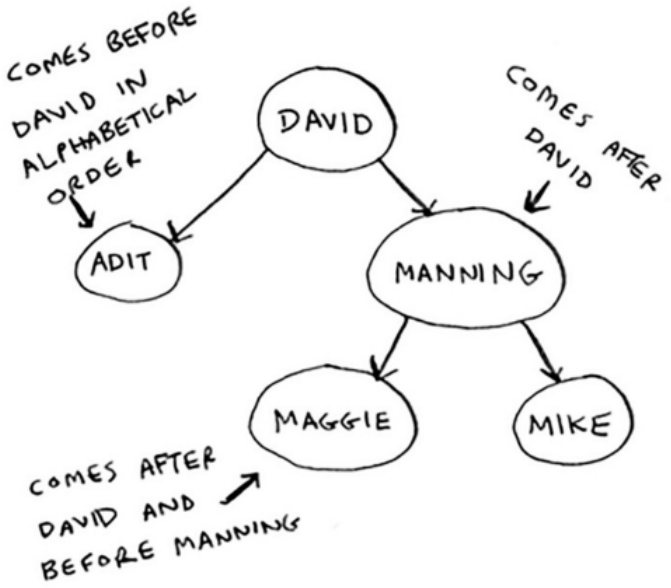
1. Features that directly correlate to the movies you're trying to recommend
2. Features that don't have a bias (for example, if you ask the users to only rate comedy movies, that doesn't tell you whether they like action movies)

Additionally, this can also fall into Machine Learning algorithms. By selecting accurate + extracting relevant features, you can determine its nearest neighbors to make an accurate prediction.

▼ Trees

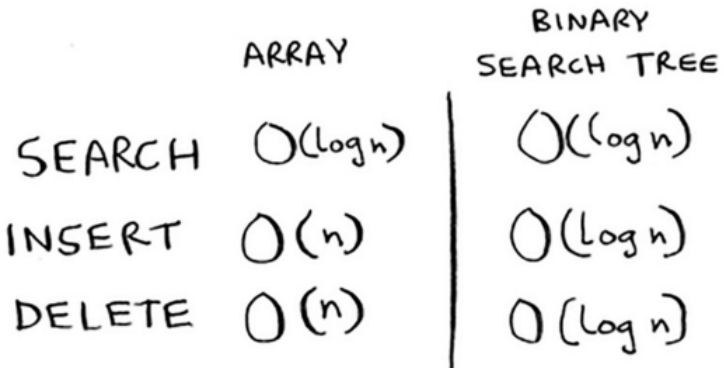
The main intuition behind binary search trees are mainly that **whenever you do binary search, you have to add and then sort the array. That's inefficient.**

For every node, the nodes on the left of it have a lower value while the ones on the right are a lot larger in value.

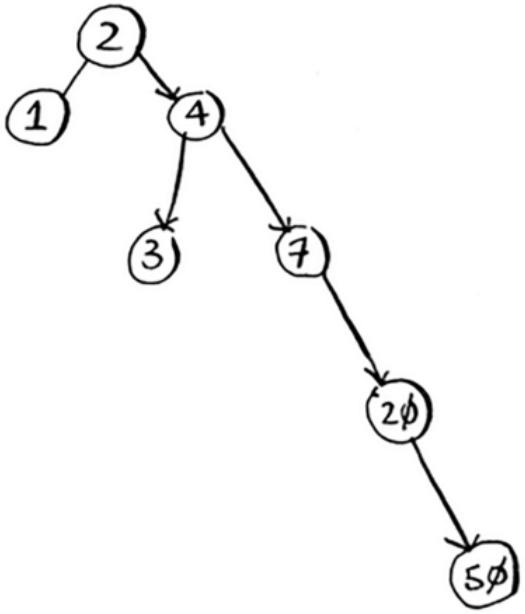



So, if I wanted to search for Maggie, I'd follow: **David → Manning → *Maggie*!**

> *"Searching for an element in a binary search tree takes O(log n) time on average and O(n) time in the worst case. Searching a sorted array takes O(log n) time in the worst case, so you might think a sorted array is better. But a binary search tree is a lot faster for insertions and deletions on average."*



