算法图解 - Grokking Algorithms

Binary Search

Binary search is an algorithm; its input is a sorted list of elements. If an element you're looking for is in that list, binary search returns the position where it's located. Otherwise, binary search returns null.

> only works when the list is in sorted order (alphabetically or numerically)

With binary search, you guess the middle number and eliminate half the remaining numbers every time.

In general, for any list of n, binary search will take log2 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 = (low + high)
    guess = list[mid]
    if guess == item:
      return mid
    if guess > item:
      high = mid - 1
    else:
      low = mid + 1
    return None
```

- Algorithm speed isn't measured in seconds.
- Algorithm times are measured in terms of *growth* of an algorithm.
  - how quickly the run time of an algorithm increases as the size of the input increases.

Run time of algorithms is expressed in Big O notation.

Big O notation, log always means log2. → log base 2

Big O established a worst-case run time, but along with it, it is also important to look at the average-case run time.

Here are five Big O run times that you’ll encounter a lot, sorted from fastest to slowest:

- O(log *n*), also known as *log time.* Example: Binary search.
- O(*n*), also known as *linear time*. Example: Simple search.
- O(*n* log *n*). Example: A fast sorting algorithm, like quicksort (coming up in chapter 4).
- O(*n*2). Example: A slow sorting algorithm, like selection sort (coming up in chapter 2).
- O(*n*!). Example: A really slow algorithm, like the traveling salesperson (coming up next!).

constants like 1/2 are ignored in Big O notation

Selection sort

Arrays and linked lists

Arrays

• With an array, all your elements are stored right next to each other.

Pro:

- Arrays are great if you want to read random elements, because you can look up any element in your array instantly.
- **Arrays are faster at reads.** This is because they provide random access.

Con:

- All elements in the array should be the same type (all ints, all doubles, and so on).

Linked lists

• With a list, elements are strewn all over, and one element stores the address of the next one.

Pro:

- **better at inserts.** adding an item to a linked list is easy: you stick it anywhere in memory and store the address with the previous item. With linked lists, you never have to move your items.
- inserting in the middle of a list is easier for lists, With lists, it’s as easy as changing what the previous element points to. But with arrays, we have to shift the rest of the elements down. And if there’s no space, you might have to copy everything to a new location!
- **lists are better at deletions,** because you just need to change what the previous element points to. With arrays, everything needs to be moved up when you delete an element.
    - Unlike insertions, deletions will always work. Insertions can fail sometimes when there’s no space left in memory. But you can always delete an element.
- Linked lists can *only* do sequential access.

Con:

- Linked lists are great if you’re going to read all the items one at a time: you can read one item, follow the address to the next item, and so on. But if you’re going to keep jumping around, linked lists are terrible.
- With a linked list, the elements aren’t next to each other, so you can’t instantly calculate the position of the fifth element in memory—you have to go to the first element to get the address to the second element, then go to the second element to get the address of the third element, and so on until you get to the fifth element.

Run times for common operatios on arrays and lists

	Arrays	Lists
Reading	O(1)	O(n)
Inserting	O(n)	O(1)
Deletion	O(n)	O(1)

*insertions and deletions are O(1) time only if you can instantly access the element to be deleted. It’s a common practice to keep track of the first and last items in a linked list, so it would take only O(1) time to delete those.

```
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

Recursion is where a function calls itself.

There’s no performance benefit to using recursion; in fact, loops are sometimes 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!

Loops vs. Recursion

```
# while loop
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print “found the key!”

# Recursion
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!”
```

Because a recursive function calls itself, it’s easy to write a function incorrectly that ends up in an infinite loop.

When you write a recursive function, you have to tell it when to stop recursing. That’s why *every recursive function has two parts: the base case, and the recursive case.* The recursive case is when the function calls itself. The base case is when the function doesn’t call itself again ... so it
doesn’t go into an infinite loop.

Call Stack

only two actions: *push* (insert) (added to the top of the list) and *pop* (remove and read the topmost item, and it’s taken off the list).

when you call a function from another function, the calling function is paused in a partially completed state. All the values of the variables for that function are still stored in memory.

Using the stack is convenient because you don’t have to keep track of a pile of boxes yourself—the stack does it for you.

