Chapter 4

Divide & Conquer (D&C)

↳ A well-known recursive technique to solve problems.

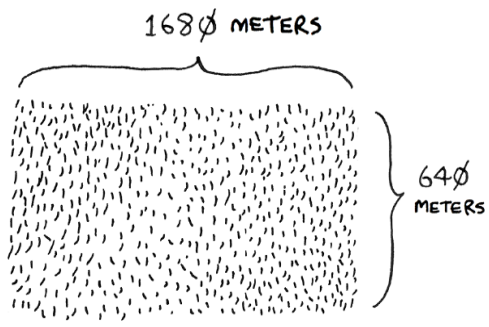
Quicksort

↳ A D&C algorithm

↳ Notably faster than selection sort.

Visual Example of D&C

Suppose that given this plot of land (1680m · 640m) you wanted to divide the plot evenly into square plots but you also want those plots to be as big as possible.



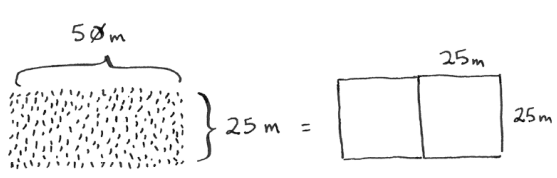
Fortunately, there are two ways to achieve this:

1. Figure out the base case ← Simplest Possible Case
2. Divide or decrease your problem until it becomes the base case.

To Solve This:

↳ Exactly: Identify your base case

↳ The easiest case would be if one side was the multiple of the other side.



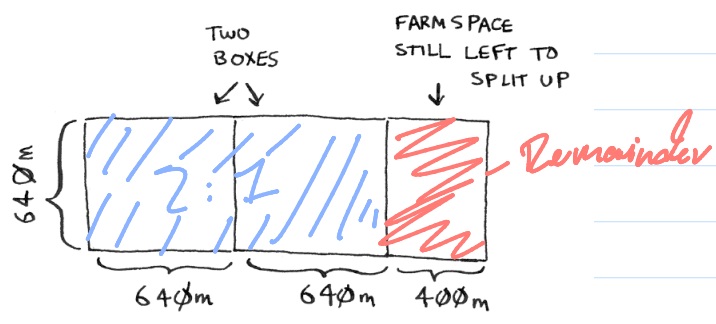
$(50m \cdot 25m) = (25m \cdot 25m) \cdot 2$

2 : 1 — 1 : 1

Now you just need to figure out what the recursive case, this is where D&C comes in.

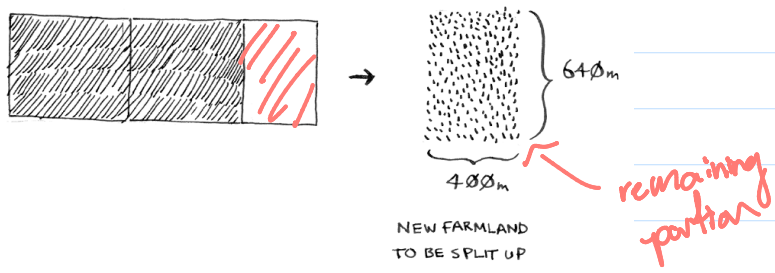
↳ With every recursive call, reduce your problem.

↳ How? Lets mark out the biggest boxes you can use with the property last mentioned



Now as you can tell, there is still a remaining portion of land that isn't like the other two.

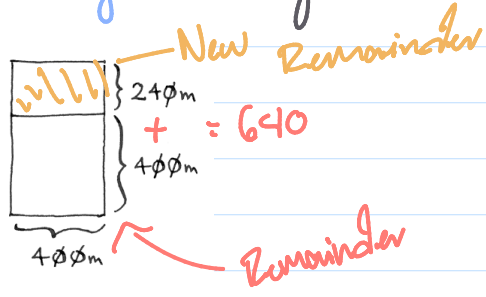
↳ Why don't we apply the same algorithm with the remaining portion?



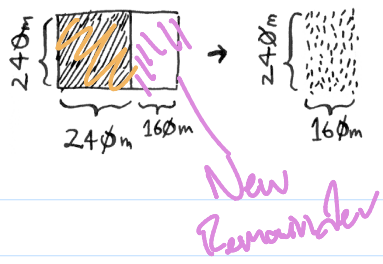
So, you started with your 1680m · 640m plot of land, but now since you split it up to even squares you need to split up your 640 · 400m remaining plot of land.

