MONASH College

# MCD4710

Introduction to algorithms and programming

## Lecture 12

Decrease and Conquer

# Objectives

Objectives of this lecture are to:

1. Know to search efficiently in ordered sequence (Binary Search)
2. Understand design paradigm decrease-and-conquer and recognise situations with logarithmic complexity
3. Demonstrate the time complexity of Euclid’s algorithm

This covers learning outcomes:

- 2 – choose and implement appropriate problem solving strategies
- 5 – determine the computational cost and limitations of algorithms

# Overview

1. The Ordered Search Problem
2. Binary Search
3. Revisiting Euclid’s Algorithm

# Search in Ordered Sequence

Gertrudis Atkinson 0463935372

Kiley Basinger 0411484152

Romana Brose 0418721183

Shayne Brotherton 0436242684

Calandra Clifton 0479753034

Roy Dupuis 0445778949

Leticia Fukushima 0436756947

Cherlyn Gayles 0483503919

…

Problem: find (the position of) a name in a phone book.

# Search in Ordered Sequence

ato.gov.au 180.149.195.3

cancer.org.au 52.187.229.23

facebook.com 31.13.71.36

google.com 172.217.12.142

monash.edu 43.245.43.30

newscientist.com 45.60.19.101

news.com.au 23.221.48.198

wikipedia.org 208.80.154.224

…

Problem: find URL in DNS records.

# Searching Algorithms

- Given a list of data, find the index of a particular value or return that the value is not present
- Linear search
  - start at first item
  - is it the one I am looking for?
  - if not, go to next item
  - repeat until target is found or all items checked
- If the items are not sorted or cannot be sorted, this approach is necessary

# Sequential Search

- Given a target value.

- Find the first item in a list, L, which has the value target.

- If target is found then return the index of the item.

- If target not found then return -1.

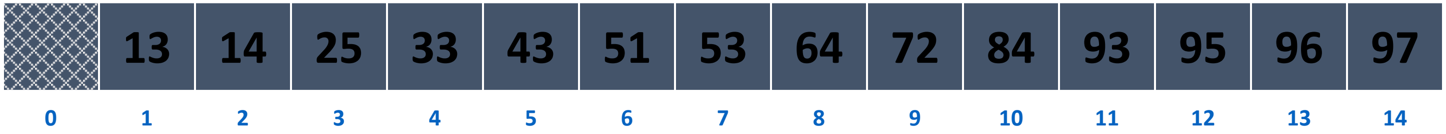
# Linear Search

- Ex. Linear search for 33.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search

- Ex.  Linear search for 33.



| ▨ | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

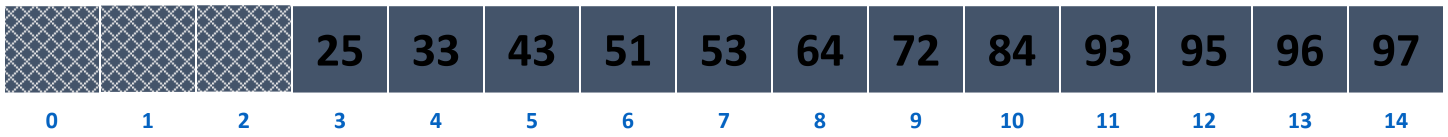
# Linear Search

- Ex.  Linear search for 33.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▨ | ▨ | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search

- Ex.  Linear search for 33.



| ▨ | ▨ | ▨ | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

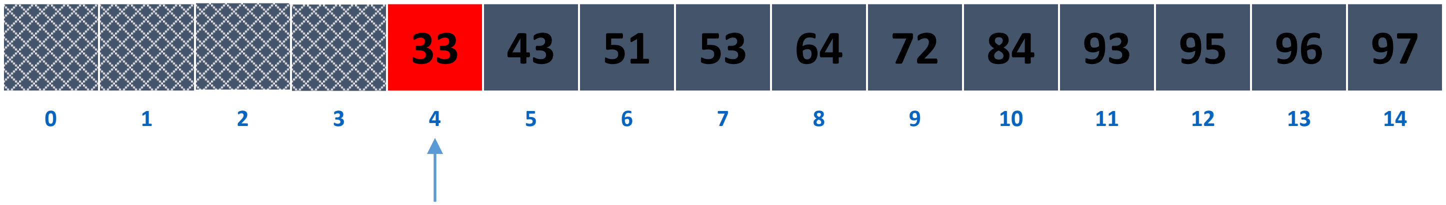
# Linear Search

- Ex.  Linear search for 33.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▨ | ▨ | ▨ | ▨ | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search

- Ex.  Linear search for 33.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▨ | ▨ | ▨ | ▨ | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Return index: 4

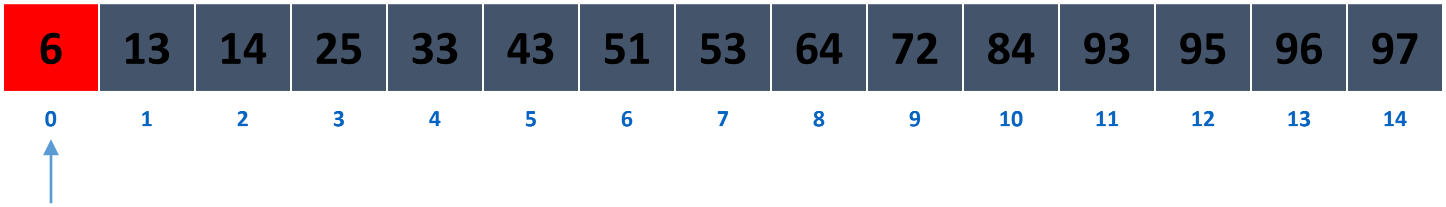
# Linear Search – Best Case

- The best case would be if we are looking for an item and find it on our first compare.
- In this example, that means our target would be 6.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **6** | **13** | **14** | **25** | **33** | **43** | **51** | **53** | **64** | **72** | **84** | **93** | **95** | **96** | **97** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search – Best Case

- Ex. Linear search for 6.
- We find the target immediately



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Return index: 0

# Linear Search – Worst Case

- The worst case would be if we are looking for an item that’s not in the list.
- In this example, if we are looking for the item 8 we would have to search through the entire list.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search – Worst Case

- Ex.  Linear search for 8 (item not in list).

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▨ | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search – Worst Case

- Ex.  Linear search for 8 (item not in list).

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▨ | ▨ | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search – Worst Case

- Ex.  Linear search for 8 (item not in list).

| ▨ | ▨ | ▨ | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

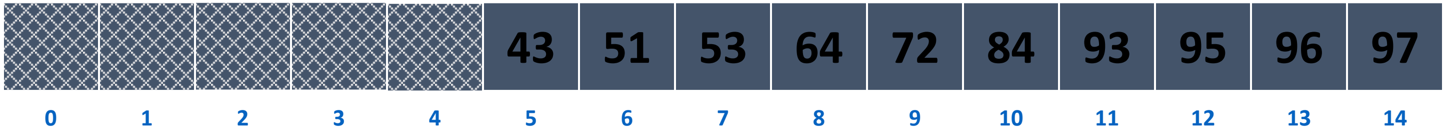
# Linear Search – Worst Case

- Ex.  Linear search for 8 (item not in list).

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▨ | ▨ | ▨ | ▨ | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Linear Search – Worst Case

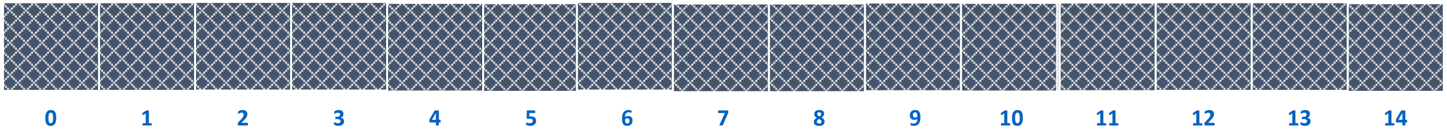
- Ex.  Linear search for 8 (item not in list).



| ▨ | ▨ | ▨ | ▨ | ▨ | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

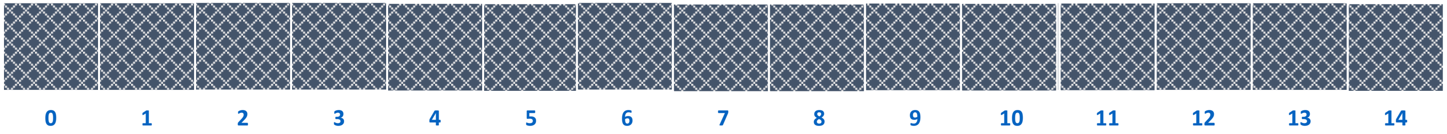
# Linear Search – Worst Case

- Ex.  Linear search for 8 (item not in list).



# Linear Search – Worst Case

- Ex.  Linear search for 8 (item not in list).
- This means that for a list of size ***n***, it would take us *n* compares to find out that the target is not in the list.



# Linear Search implementation

```python
def linearSearch(aList, target):
# Returns index if target is in aList, -1 if it is not
      for index in range(len(aList)):
            if target == aList[index]:
                  return index
      return -1

testlist = [0, 1, 2, 18, 13, 17, 19, 32, 42, 15]
print(linearSearch(testlist, 13))
```

Invariant : At the *i*th iteration, *array[0..i-1]* consists of elements that are not equal to *target*.

# What is the worst case time complexity of linear search?

A. O(1)

B. O(n)

C. O(n^2)

D. O(log(n))

1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol and enter the code: UF7BD9
4. Answer questions when they pop up.

# We can do better than that!

## Binary Search

The list must be sorted

# Decrease-and-Conquer: reduce problem to smaller subproblem

- If items are sorted then we can *decrease and conquer*
- decreasing the search space in half with each step
  - generally a good thing
- Binary Search
  - Start at middle of list
  - is that the item we are looking for?
  - If not, is it less than or greater than the item?
  - less than, move to second half of list
  - greater than, move to first half of list
  - repeat until found or sub list size = 0

- Binary Search
  - Start at middle of list
  - is that the item we are looking for?
  - If not, is it less than or greater than the item?
  - less than, move to second half of list
  - greater than, move to first half of list
  - repeat until found or sub list size = 0

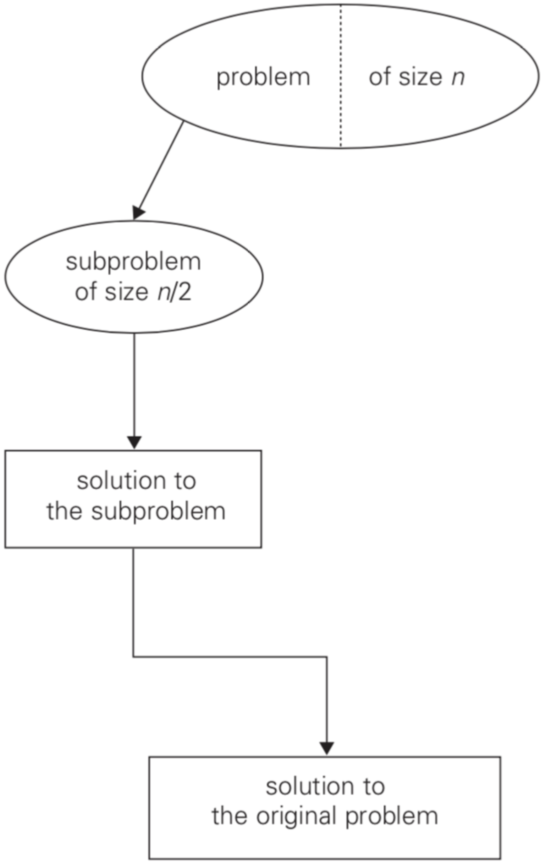
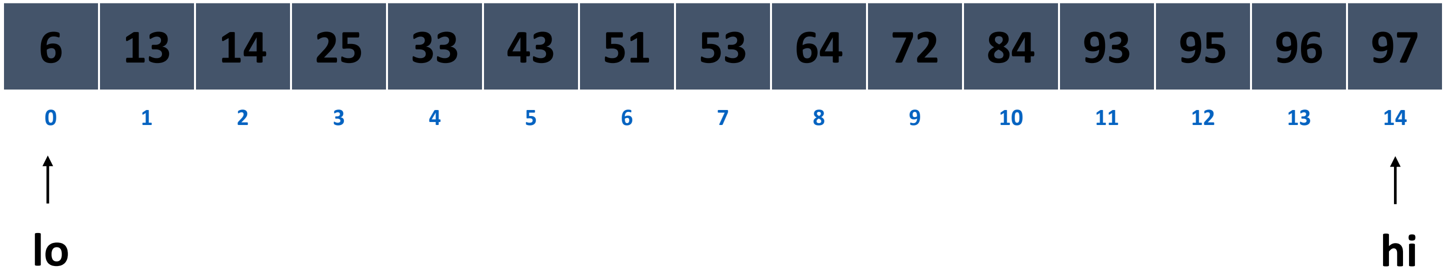


illustration: [Levitin, p. 133]

# Binary Search

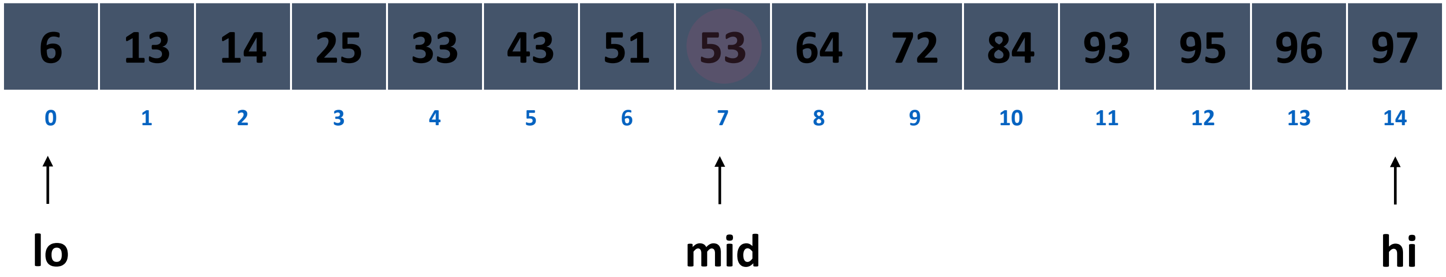
- Ex. Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

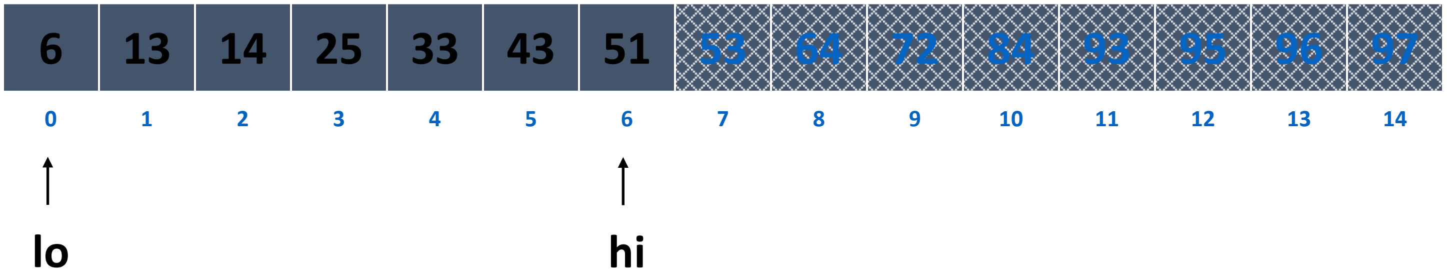
- Ex. Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

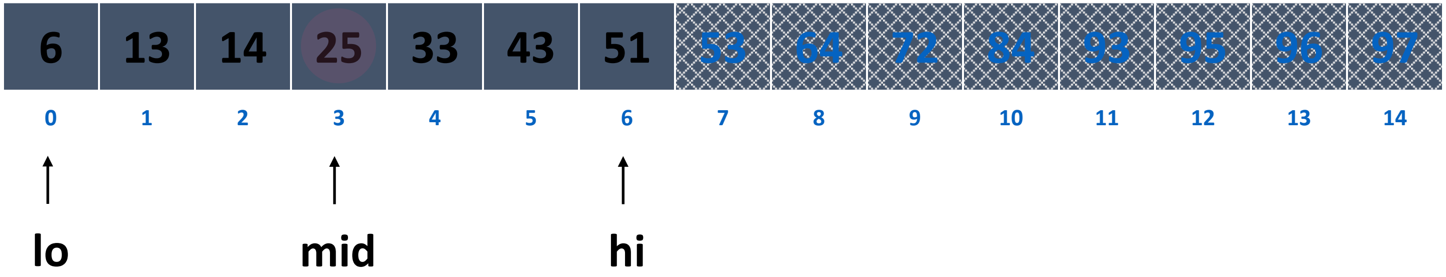
- Ex. Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