Using the stack is convenient, but there’s a cost: saving all that info can take up a lot of memory. Each of those function calls takes up some memory, and when your stack is too tall, that means your computer is saving information for many function calls. At that point, you have two options:

- You can rewrite your code to use a loop instead.
- You can use something called *tail recursion*. That’s an advanced recursion topic that is out of the scope of this book. It’s also only supported by some languages, not all.

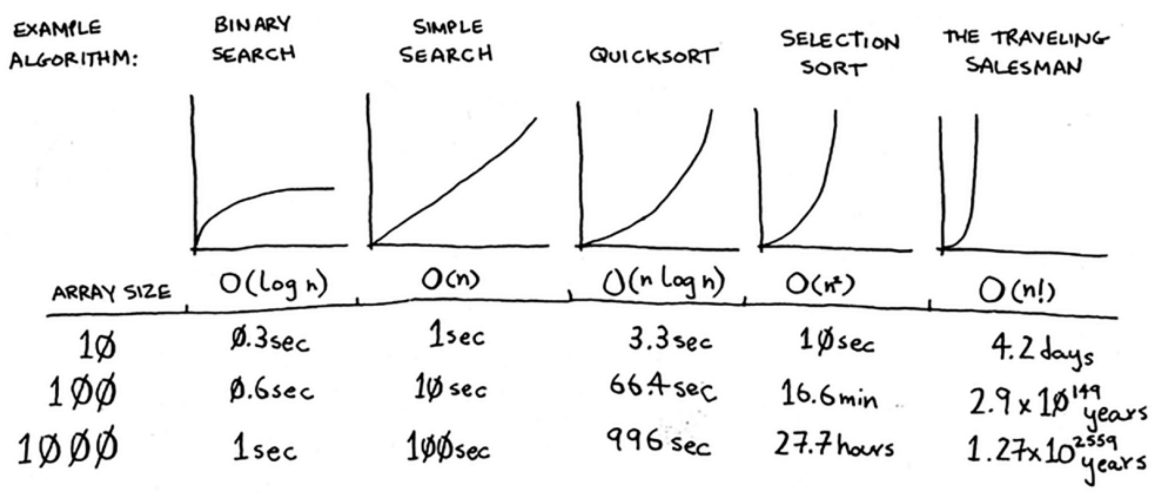
- All function calls go onto the call stack.
- The call stack can get very large, which takes up a lot of memory.

Quicksort

quicksort is an elegant sorting algorithm that’s often used in practice that uses divide-and-conquer.

D&C works by breaking a problem down into smaller and smaller pieces. If you’re using D&C on a list, the base case is probably an empty array or an array with one element. When you’re writing a recursive function involving an array, the base case is often an empty array or an array with one element. If you’re stuck, try that first.

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



EXAMPLE ALGORITHM:	BINARY SEARCH	SIMPLE SEARCH	QUICKSORT	SELECTION SORT	THE TRAVELING SALESMAN
ARRAY SIZE	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n!)$
1Ø	Ø.3sec	1sec	3.3sec	1Øsec	4.2 days
1ØØ	Ø.6sec	1Ø sec	66.4 sec	16.6min	$2.9 \times 1Ø^{149}$ years
1ØØØ	1sec	1ØØsec	996 sec	27.7 hours	1.27×10^{2559} years

for quicksort, in the worst case, quicksort takes O(n^2) time. in the average case, quicksort takes O(n log n) time.

Merge sort vs. quicksort

If quicksort is O(n log n) on average, but merge sort is O(n log n) always, why not use merge sort? Isn’t it faster?

Big O really means = c*n

we usually ignore constant because if 2 algorithms have different Big O times, the constant doesn’t matter since it doesn’t make a difference

> The constant almost never matters for simple search versus binary search, because O(log n) is so much faster than O(n) when your list gets big

but for quicksort and merge sort, the constant can make a difference. quicksort has a smaller constant than merge sort, if they are both O(n log n) time, quicksort is faster. And quicksort is faster in practice because it hits the average case way more often than the worst case.

Average case vs. worst case

The performance of quicksort heavily depends on the pivot you choose.

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.
In the worst case, there are O(n) levels, so the algorithm will take O(n) * O(n) = O(n2) time

the best case is also the average case. If you always choose a random element in the array as the pivot, quicksort will complete in O(n log n) time on average.