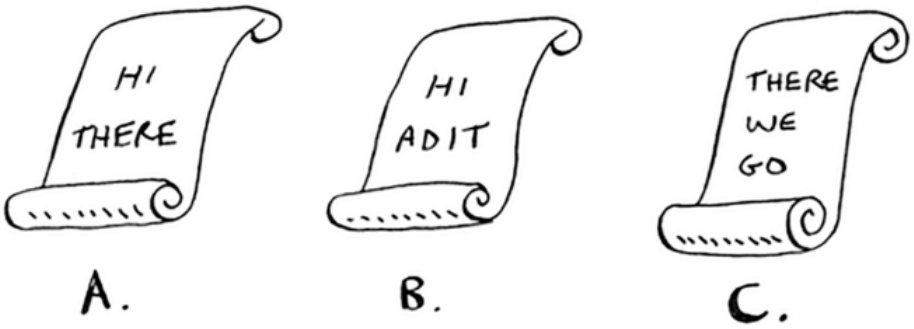

Some of the main downsides of binary trees are that you don't get random access (indexing). The performance of a tree rely fully on **how balanced the tree is.** Look at the tree below:



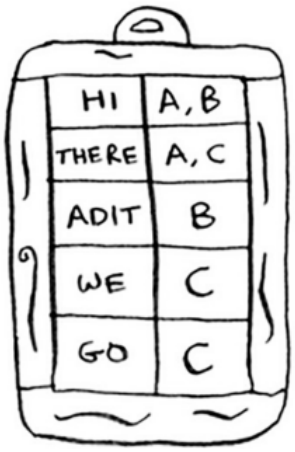

Some trees are also able to balance themselves, like the red-black tree. B-trees are used in storing data in databases.

▼ Inverted indexes

Let's say that we have 3 web pages with this content:




From there, let's build a hash table from that:




Now let's say that the user *searched for **there.*** Now we easily know that we can **display pages A and C.** We're essentially creating a **hash that can map words to places where they appear (web pages).** This is exactly what inverted indexes are.

▼ Fourier transform

Fourier transform is essentially this algorithm that can basically, *given a smoothie, the Fourier transform will tell you the ingredients in that smoothie. Or given a song, the transform can separate it into individual frequencies.*

What's the use case? **With songs, you can boost the individual frequencies you care about (boost bass, hide the treble).**

Can also be used in compressing music. Break down the audio file into its ingredient notes. From there, the transform will tell you exactly how much each note contributes to the song. Then you delete the irrelevant notes. That's exactly how mp3 works!

▼ Parallel algorithms

The main focus with parallel algorithms is to **take advantage of the multiple cores in a computer.**

For example, the best time you can get with a sorting algorithm is generally **O(n log n) time.** You can't generally sort an array in **O(n)** time, **unless you use a parallel algorithm!**

Parallel algorithms are generally super hard to design + get it working.

> *"So if you have two cores in your laptop instead of one, that almost never means your algorithm will magically run twice as fast."*

1. Parallelism management - if you have to sort an array of 1,000 items, how do you divide this task among the two cores? Even if you give each core 500 items to sort and then merge them, merging itself takes time.
2. Load distribution amongst processors - if you have 10 tasks to do, you'd probably give each core 5 tasks. What's the problem? The first core is given the easy tasks whereas the other core gets all the hard tasks. **Now a core stays idle while it could be used to work.**

Let's take a look at examples of parallel algorithms:

▼ MapReduce

A special type of parallel algorithm used is MapReduce that's becoming increasing famous, known as **distributed algorithms**. MapReduce is essentially a popular distributed algorithm to manage workloads amongst hundreds of cores.

The main benefit here is that distributed algorithms allow you to do lots of work while the objective is to speed up the time it takes to do it.

MapReduce works on 2 main functions: **map** and **reduce:**

The map function basically **applies a function to each item in an array** while distributing the workload on each machine/core.

The reduce function basically **downsizes a list of items into a singular output.** An example might be...