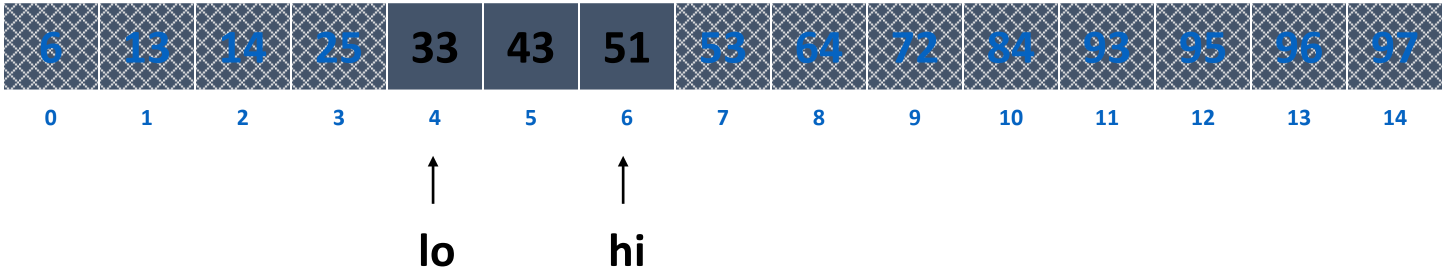
- Ex. Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

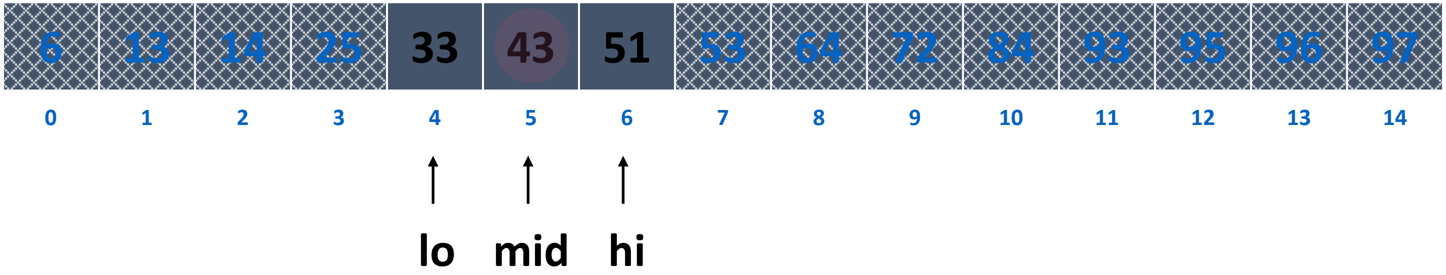
- Ex. Binary search for 33.



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

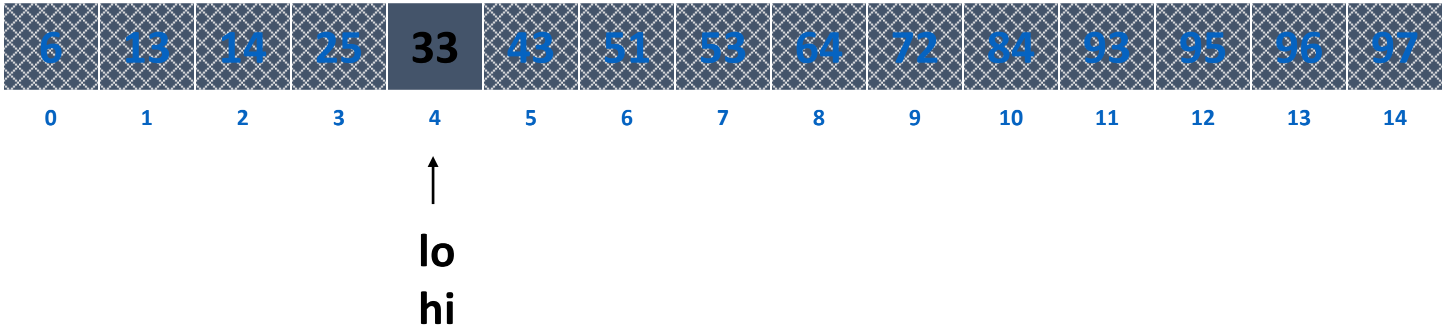
- Ex.  Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

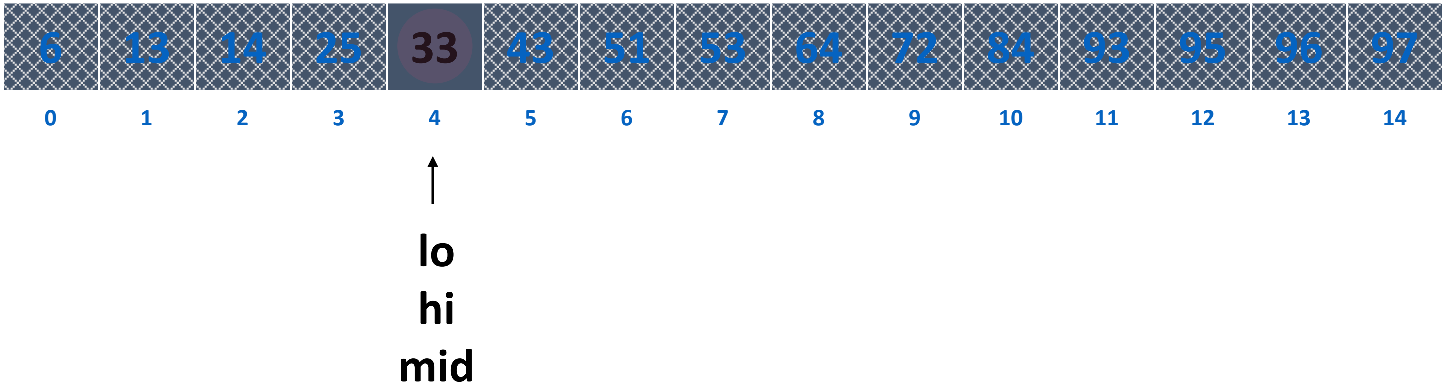
- Ex. Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

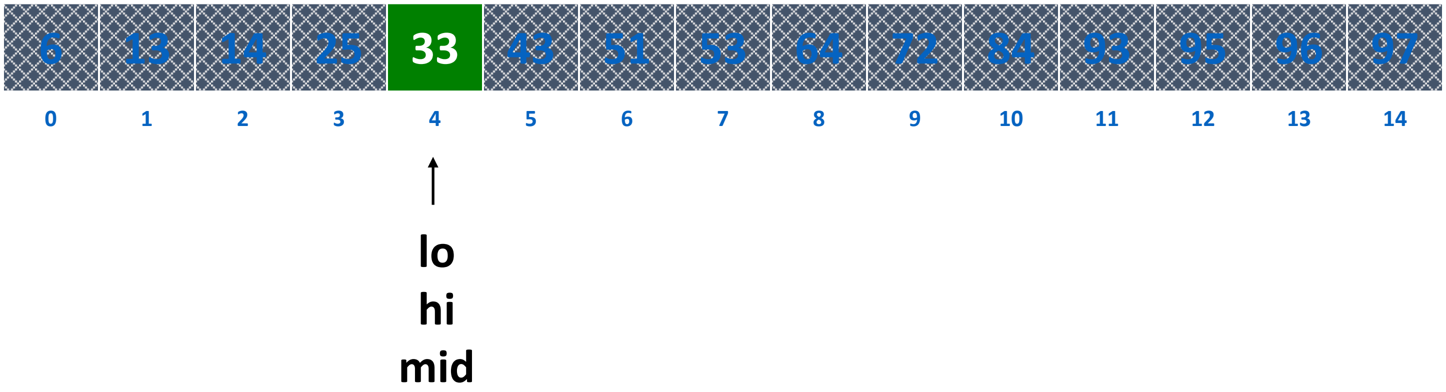
- Ex. Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search

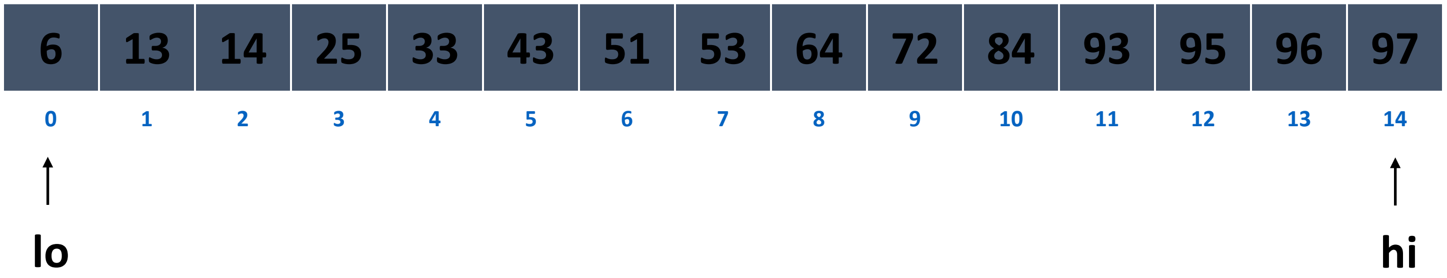
- Ex. Binary search for 33.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Best Case

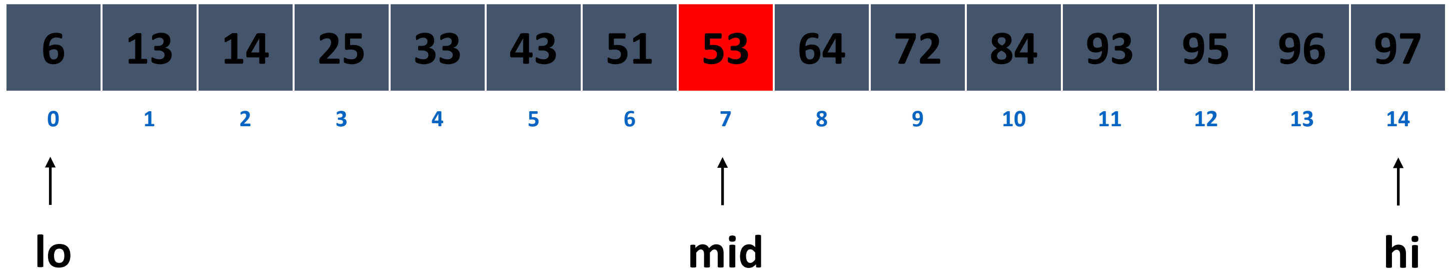
- Just like for linear search, the first item we compare is our target.
- In this case, 53.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |

# Binary Search – Best Case

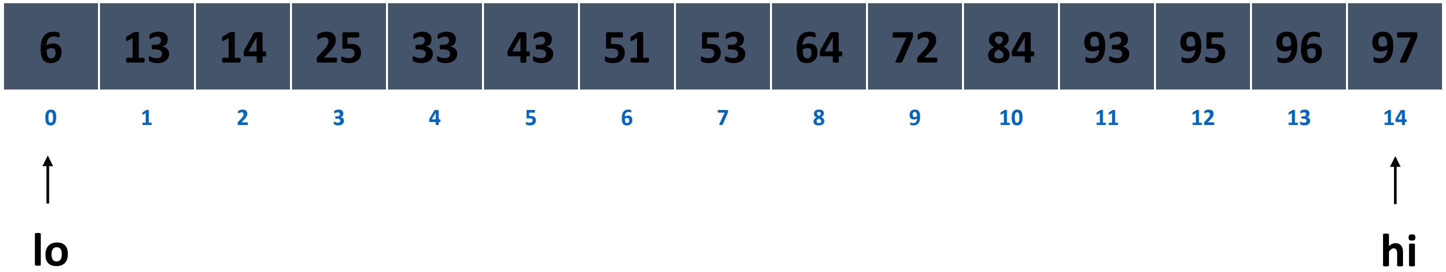
- Just like for linear search, the first item we compare is our target.
- In this case, 53.



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

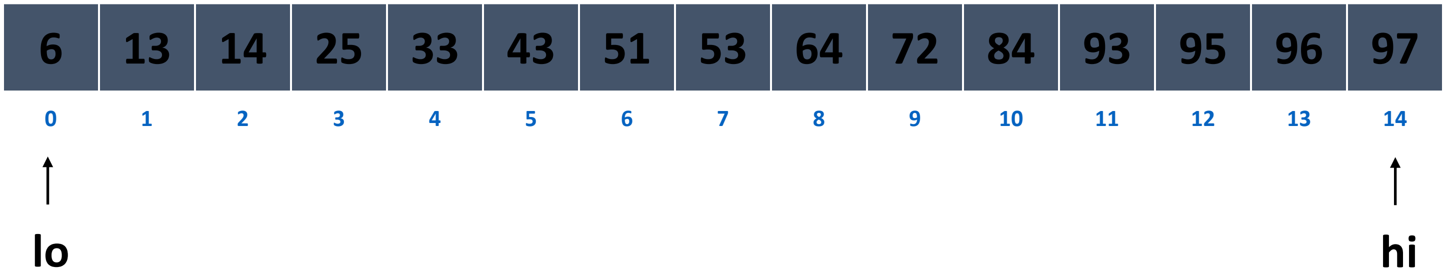
- Just like for linear search, the worst case would be if the item we are looking for is not in the list (e.g. 12)



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

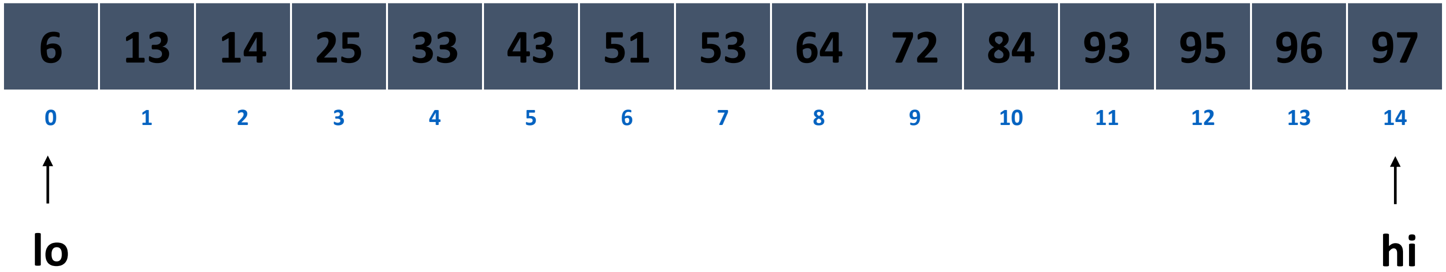
- Unlike linear search, in binary search we would not have to search through the entire list to find out that our target is not in the list.