↳ If you find the biggest box that will work for this size, then that will be the biggest box that will work for the entire 1680m · 640m farm.

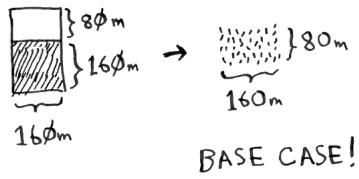
Now, let's apply that same algorithm again:



And again...

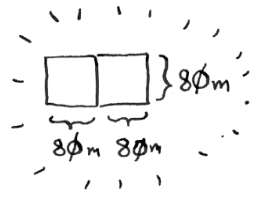


And Again till you see the base case... (2:1)

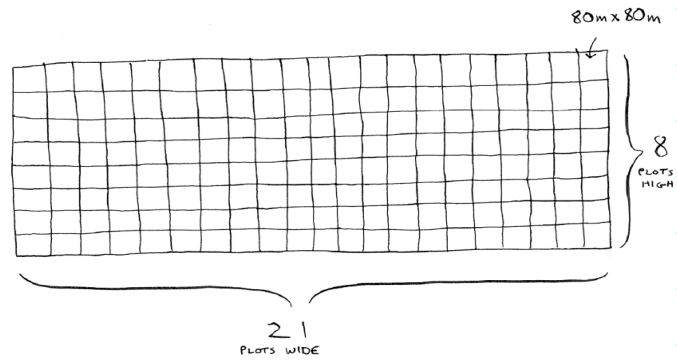


Remember, the base case indicated that one side must be a multiple of the other side.

↳ Meaning... there's nothing left to do!



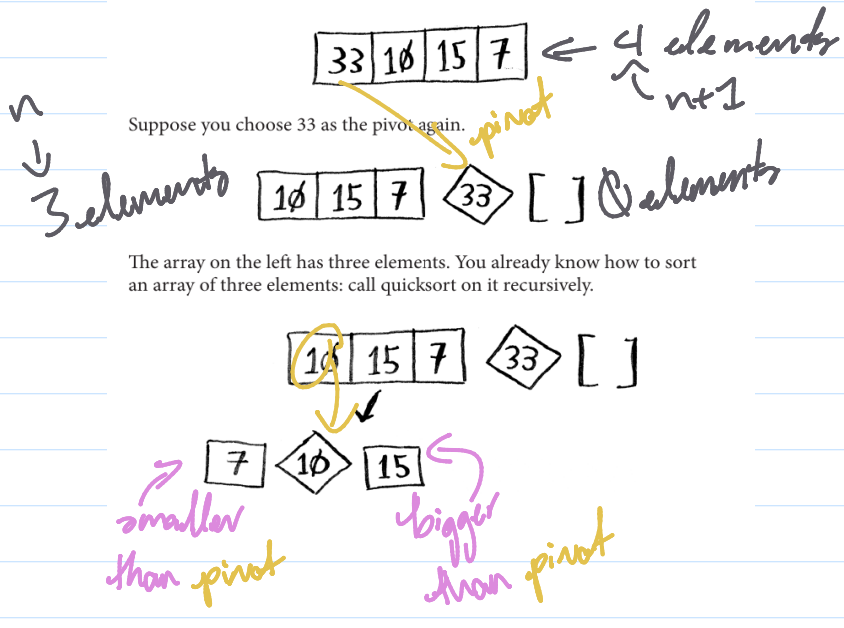
Thus, since we know that if we found the size that reduces the base case, it will also be the biggest square for the entire 1680m x 640m plot of land.



Recap:
- D&C works by:
  1. Figuring out a simple case as the base case.
  2. Figuring how to reduce your problem and getting to the base case.

Quicksort
- Notably faster than selection sort.
  - Uses D&C
- Pivot:
  - A chosen element in an array.
    - Usually it's chosen to divide the given array into sub-arrays.
- Partitioning a.k.a.
  - The act of dividing a given array into two subarrays according to the location of the pivot.
    - How? Comparing all elements to the pivot and creating two subarrays that are "less" than and "greater" than the pivot.
- Recursively Call QuickSort
  - Assuming that you understand what the base case is (a case where no sorting is necessary, aka 1 or 0 element arrays) — Concept 1
    - + Sorting 2 element arrays by comparison (if statement) — Concept 2
  - Then, to recursively sort your sub-arrays, apply the same two concepts to the sub-arrays.
    - Pivot location will not matter.

↳ If you repeat the concept of: "How to quicksort array of n+1 array" when you figured out how to sort an array of n elements, then nothing will stop you from doing so.