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
15
```

▼ Bloom filters + HyperLogLog

The main problem is that **you want to go through a large list to figure out whether an element is in a large set.**

The fastest way to do this could be with a has; you have 2 columns: {website_name: checked_or_not}. So, if I wanted to check if I already saw srianumakonda.com, then I can just look it up in the hash.

Remember that the average lookup time is **O(1).**

What's the problem though: **these hash tables are HUGE. It's a space problem.**

Bloom filters are essentially probabilistic data structures. Although false positives might be possible, false negatives are NOT.

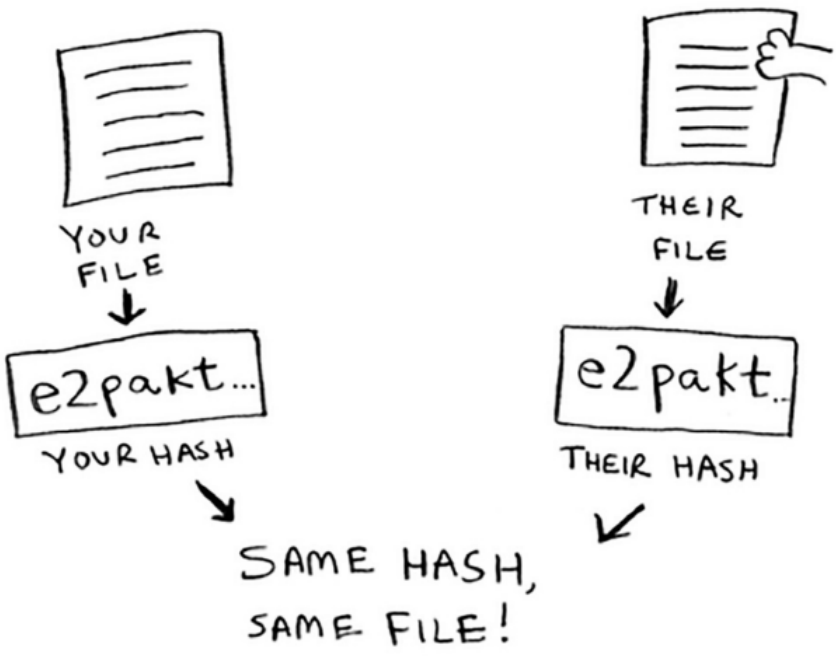
The main benefit here is that Bloom filters take up **very little space to operate it at.** Tables have to store every single URL by Google while it's ok to be wrong in these scenarios.

**HyperLogLog is also quite similar.** You can approximate the number of unique elements in a set (ex. Amazo wants to count the # of unique elements that users looked at today) **with a fraction of the memory.**

▼ SHA algorithms

SHA (**Secure hash algorithms**) convert an input to a hash for that string. *Note: hash's are just a short string in this context where the goal is string→string.*

SHAs can be used in situations to check the similarity of files, especially in the case of extremely large files.




Remember that this is only **one-way. If someone gets the SHA hash, they can't convert that back into the original string.**

> *"SHA is actually a family of algorithms: SHA-0, SHA-1, SHA-2, and SHA-3. As of this writing, SHA-0 and SHA-1 have some weaknesses. If you're using an SHA algorithm for password hashing, use SHA-2 or SHA-3. The gold standard for password-hashing functions is currently bcrypt (though nothing is foolproof)."*

The other thing to note that is **even a single change of the character in a sting results in completely different outputs!** In the case that you a locally-sensitive hash function, you can use something like **Simhash.**

▼ Diffie-Hellman key exchange

The DH key exchange solves a problem that's lurking for quite some time now: **How do you encrypt a message so it can only be read by the person you sent the message to?**

The easiest way to solve this would be to come up with a cipher i.e. a=1, b=2, etc. The main problem here is that **both parties need to agree on the cipher.**

The benefits?

1. **Both parties don't need to know the cipher.**
2. The encrypted messages are **extremely hard to decode.**

Diffie-Hellman has 2 keys: **a public key and a private key.** Everyone can have your public key but when you want to send a message, you can encrypt it using the public key. **Only the private key can be used to decrypt it.**