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

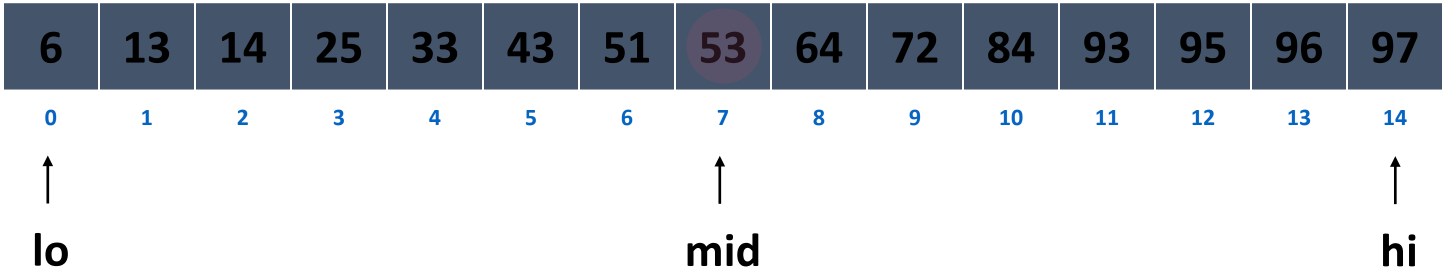
- Ex.  Binary search for 34.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

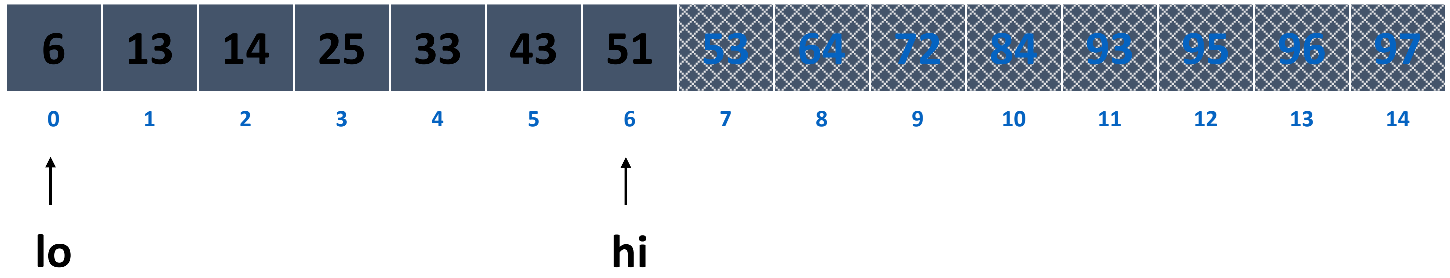
- Ex. Binary search for 34.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| lo | | | | | | | mid | | | | | | | hi |

# Binary Search – Worst Case

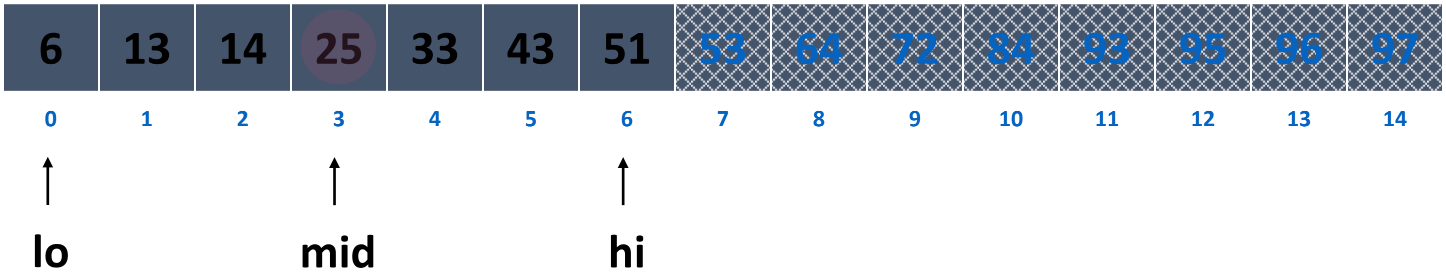
- Ex.  Binary search for 34.



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

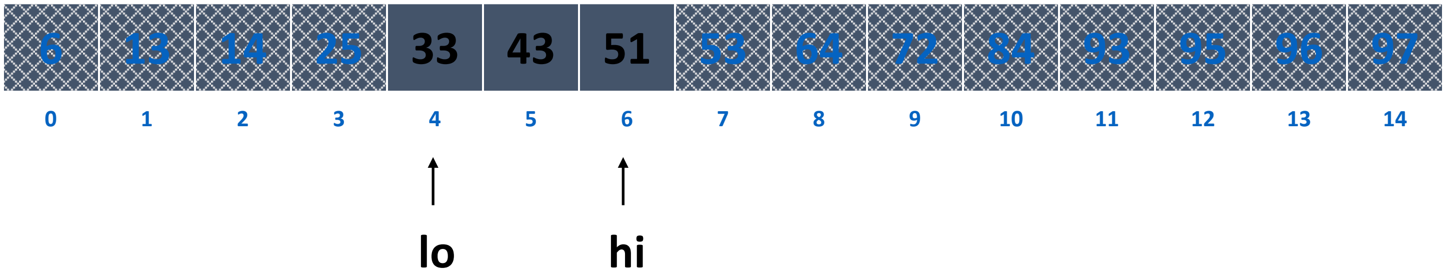
- Ex.  Binary search for 34.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

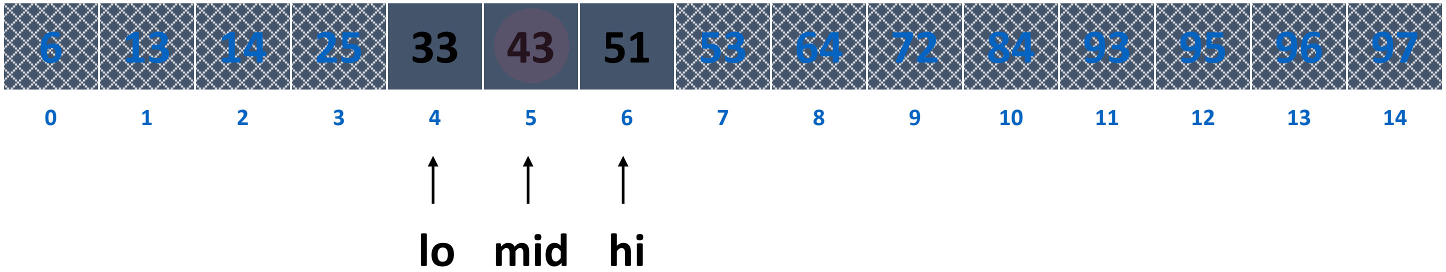
- Ex.  Binary search for 34.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

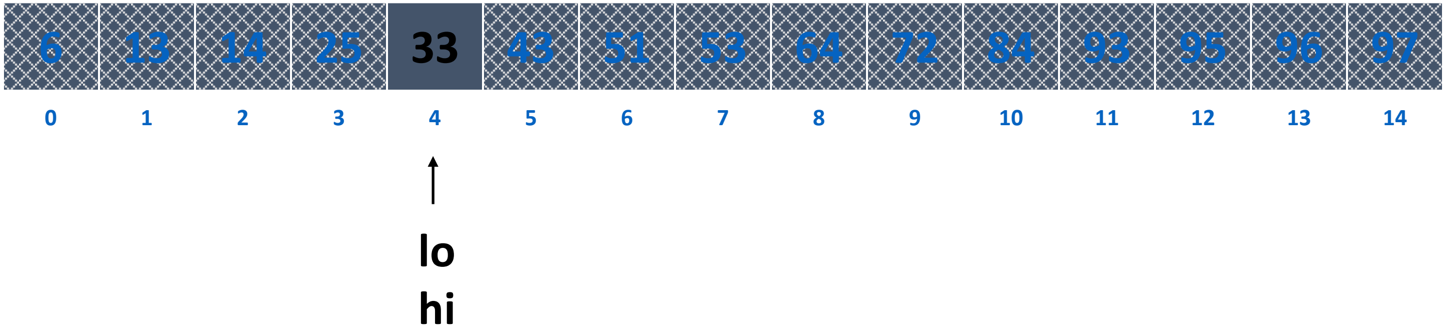
- Ex. Binary search for 34.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

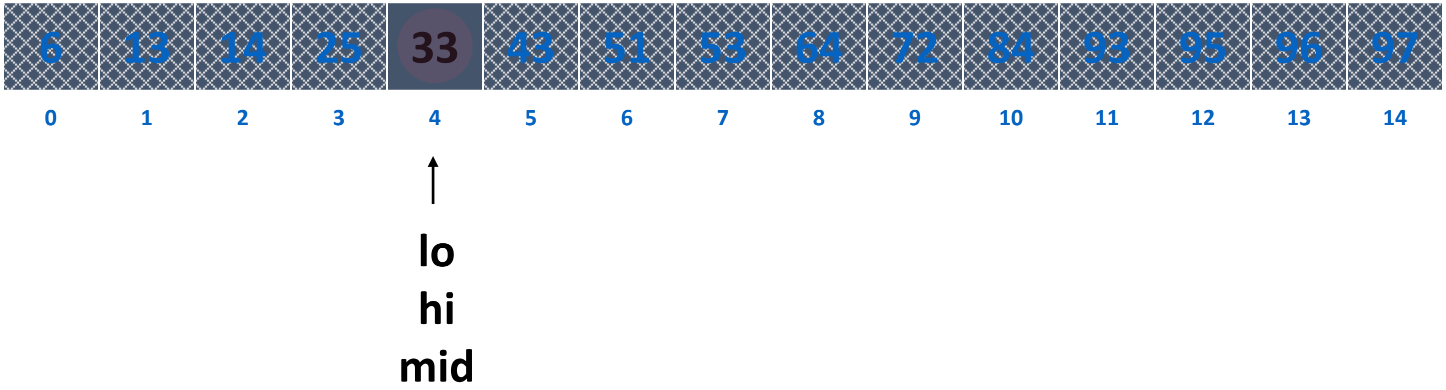
- Ex. Binary search for 34.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

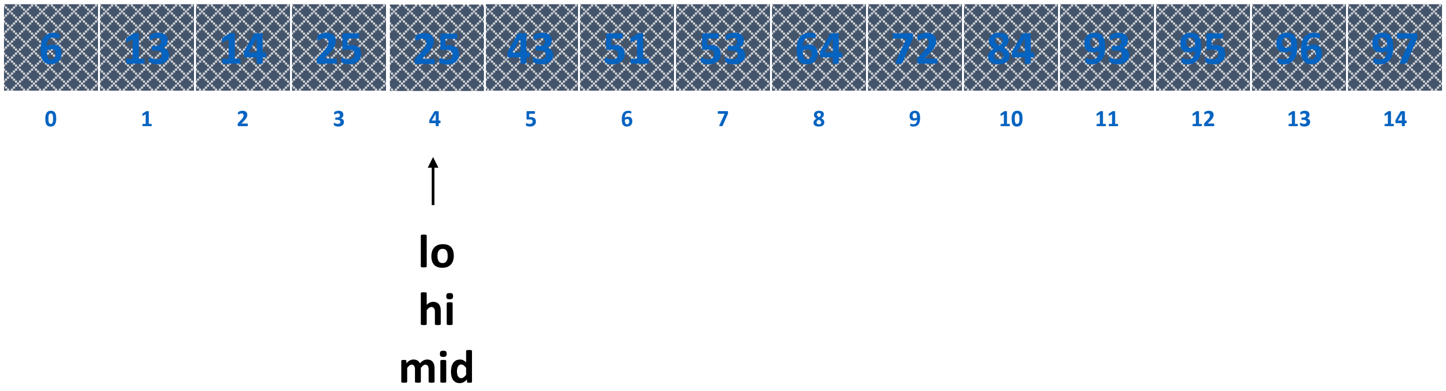
- Ex. Binary search for 34.



| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

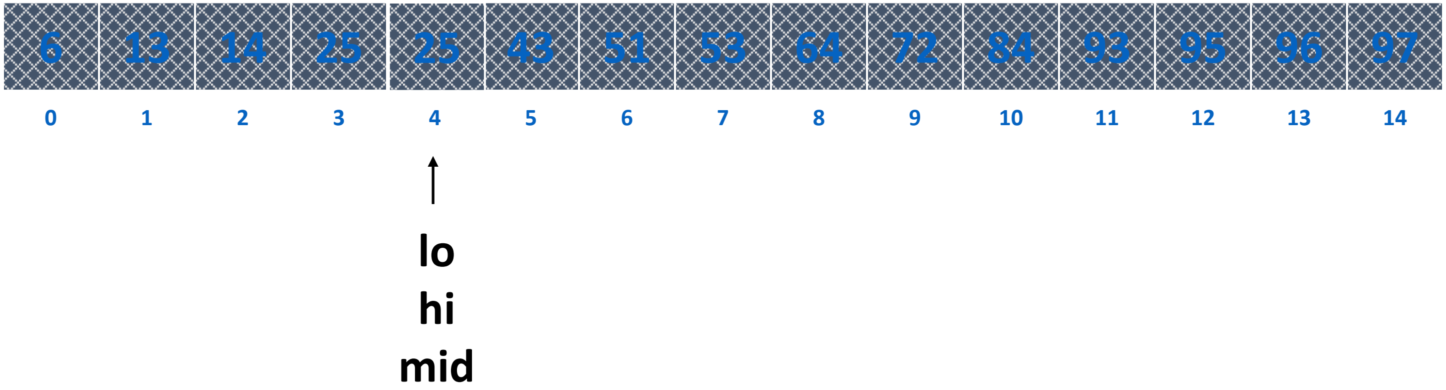
- Ex.  Binary search for 34.



| 6 | 13 | 14 | 25 | 25 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

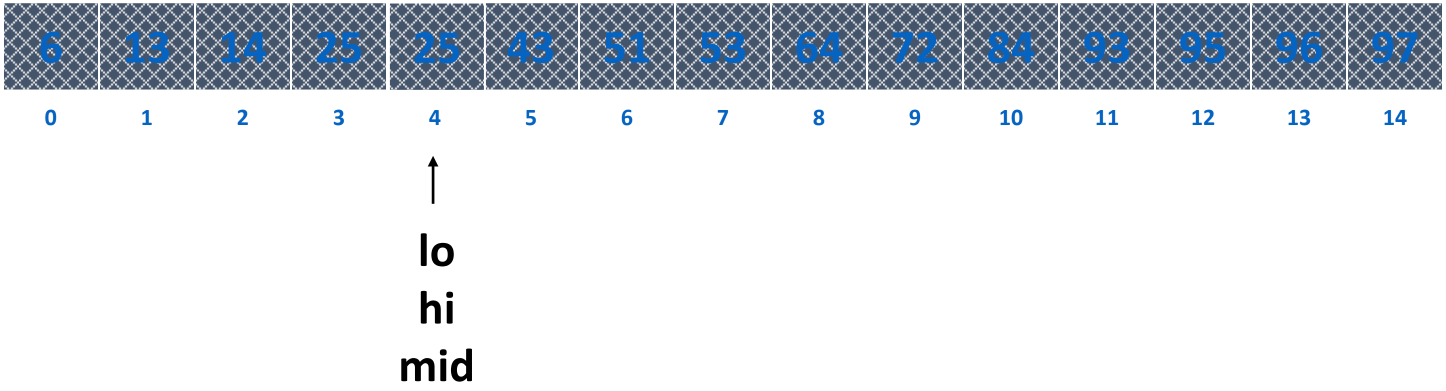
- How many comparisons would we have to make?



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 25 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

- How many comparisons would we have to make?
- Since we are dividing the list in half each time, how many times can we divide a list of length ***n*** by 2 before it reaches 1.



| 6 | 13 | 14 | 25 | 25 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Binary Search – Worst Case

- How many times can I divide *n* before it equals 1?
- If I can divide *n*, *k* times before I get 1 then:

$$2^k = n$$

and

$$k = log_2 n$$

- This means it would take me $log_2 n$ compares to find out that the item is not in the list.

# Binary Search – $O(\log_2 N)$

- We split the search space in half each time.

- If we had 100 items
  - 100 → 50 → 25 → 12 → 6 → 3 → 1

- If we doubled it to 200 items, only 1 more search!
  - 200 → 100 → 50 → 25 → 12 → 6 → 3 → 1

- Each time we double the amount of items, it only takes 1 extra iteration of the search algorithm.

# Binary Search

```
def binarySearch(aList, target):
# Returns index if target is in aList, -1 if it is not
# Assumes aList is sorted in ascending order
       low = 0
       high = len(aList)-1
       while low <= high:
              mid = (low + high) // 2
              if aList[mid] == target:
                     return mid
              elif aList[mid] > target:
                     high = mid - 1
              else: low = mid + 1
       return -1 # item not in list

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42]
print(binarySearch(testlist, 13))
```

# What is the worst case time complexity of binary search?

A. O(1)
B. O(n)
C. O(n^2)
D. O(log(n))

1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol and enter the code: UF7BD9
4. Answer questions when they pop up.