Here's the code for quicksort:

```
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)

print(quicksort([10, 5, 2, 3]))
```

Big O Notation: Revisited

↳ Quicksort's speed is dependent on the pivot you choose.

↳ Here's the most common runtimes under Big O

EXAMPLE ALGORITHM:	BINARY SEARCH	SIMPLE SEARCH	QUICKSORT	SELECTION SORT	THE TRAVELING SALESMAN
ARRAY SIZE	O(log n)	O(n)	O(n log n)	O(n²)	O(n!)
10	0.3 sec	1 sec	3.3 sec	10 sec	4.2 days
100	0.6 sec	10 sec	664 sec	16.6 min	2.9×10^{149} years
1000	1 sec	100 sec	996 sec	27.7 hours	1.27×10^{2559} years

Quicksort in it's worst case will take $O(n^2)$ time

- On average quicksort takes $O(n \log n)$ time.

Rethinking Big O:

↳ Suppose that you have a function to print all items in an array.

```
def print_items(myList):
  for item in myList:
    print(item)
```

↳ Now you have another function that does the same thing, but pauses for 1 second before printing the next element.

```
from time import sleep
def print_items2(myList):
  for item in myList:
    sleep(1)
    print(item)
```

The funny part about this is that both functions have the same time complexity, although the second function is not practical.

↳ When you write in Big O notation it usually means:

$c * n$

SOME FIXED AMOUNT OF TIME

constant (a.k.a "some fixed amount of time")

But when comparing two algorithms with different Big O times the constant shouldn't matter

↳ Why? Imagine comparing Binary Search with Simple Search and they had different constants

$10ms * n$ — SIMPLE SEARCH

$1sec * \log n$ — BINARY SEARCH

rate of performance

Some would probably assume that since Simple Search have a constant of 10 ms it would perform better than Binary Search with a constant of 1 second, right?

↳ No! Why? Imagine you are searching through 4 billion elements!

SIMPLE SEARCH	10ms * 4 BILLION	=	463 days
BINARY SEARCH	1sec * 32 ($\log_2$(4 billion))	=	32 seconds

> Massive Difference!

As you can see, the rate of performance is consistent, thus most of times, constants don't matter.

↳ But there are cases in which constants DO matter!

↳ Merge Sort vs Quicksort

↳ Given that both are implemented the appropriate way, if both algorithms have $O(n \log n)$ time complexity, then quicksort is faster in applicability

↳ Quicksort is also faster in practice because it hits the average case more often than the worse case.

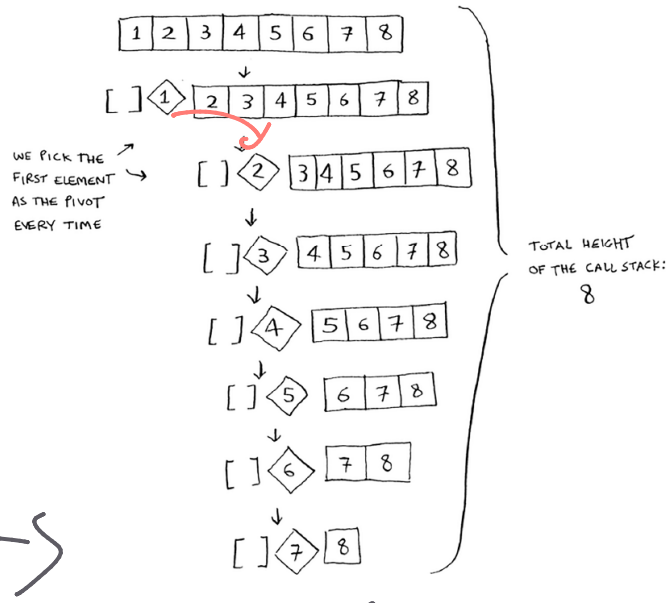
Merge Sort

↳ Constant $O(n \log n)$ time complexity.

Average Case vs Worst Case

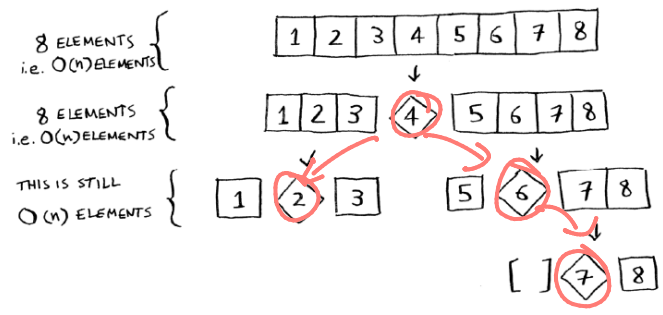
↳ The performance of Quicksort heavily depends on the pivot you choose.