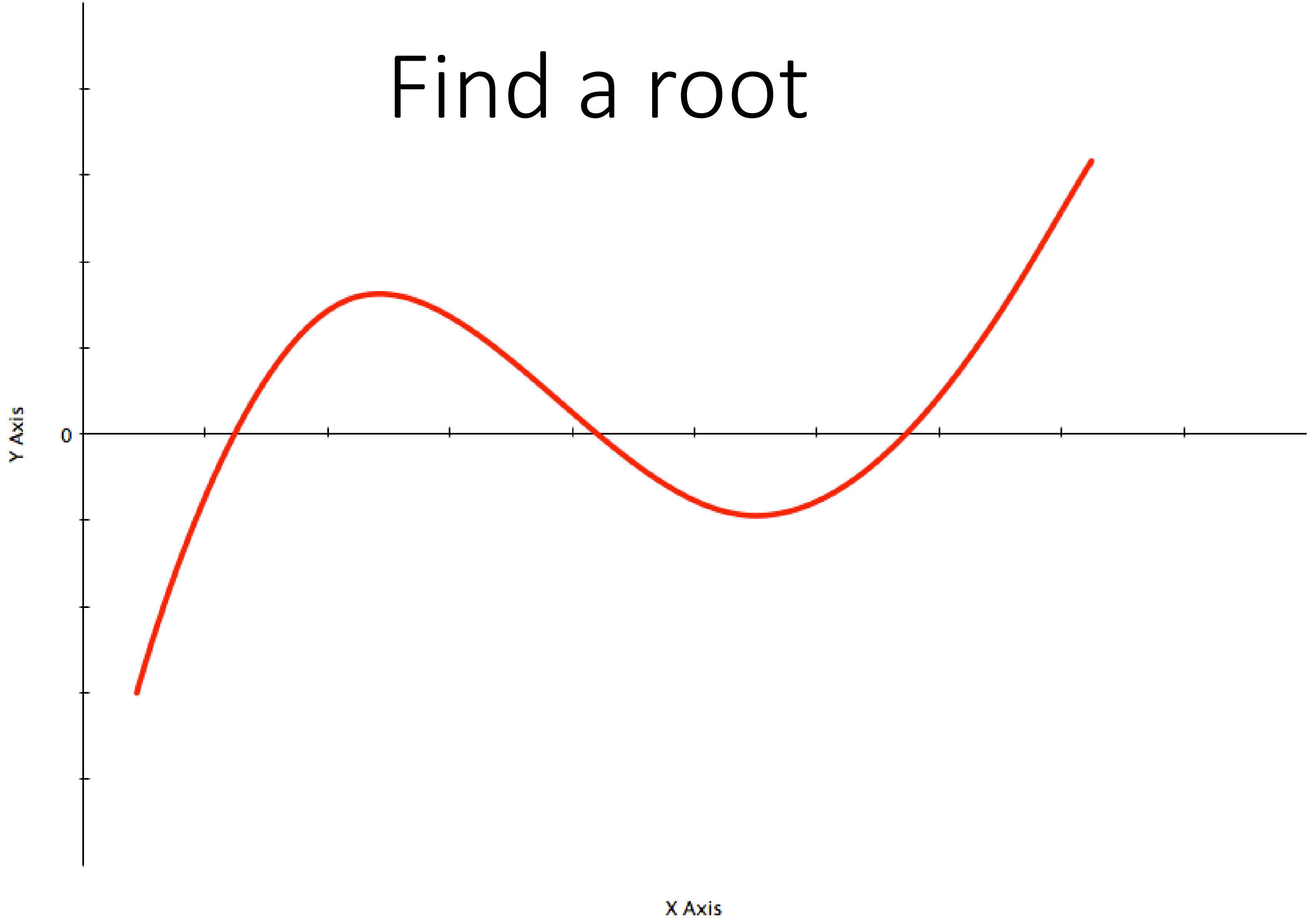
# Binary Search – O(log N)

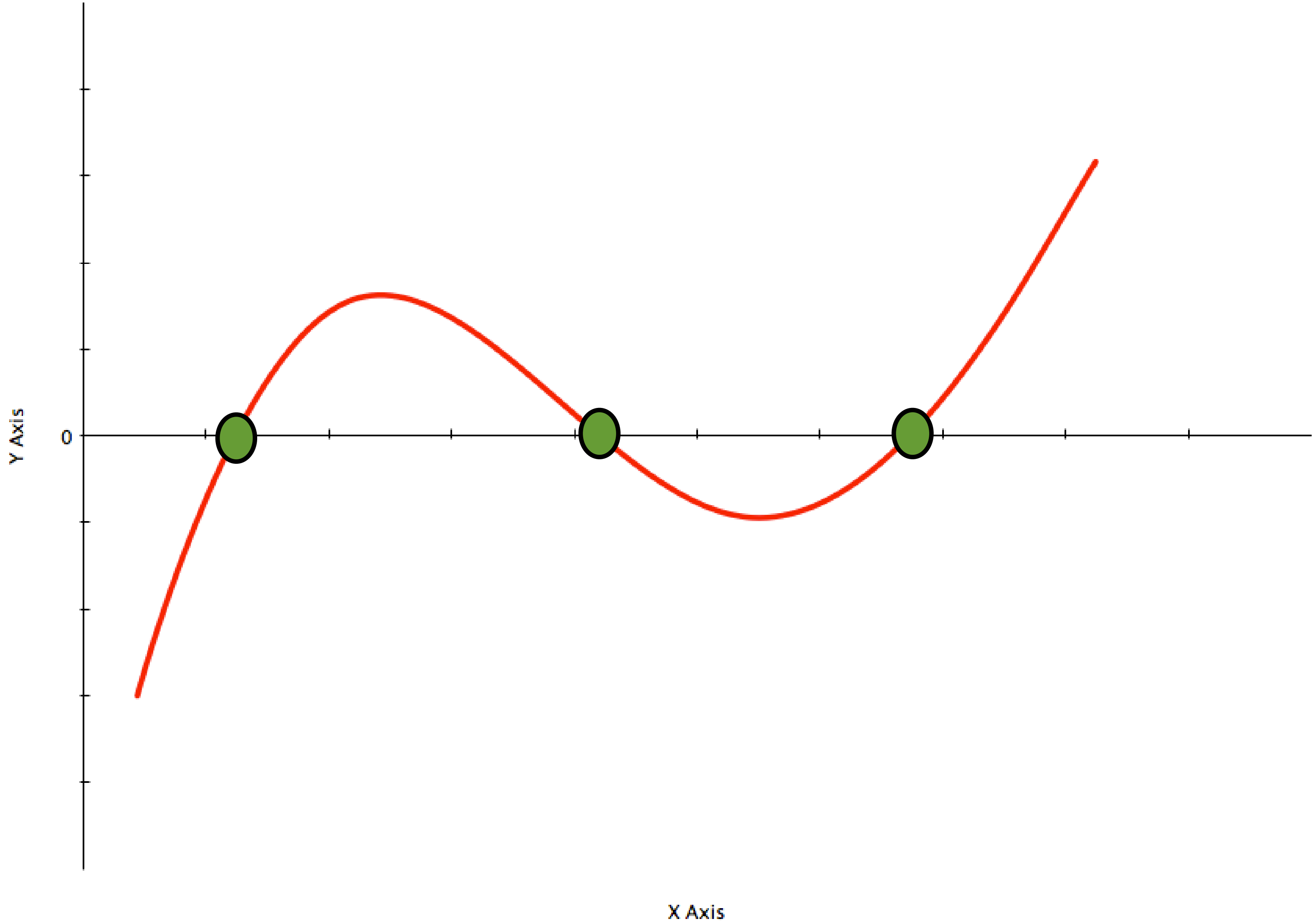
- 1 items = 1 search
- 2 items = 2 searches
- 4 items = 3 searches
- …
- 1024 items = 10 searches
- 2048 items = 11 searches
- …
- 1 million items ~= 20 searches
- 1 billion items ~= 30 searches

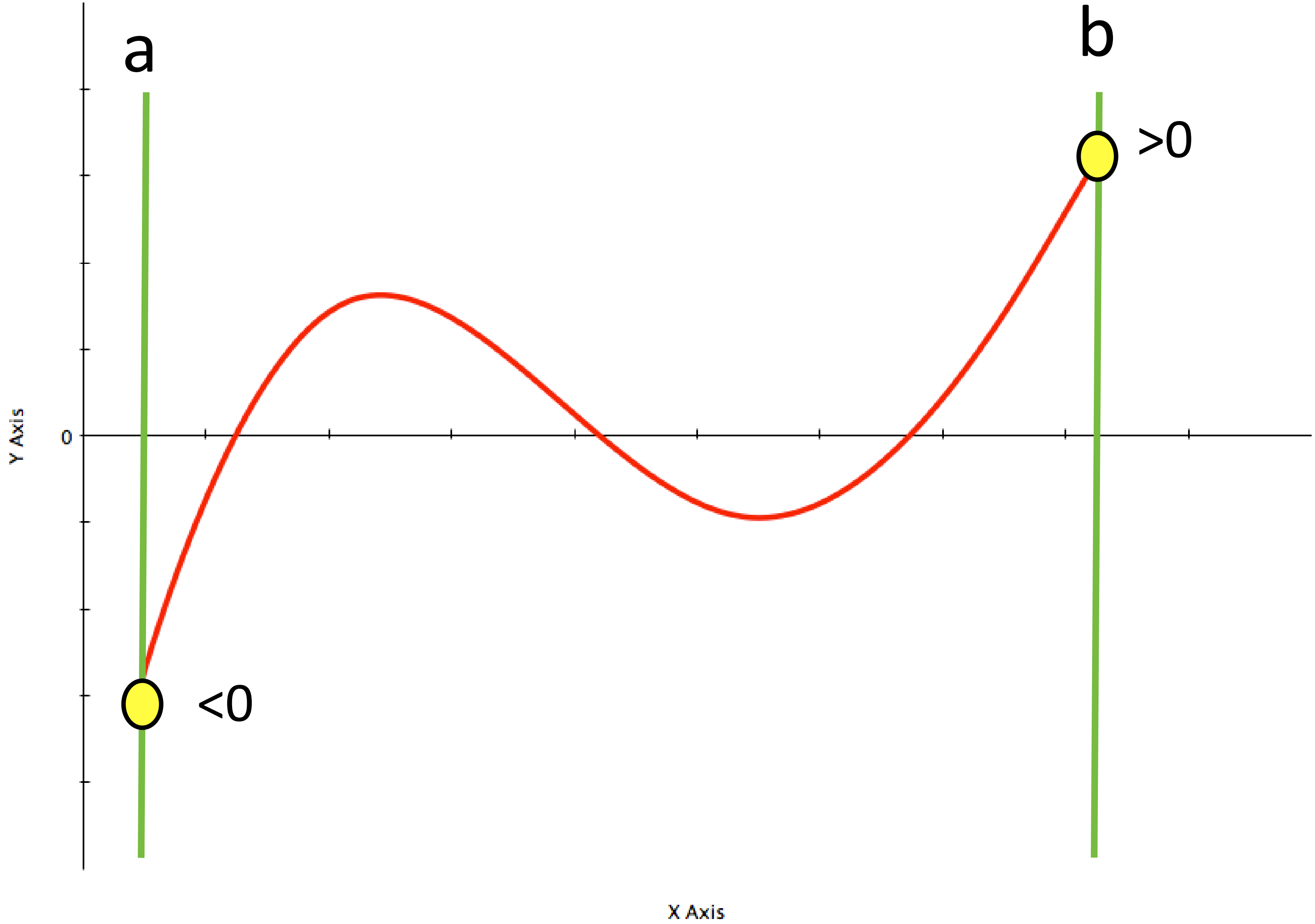
- Logarithmic!
- Binary Search = *O*(*log N*)
  - Compare to Linear Search *O*(*N*)

# Binary Search Application- Find the root (Bisection Method)

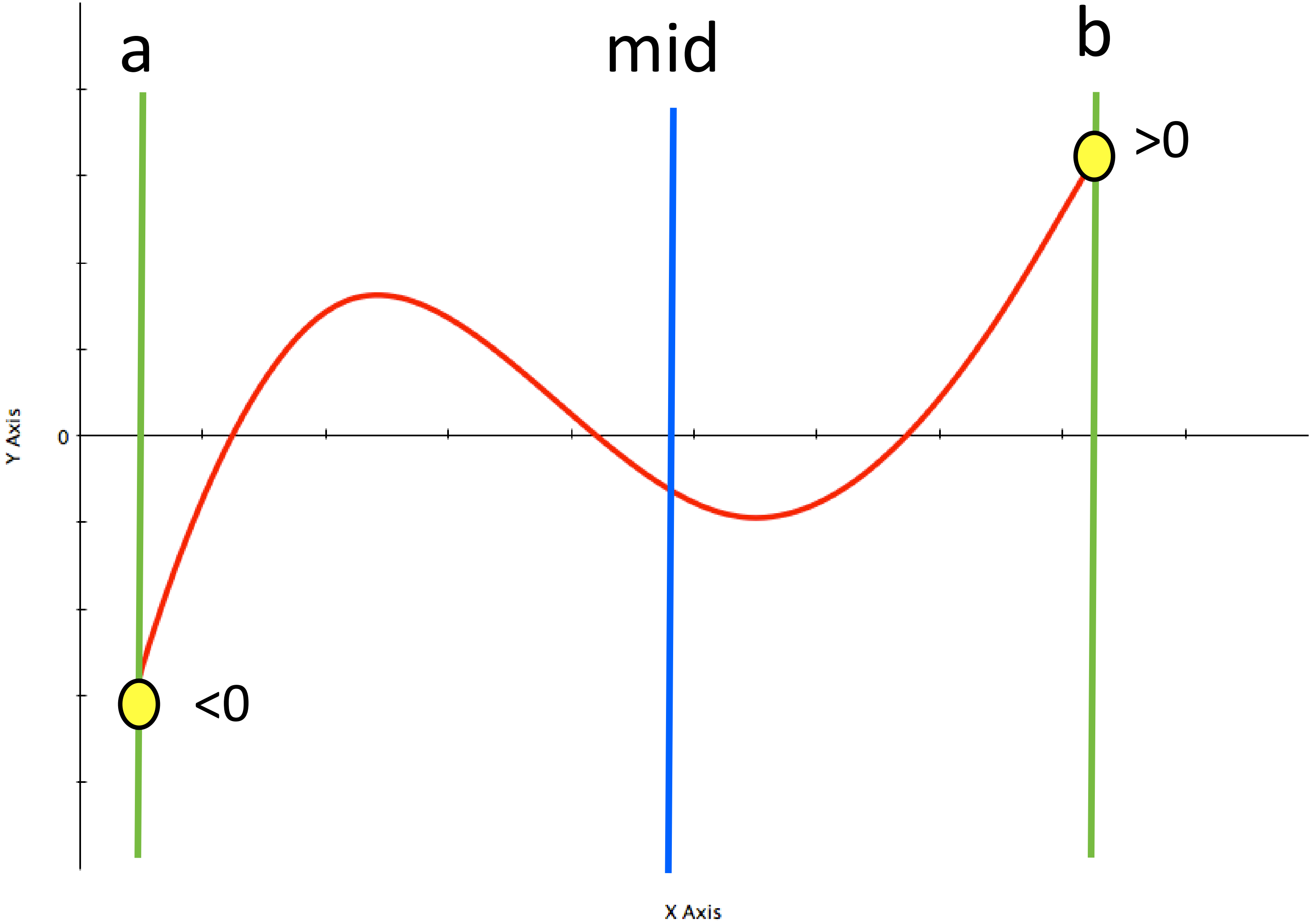
# Find a root

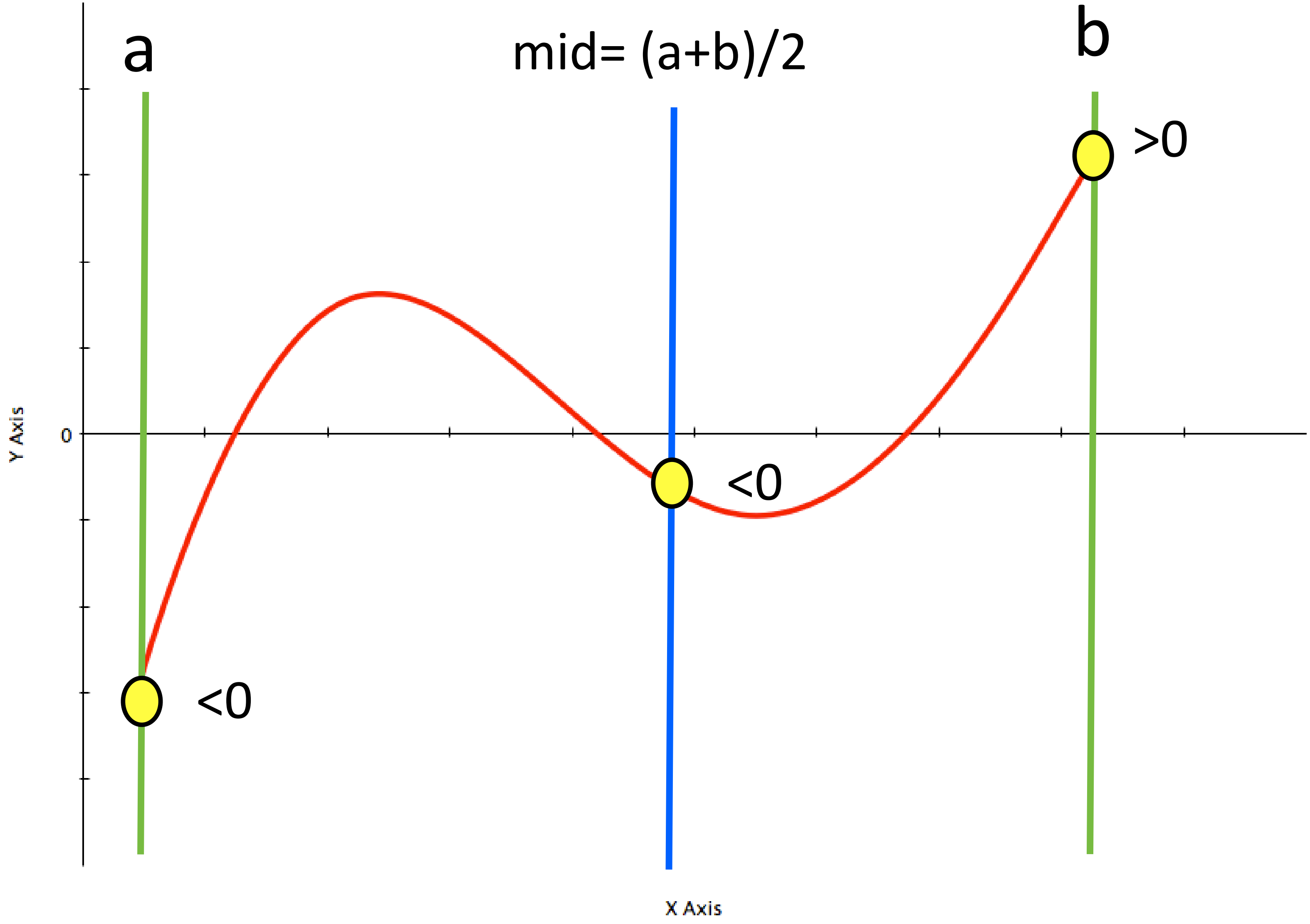


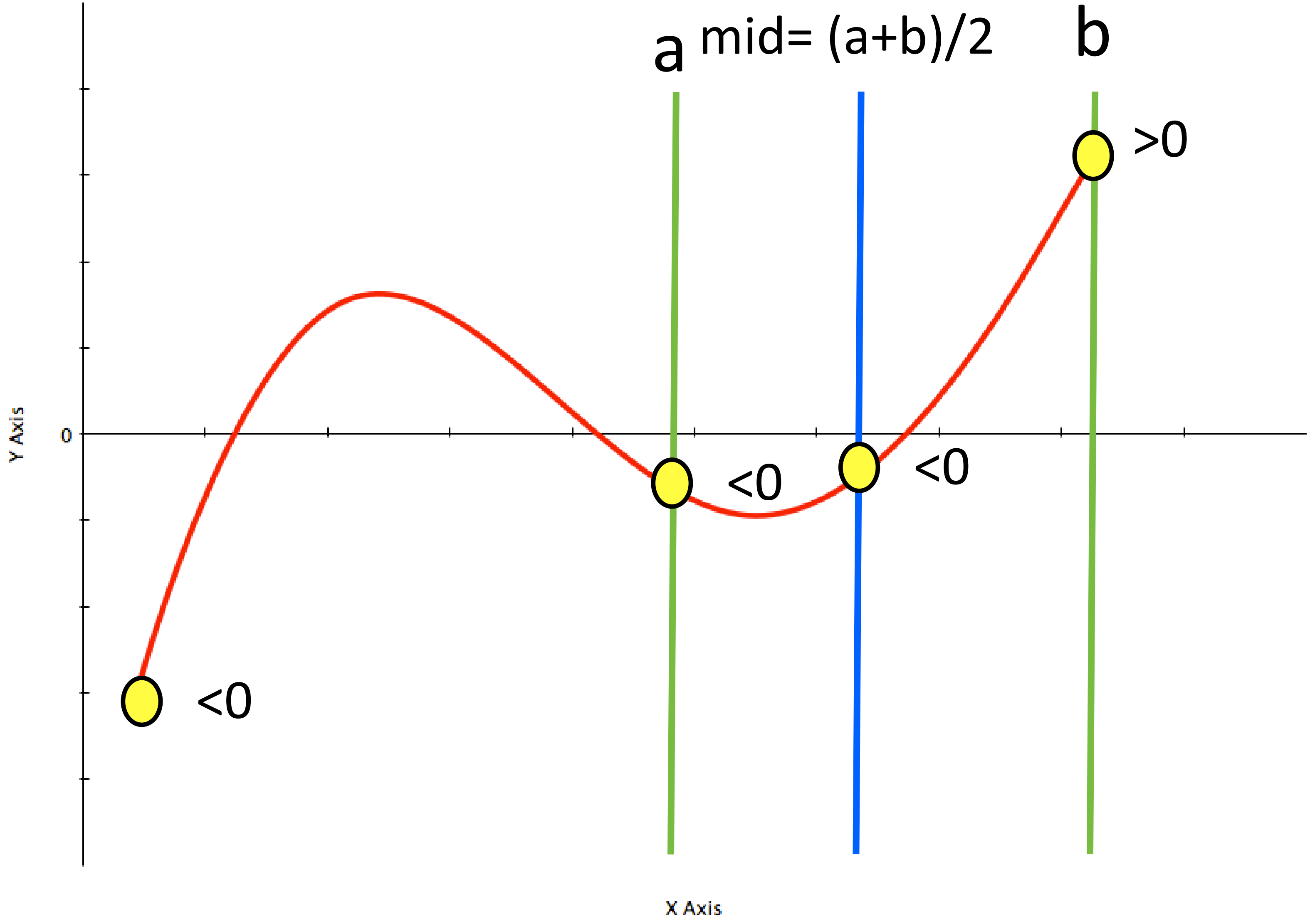

0
Y Axis
X Axis

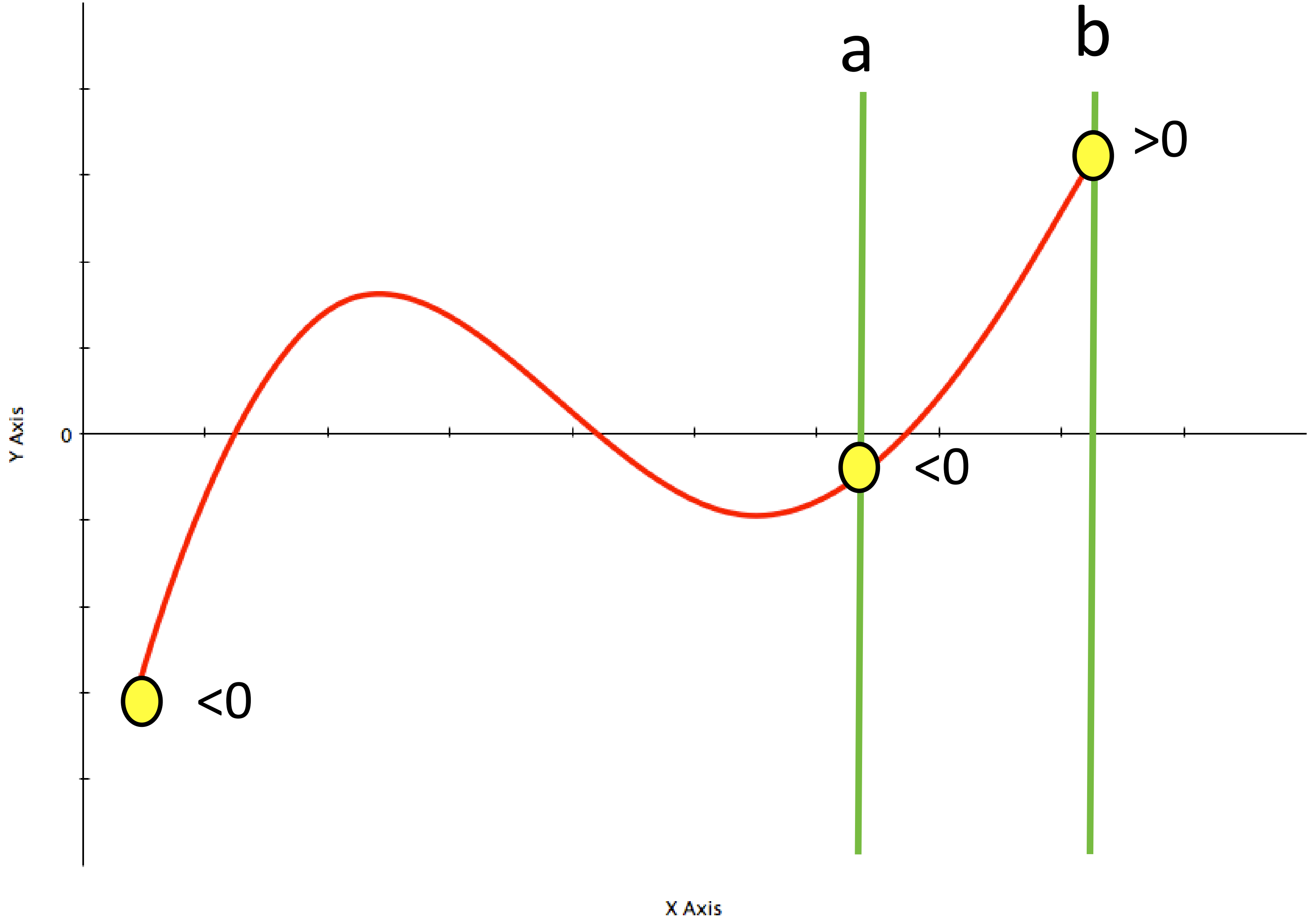


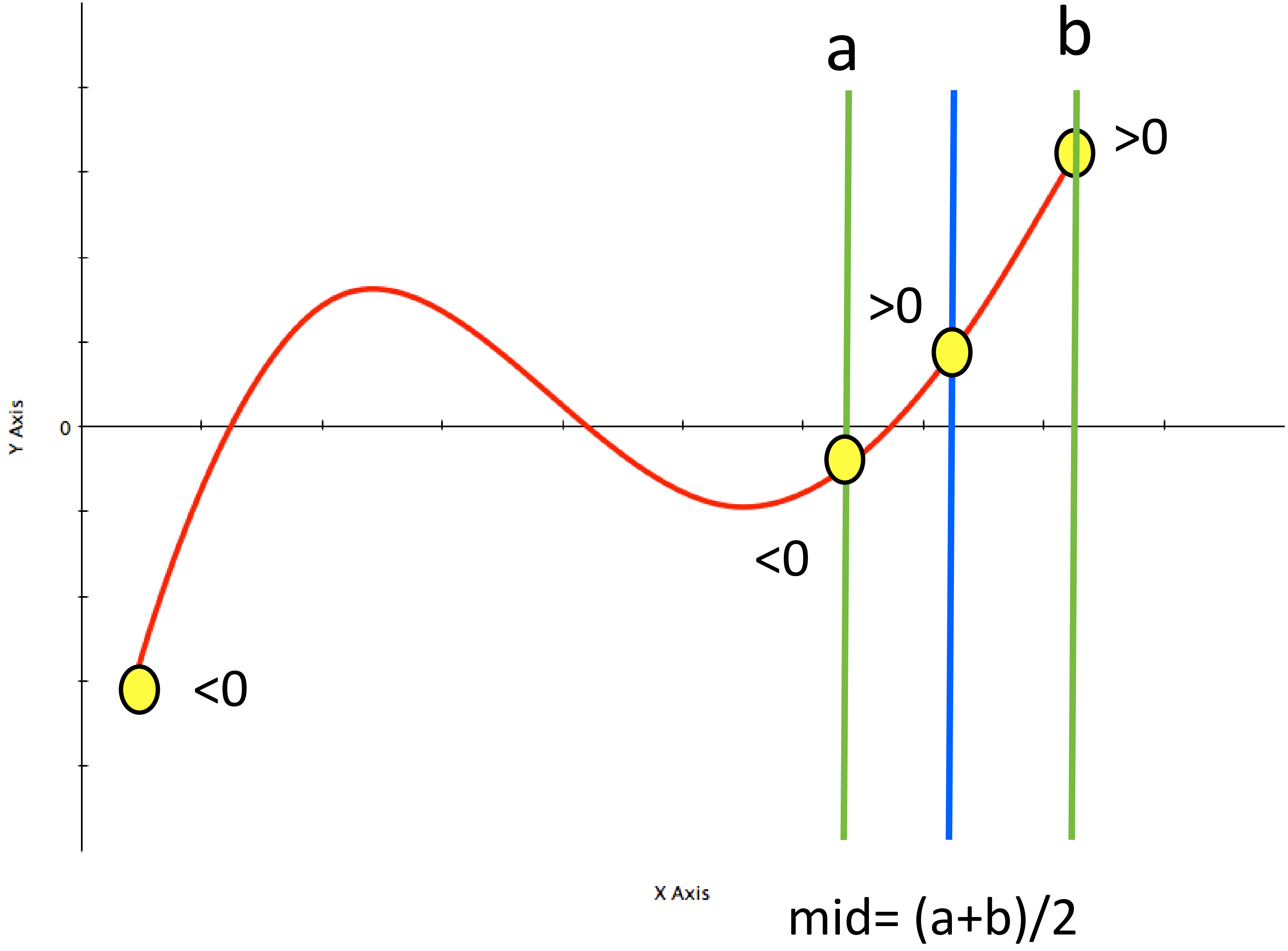
Assume that the function is continuous


a
mid
b
>0
<0
0
Y Axis
X Axis

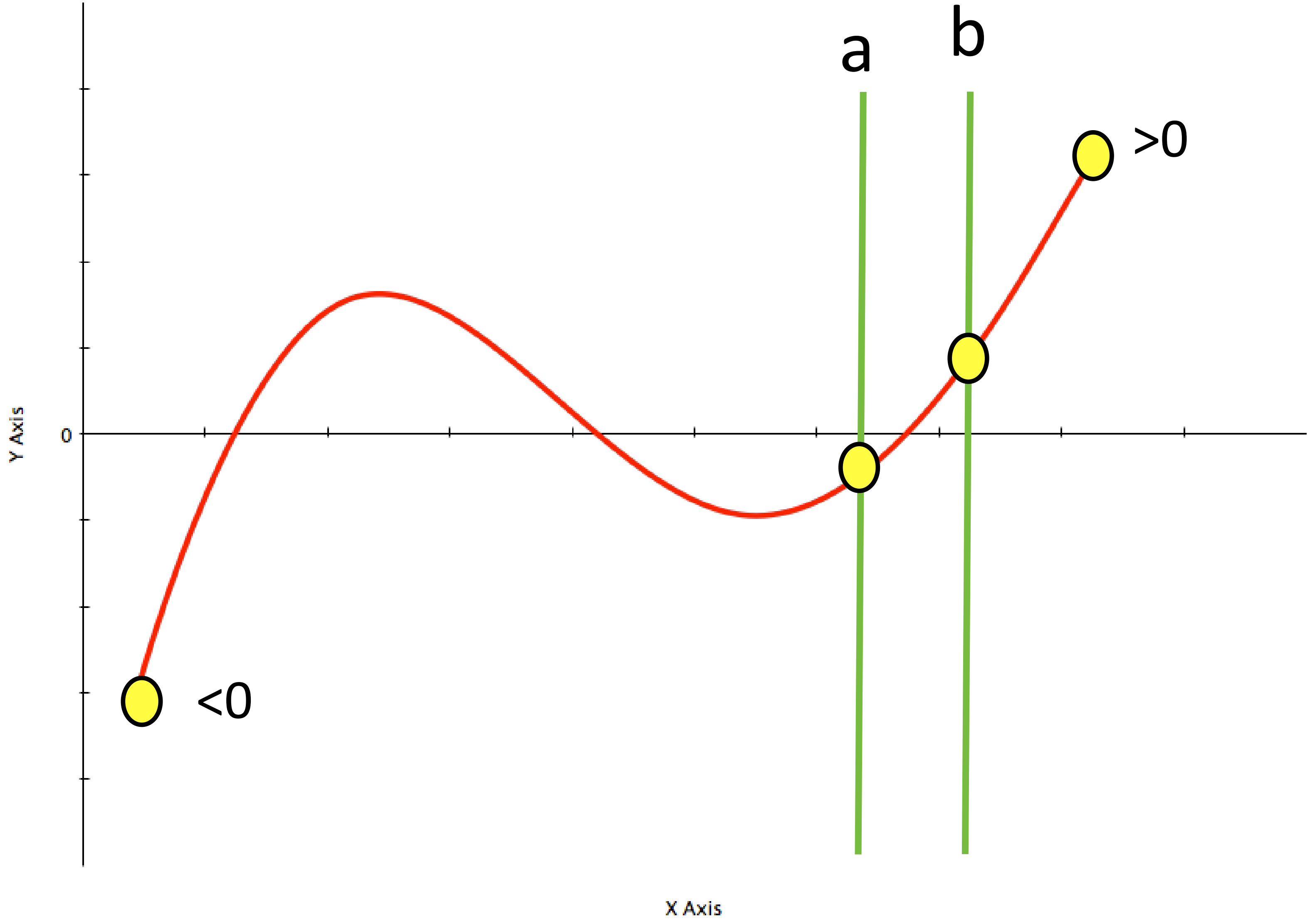

a
mid= (a+b)/2
b
>0
<0
<0
0
Y Axis
X Axis

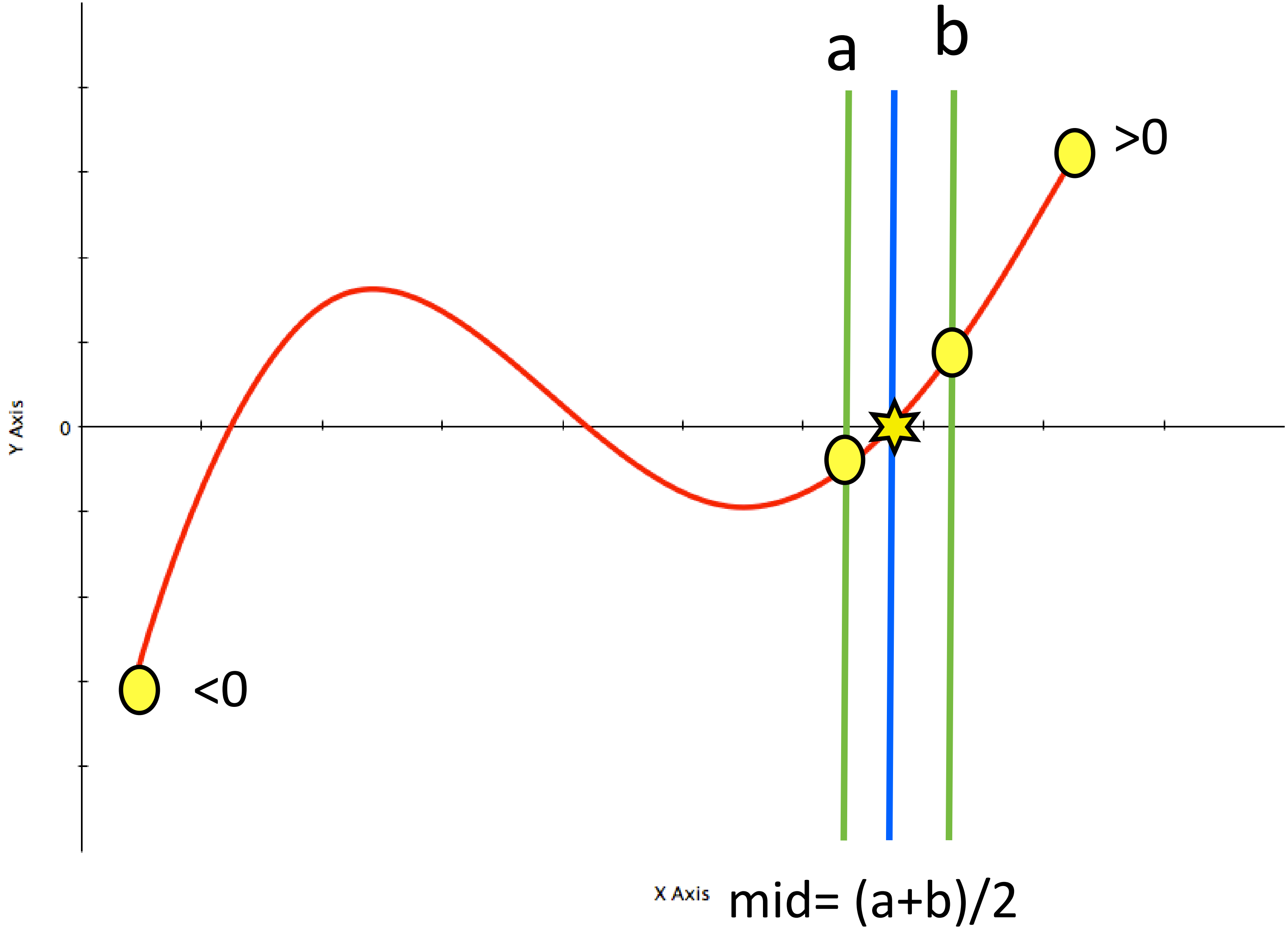

a
mid= (a+b)/2
b
>0
<0
<0
<0
0
Y Axis
X Axis


a
b
>0
<0
<0
0
Y Axis
X Axis



mid= (a+b)/2


a
b
>0
<0
0
Y Axis
X Axis



mid= (a+b)/2

- In a computer all real values are approximations

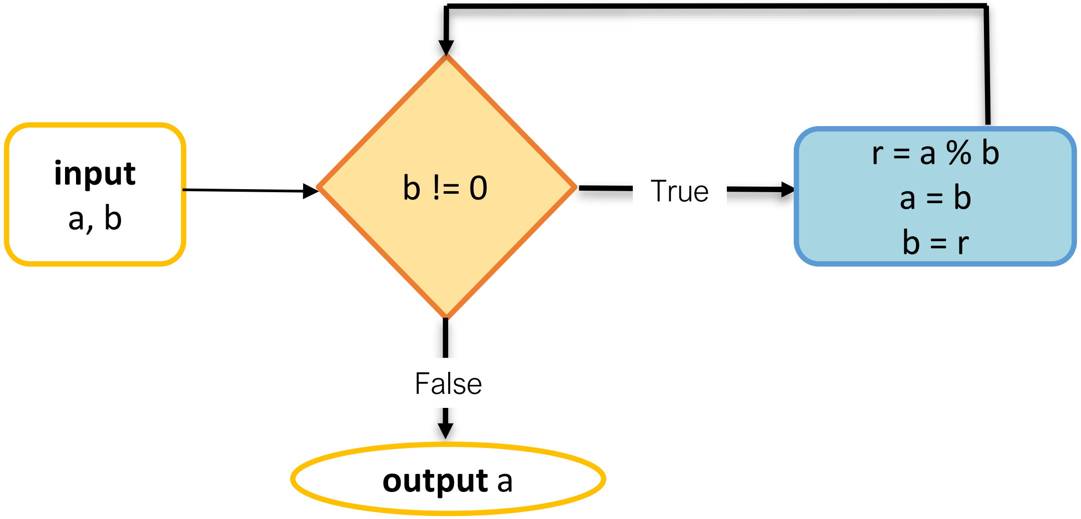
- Two real values may never be exactly the same,

- But can be close enough

# Overview

1. The Ordered Search Problem
2. Binary Search
3. Revisiting Euclid’s Algorithm

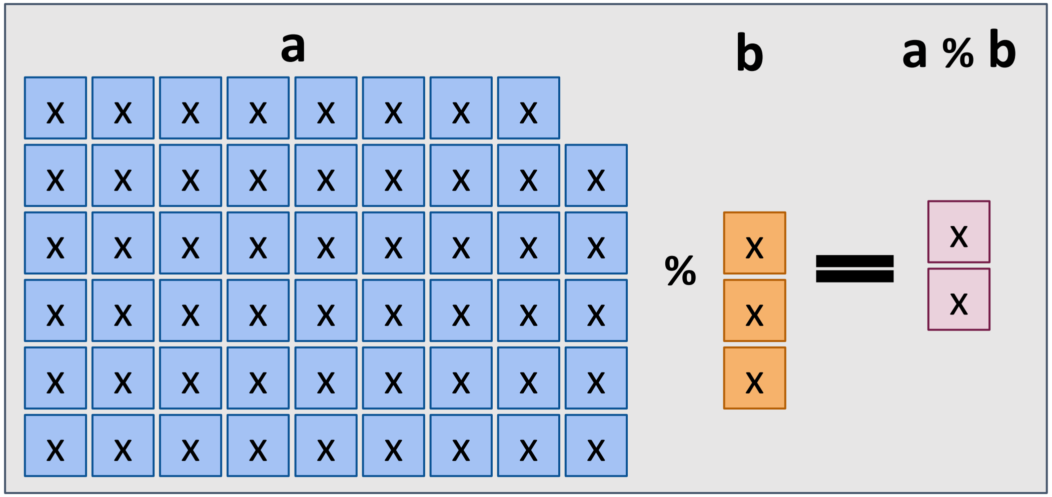
# Recall Euclid’s Algorithm

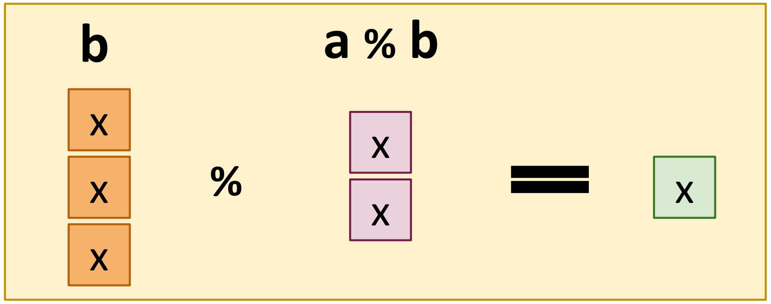


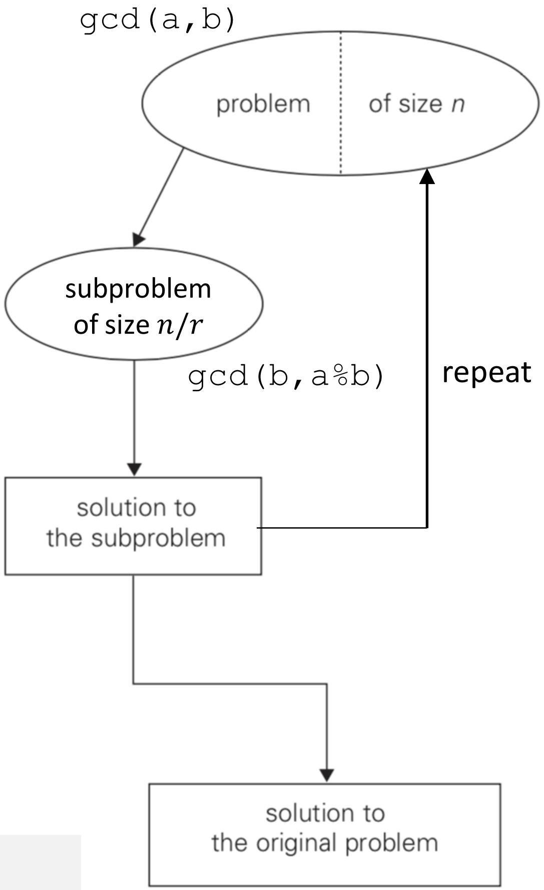
Eukleides of Alexandria
3xx BC – 2xx BC

```python
def gcd_euclid(a, b):
    """
    Input : integers a and b such that not a==b==0