↳ Suppose you always choose the first element as the pivot and you call quicksort on an already sorted array...



If you haven't noticed it yet, you haven't divided the arrays evenly thus making the call stack too long. ← Inefficient

↳ Instead, always pick the middle element as your pivot...

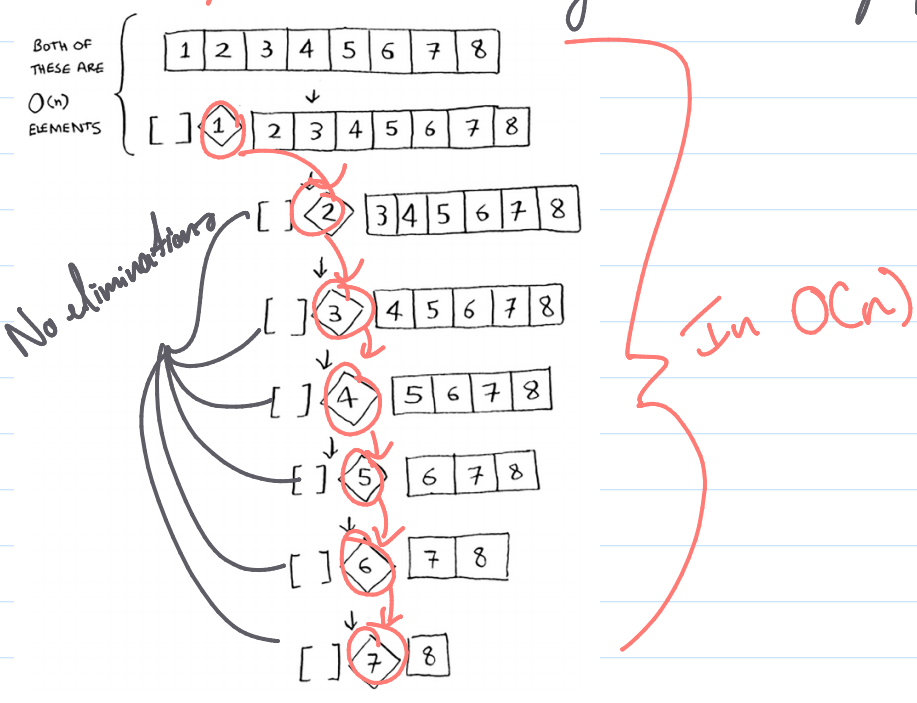


Very short right? You hit the base case much more sooner!

↳ 1st Example (8 iterations): Worst-case scenario. O(n) ← Stack Size

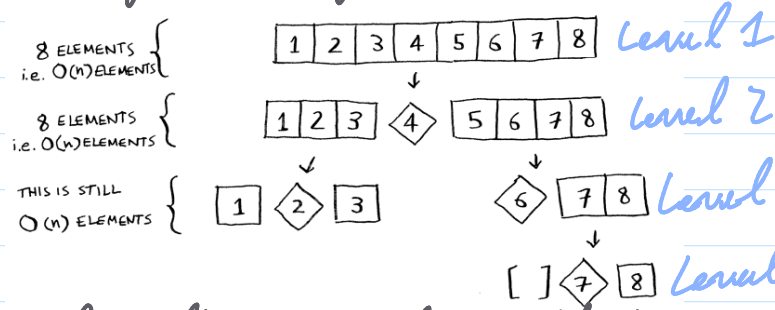
↳ 2nd Example (4 iterations): Best-case scenario. O(log n)

In explanation for why the stack size is in O(n) is because: You are touching every element as the pivot in the array, not allowing for any eliminations.



Example 2 Explained

↳ Think of how Binary Search works, it is by consecutive halfing of the array, which is in $O(\log n)$, this example brings into light with Quicksort.



why? Because we halved the worst-case scenario of 8 levels ↳ to 4!

For this example, the height stack for this example would be: $O(\log n)$

↳ Each level would still take $O(n)$ time to finish.

↳ The entire algorithm will take $O(n) \cdot O(\log n) = O(n \log n)$ time.

↳ Worst-case: $O(n) \cdot O(n) = O(n^2)$ time.

Unless all elements are the same, then you hit the worse-case.

No worries though! The best case is also the average case, meaning that if you chose any random pivot, quicksort will complete it in $O(n \log n)$ time on average.