    Output: the greatest common divisor of a and b
    """
    while b != 0:
        a, b = b, a % b
    return a
```

# Instance of decrease and conquer



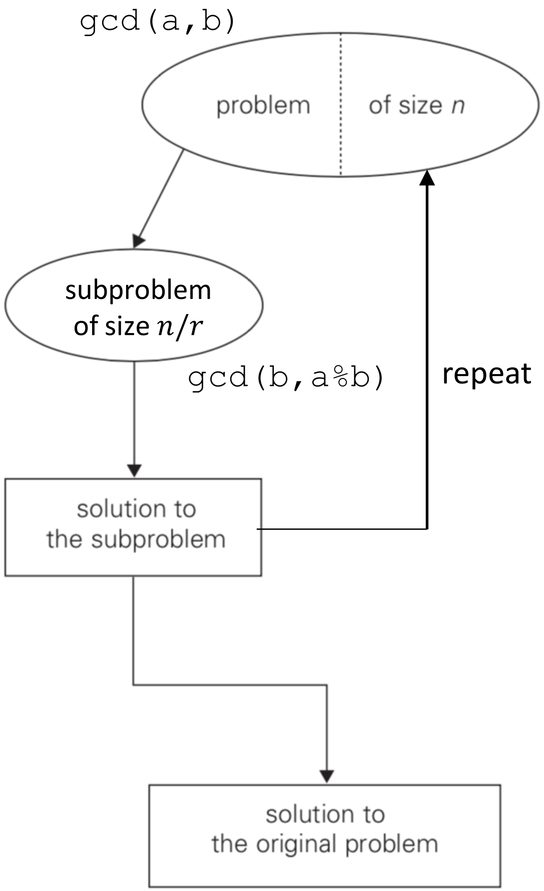




```python
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

# Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a and b such
       that not a==b==0
    O: gcd(a,b)      """
    while b != 0:
        a, b = b, a % b
    return a
```



gcd(a,b)

problem

of size $n$

subproblem of size $n/r$

gcd(b,a%b)

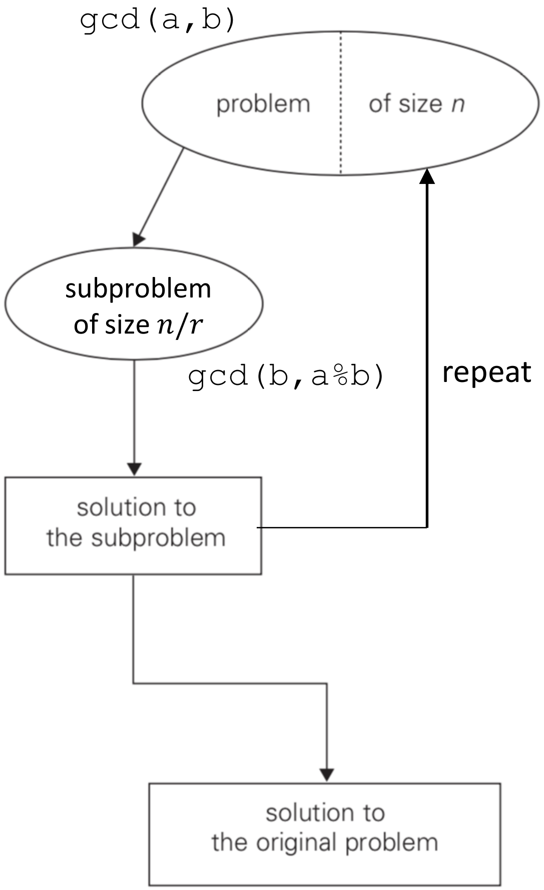
repeat

solution to the subproblem

solution to the original problem

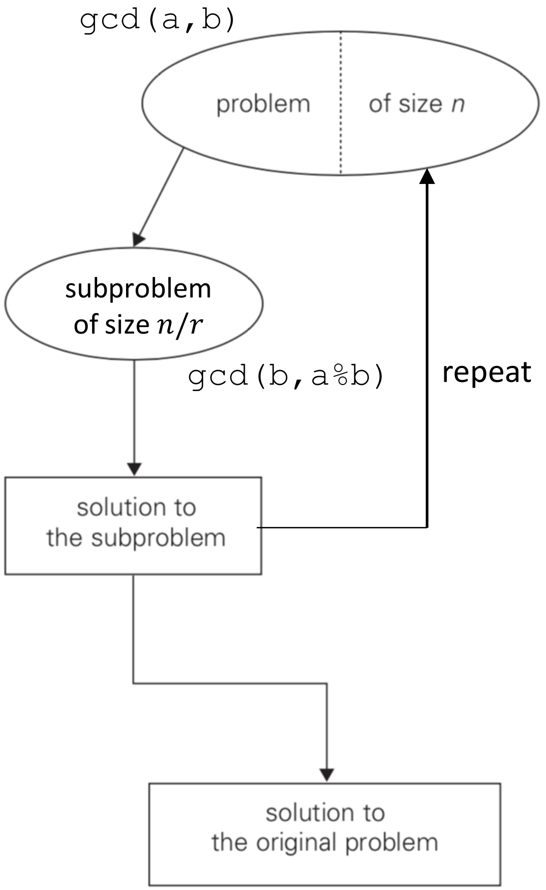
# Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a and b such
       that not a==b==0
    O: gcd(a,b)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        a, b = b, a % b
    return a
```



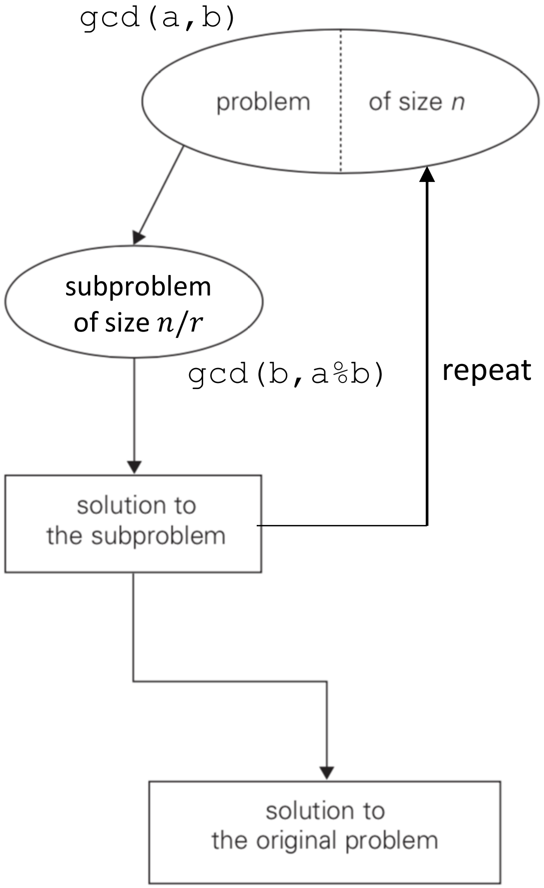
# Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a and b such
       that not a==b==0
    O: gcd(a,b)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        #I: gcd(a,b)==gcd(a0,b0)
        a, b = b, a % b
        #I: gcd(a,b)==gcd(a0,b0)
    return a
```



# Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a0 and b0 such
       that not a0==b0==0
    O: gcd(a0,b0)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        #I: gcd(a,b)==gcd(a0,b0)
        a, b = b, a % b
        #I: gcd(a,b)==gcd(a0,b0)
    #EXC: b==0
    return a
```



gcd(a,b)

problem | of size $n$

subproblem of size $n/r$

gcd(b,a%b)

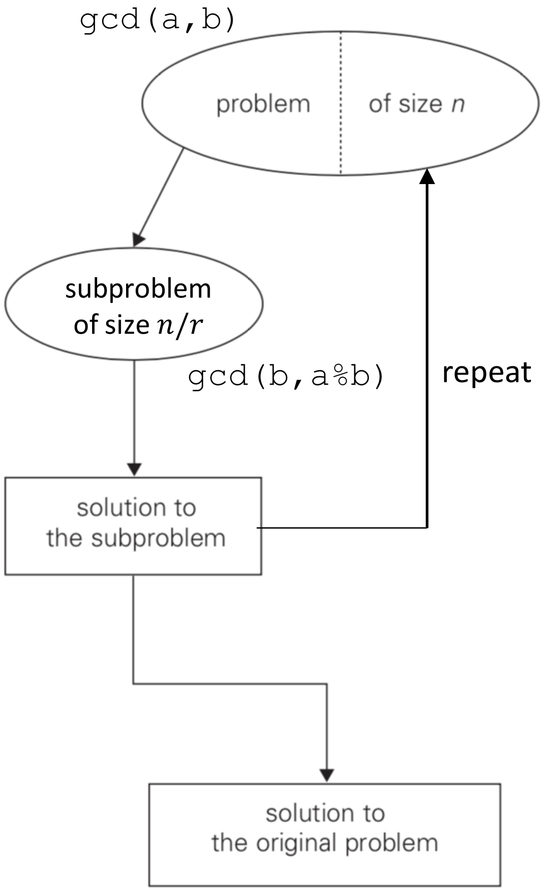
repeat

solution to the subproblem

solution to the original problem

# Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a and b such
       that not a==b==0
    O: gcd(a,b)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        #I: gcd(a,b)==gcd(a0,b0)
        a, b = b, a % b
        #I: gcd(a,b)==gcd(a0,b0)
    #EXC: b==0
    #POC: a==gcd(a,b)==gcd(a0,b0)
    return a
```

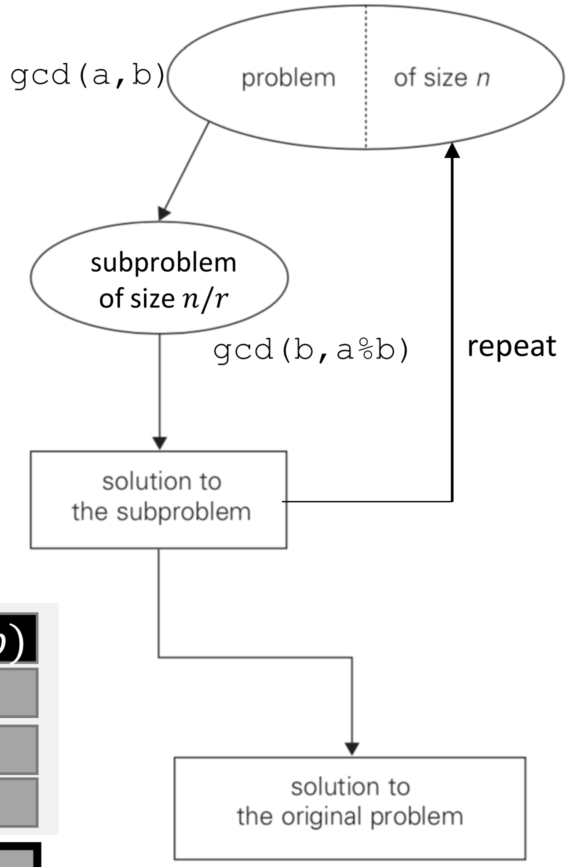


# Can we analyse computational complexity as for binary search?

Need to determine how many iterations we can have in worst case!

```
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

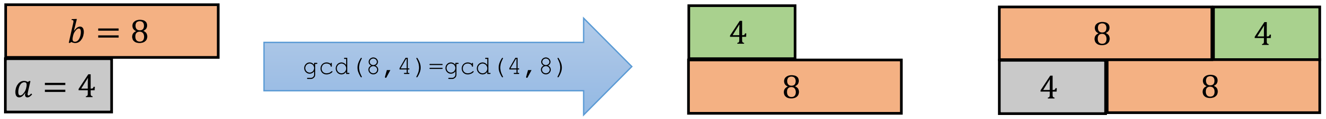
| | | |
|---|---|---|
| `def gcd_euclid(a, b):` | $n = \text{abs}(a) + \text{abs}(b)$ | |
| `while b != 0:` | $O(1)$ | ? |
| `a, b = b, a % b` | 0 | ? |
| `return a` | 0 | 0 |
| | $O(1)$ | ? |



# By what factor is problem decreased per iteration of Euclid’s Algorithm?

Example 1

No decrease in problem size!



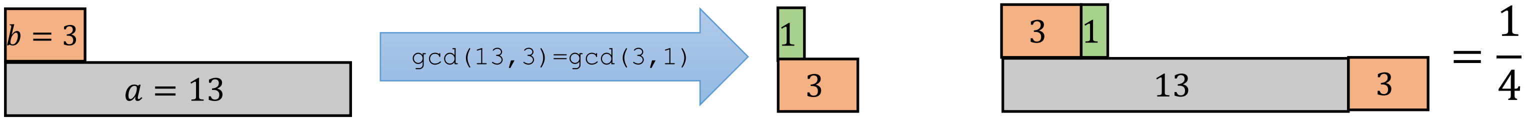
$b = 8$

$a = 4$

gcd(8,4)=gcd(4,8)

But this can only happen once in the beginning (afterwards $b > a\%b$ guarantees $a > b$)

Example 2



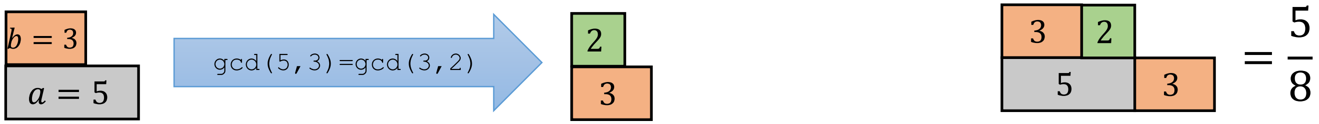
$b = 3$

$a = 13$

gcd(13,3)=gcd(3,1)

$= \frac{1}{4}$

Example 3



$b = 3$

$a = 5$

gcd(5,3)=gcd(3,2)

$= \frac{5}{8}$

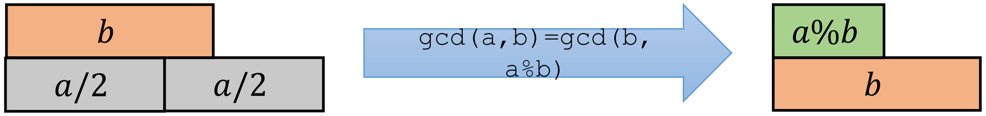
When $a > b$ there is decrease but at varying rate!

Do we need a fixed rate of decrease for logarithmic complexity?

No, just guarantee that reduction factor is always at least some $\alpha > 1$

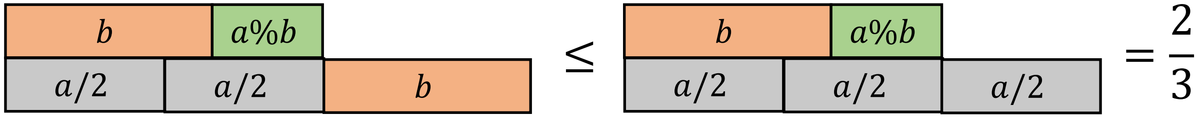
# First case: “large b”

Case $b \geq a/2$



# Relative size of decreased problem with large b

Case $b \geq a/2$



$b$ $a\%b$

$a/2$ $a/2$ $b$

$\leq$

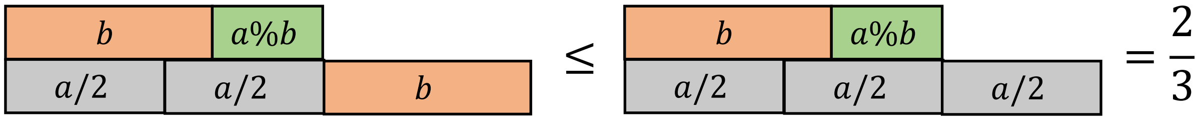
$b$ $a\%b$

$a/2$ $a/2$ $a/2$

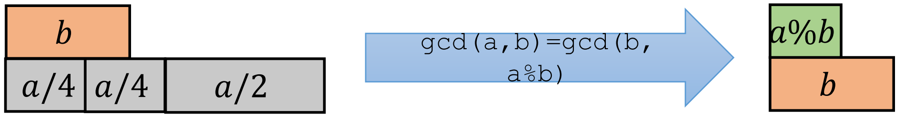
$= \frac{2}{3}$

# Second case: “small b”

Case $b \geq a/2$

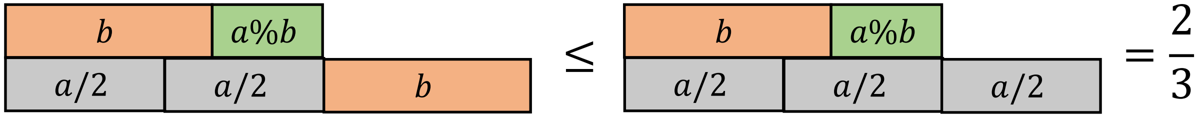


Case $a/2 > b \geq a/4$

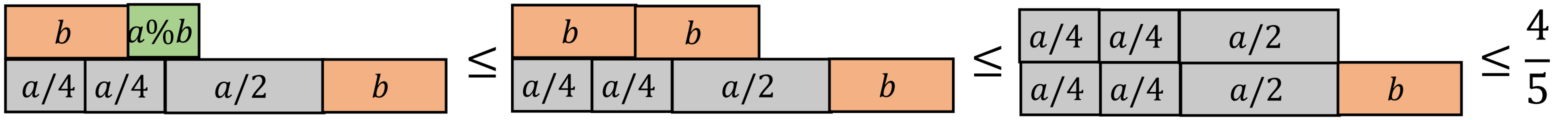


# Relative size of decreased problem in second case

Case $b \geq a/2$

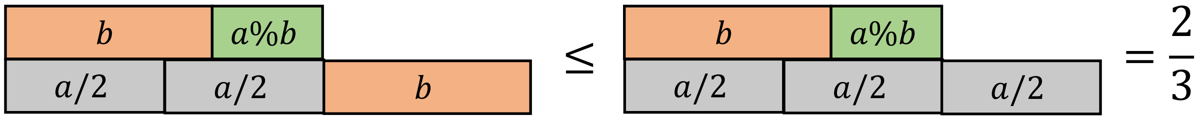


Case $a/2 > b \geq a/4$

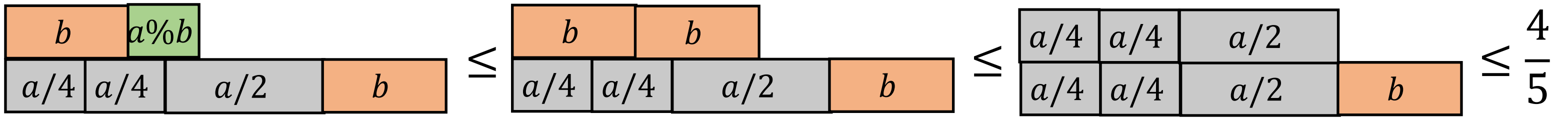


# Final case: “tiny b”

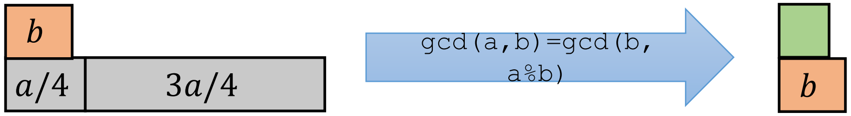
Case $b \geq a/2$



Case $a/2 > b \geq a/4$

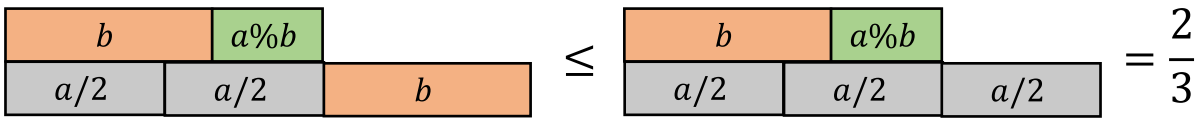


Case $a/4 > b$

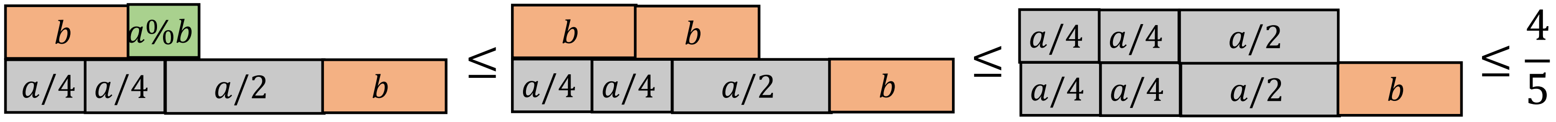


# Relative size of decreased problem in final case

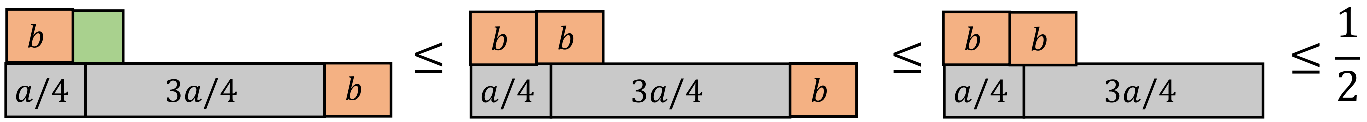
Case $b \geq a/2$



Case $a/2 > b \geq a/4$



Case $a/4 > b$

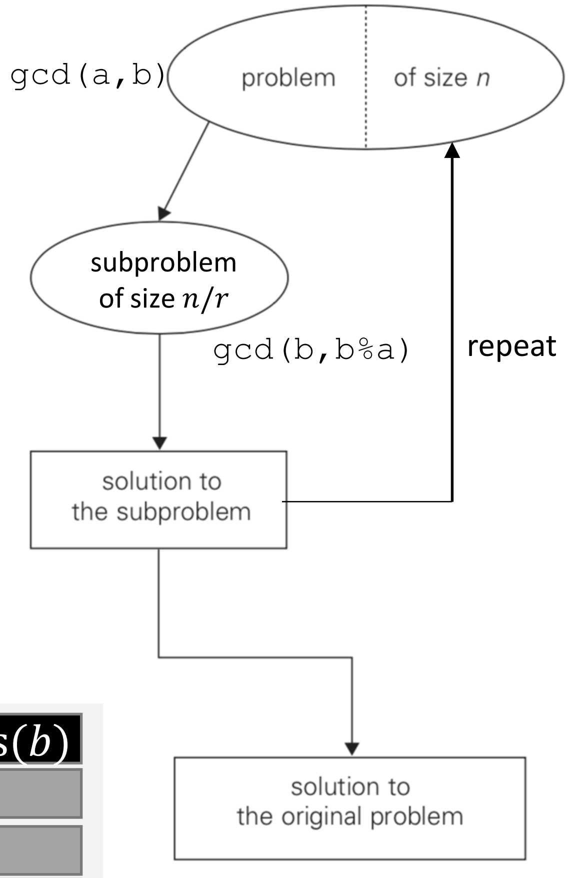


In all cases: problem is at least decreased by a rate r of 5/4

MONASH College

# Almost identical analysis as for binary search

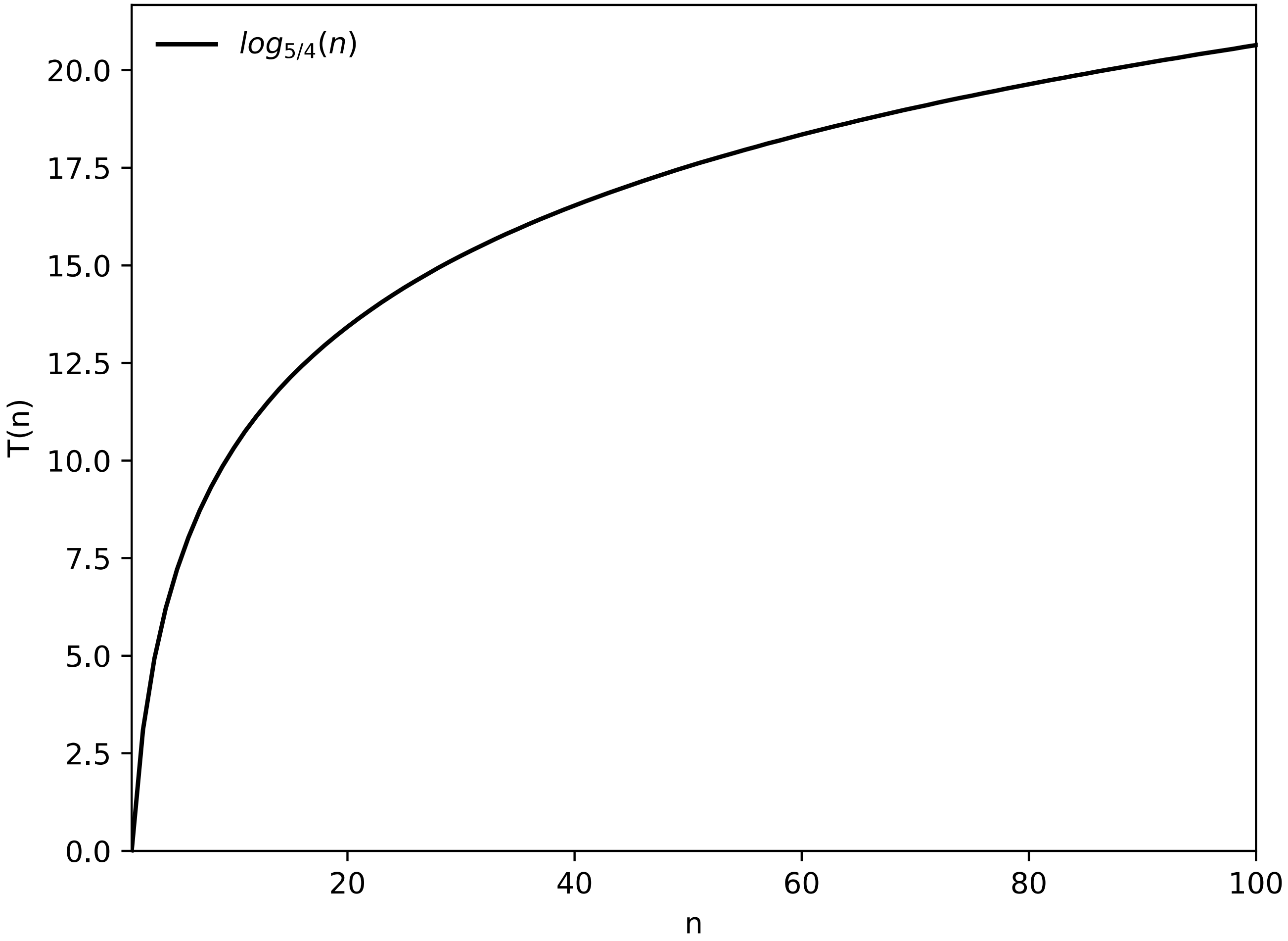
- Let $n_i = a_i + b_i$ be problem size after $i$ iterations of loop
- In the beginning: $n_0 = n$
- Per iteration size is reduced by at least: $n_i = \lceil n_{i-1}/r \rceil$, i.e., $n_i = \left\lceil n/r^i \right\rceil$
- After at most $k = \lceil \log_r n \rceil$ iterations: $n_k = \left\lceil n/r^{\log_r n} \right\rceil = 1$, i.e., $b_k = 0$ and $a_k = 1$
- So at most $O(\log_r n)$ loop iterations



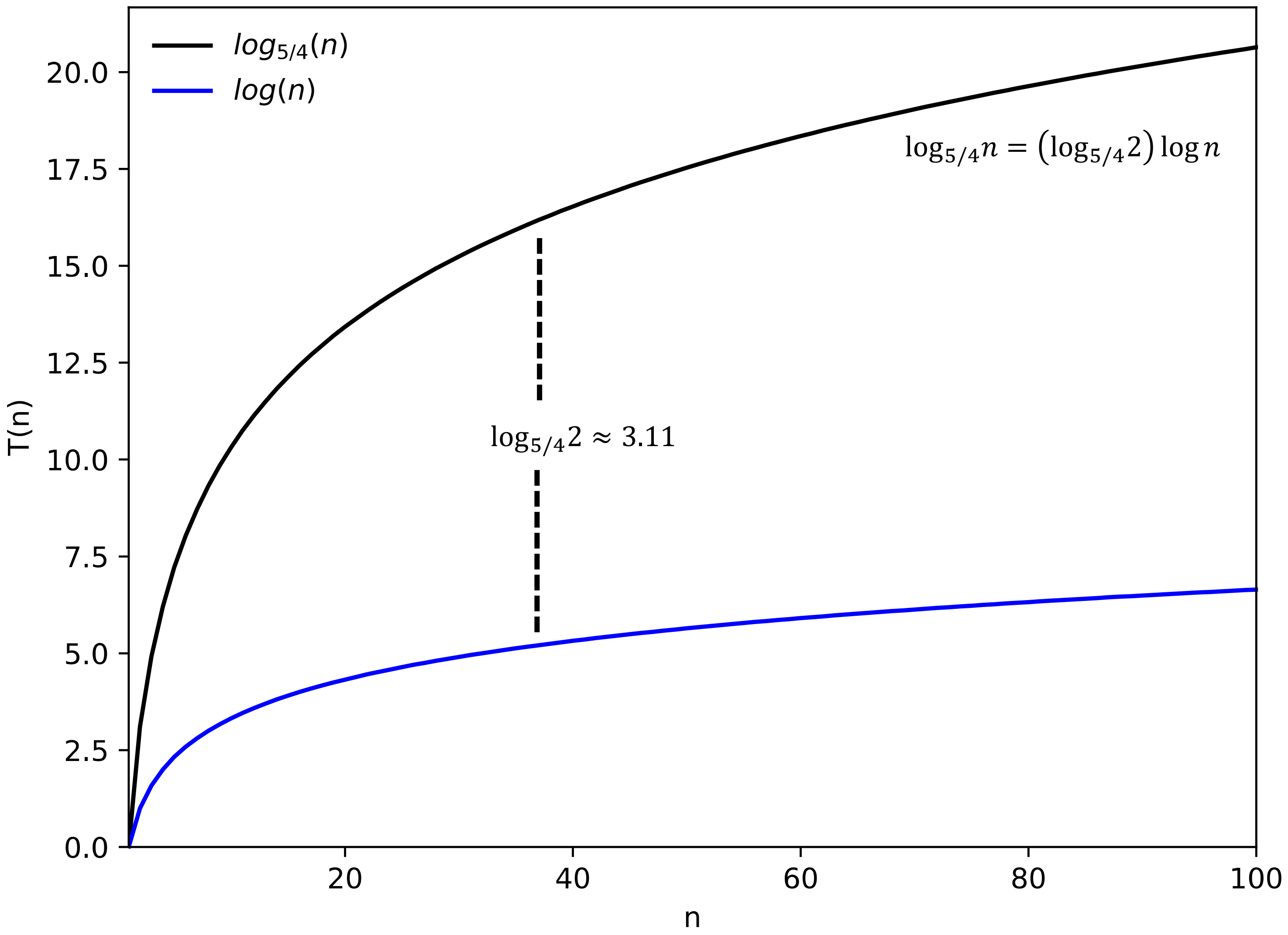
```
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

| | | |
|---|---|---|
| `def gcd_euclid(a, b):` | $n = \text{abs}(a) + \text{abs}(b)$ | |
| `while b != 0:` | $O(1)$ | ? |
| `a, b = b, a % b` | 0 | ? |
| `return a` | 0 | 0 |
| | $O(1)$ | ? |

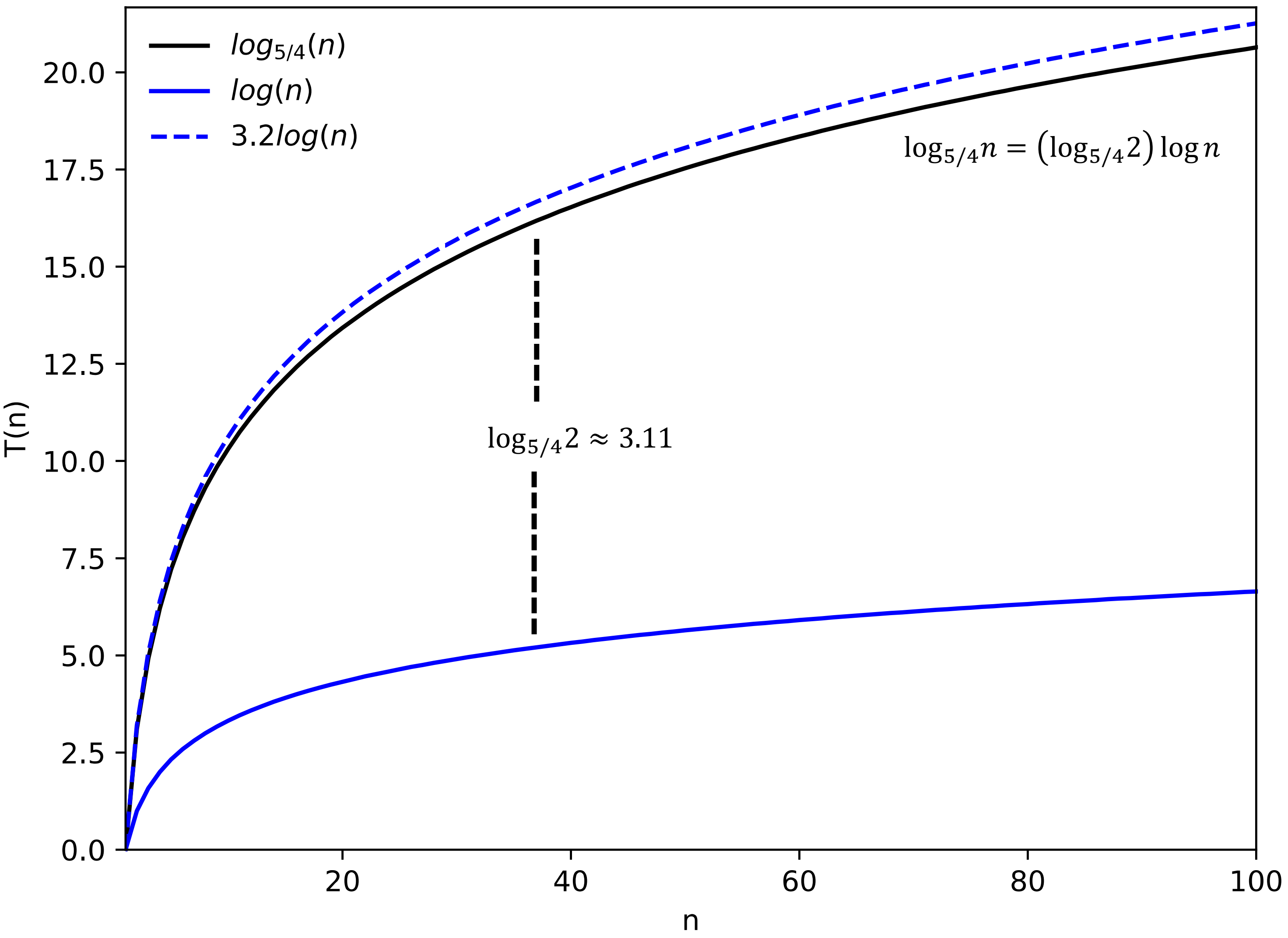
# What does this mean in terms of order of growth?



# Is order of growth log base 5/4 higher than log base 2?

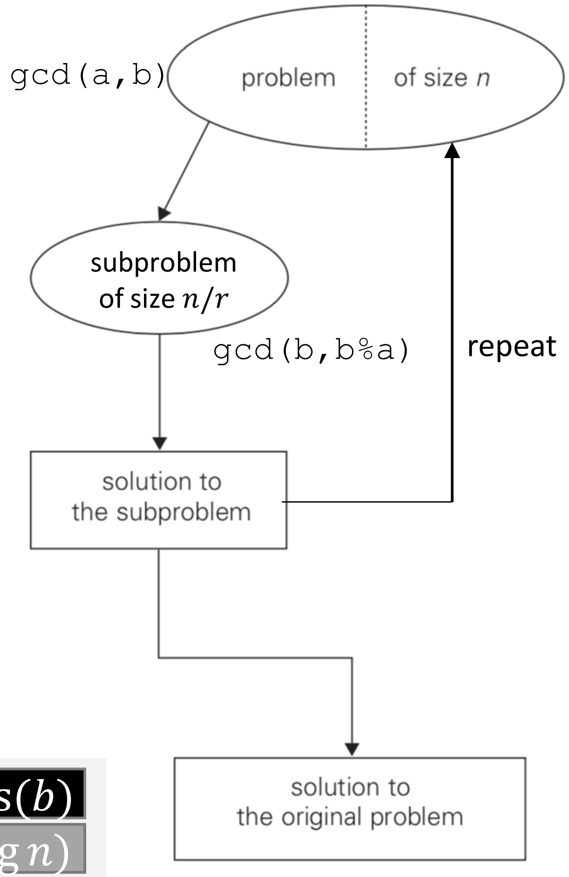


# No: $O(\log n) = O(\log_r n)$



# Almost identical analysis as for binary search

- Let $n_i = a_i + b_i$ be problem size after $i$ iterations of loop
- In the beginning: $n_0 = n$
- Per iteration size is reduced by at least: $n_i = \lceil n_{i-1}/r \rceil$, i.e., $n_i = \lceil n/r^i \rceil$
- After at most $k = \lceil \log_r n \rceil$ iterations: $n_k = \lceil n/r^{\log_r n} \rceil = 1$, i.e., $b_k = 0$ and $a_k = 1$
- So at most $O(\log_r n)$ loop iterations



```
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

| | | |
|---|---|---|
| `def gcd_euclid(a, b):` | $n = \text{abs}(a) + \text{abs}(b)$ | |
| `while b != 0:` | $O(1)$ | $O(\log n)$ |
| `a, b = b, a % b` | $0$ | $O(\log n)$ |
| `return a` | $0$ | $0$ |
| | $O(1)$ | $O(\log n)$ |

# Summary

Algorithmic paradigm: decrease-and-conquer

- decreasing problem size by at least some rate r>1 leads to trivial subproblems after logarithmically many reductions
- if not too much overhead: allows to replace linear complexity term by logarithmic term

Binary Search allows logarithmic time look-up of value in sorted sequence

Euclid’s Algorithm finds gcd in time logarithmically in sum of input abs(a) + abs(b)

# Coming Up

- More examples for algorithm analysis
- Divide and conquer