Analysis of Algorithms: Sorting algorithms (Selection sort and Quicksort)

Garrett Dancik, PhD

Fall 2021

Course Notes: https://gdancik.github.io

What do we mean by Sorting?

- One of the most common operations in computer science is to sort data numerically or alphabetically
- We have seen previously that sorted data can be searched much more efficiently than unsorted data. Why?
- In addition, for presentation purposes, elements such as names, states, ages, GPAs, etc, are often displayed in sorted order (numeric data may be sorted from low to high or high to low; when we say that numeric data is sorted we will mean low to high)

11	21	18	3	15	19

- The list above in sorted order is: 3, 11, 15, 18, 19, and 21

Selection sort

- Find the maximum element in the list (all *n* elements)
  - Swap this maximum element with the last element in the list
- Find the maximum element in the list (first *n* – 1 elements)
  - Swap this maximum element with the second to last element in the list
- Find the maximum element in the list (first *n* – 2 elements)
  - Swap this maximum element with the third to last element in the list
- This process repeats until we are down to the first element. This is the minimum element, which is now the first element in the list

11	21	18	3	15	19

Selection sort (example)

- We search all $n = 6$ elements for the maximum, and swap this maximum element with the last one in the list (the 6^{th} one)

The max is 21 → swap this with the last element

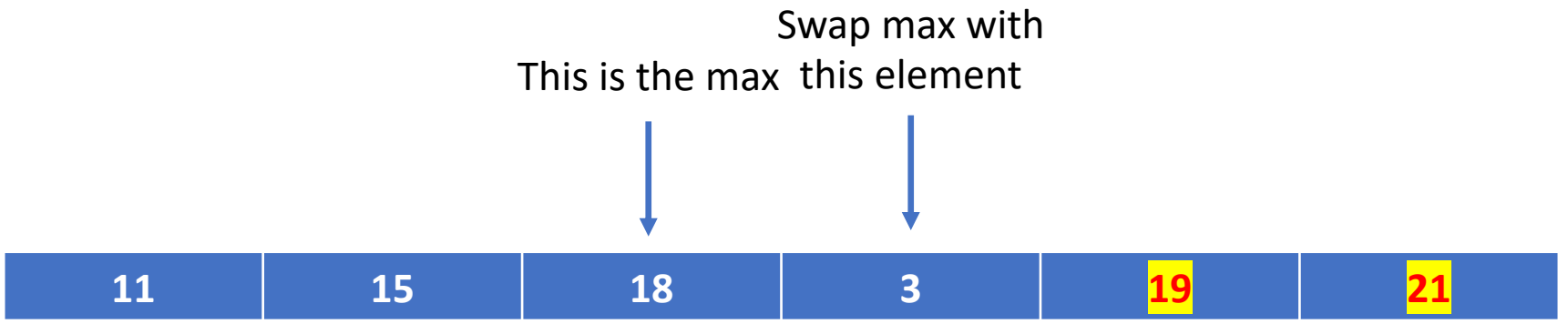
Selection sort (example)

- We search the first *n-1* = 5 elements for the maximum, and swap this maximum element with the 5th one (or the second to last one)

The max is 19 → swap this with the 5th element

Selection sort (example)

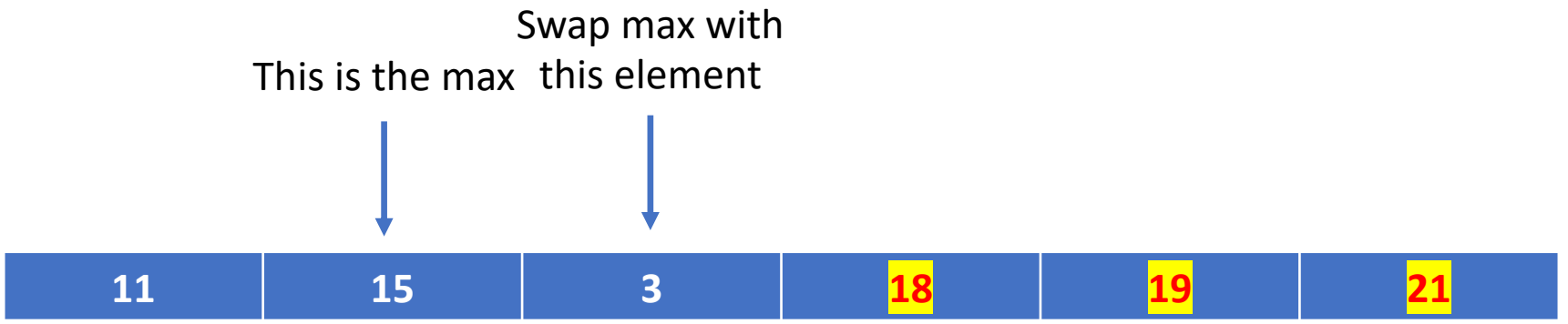
- We search the first *n-2* = 4 elements for the maximum, and swap this maximum element with the 4th one



The max is 18 → swap this with the 4th element

Selection sort (example)

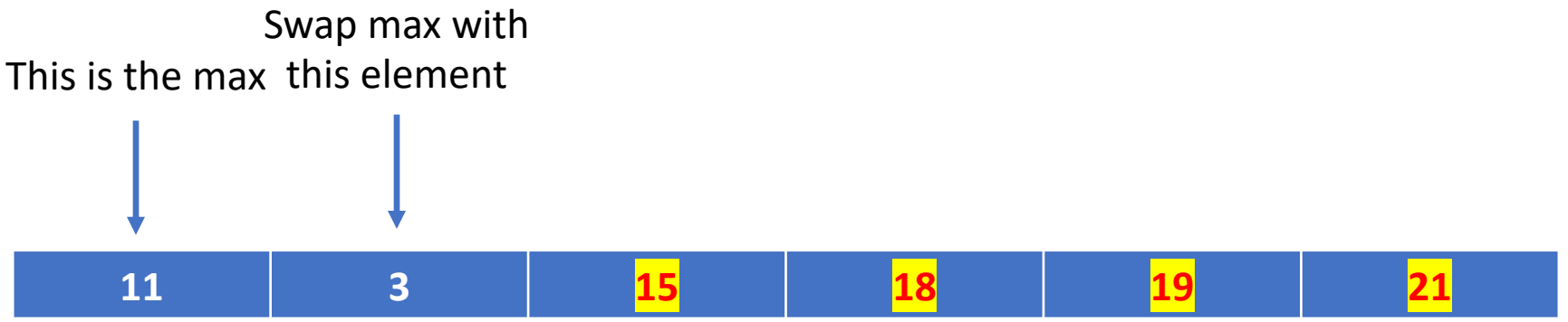
- We search the first n – 3 = 3 elements for the maximum, and swap this maximum element with the 3rd one



The max is 15 → swap this with the 3rd element

Selection sort (example)

- We search the first n – 4 = 2 elements for the maximum, and swap the maximum element with the 2nd one



The max is 11 → swap this with the 2nd element

Selection sort (example)

- Once we have only 1 element left (we are finding the max of just the 1st element), then we are done. The list is now sorted.

Selection sort algorithm

- *end* = n – 1
- while *end* > 0 :
  - Set *max_index* to the index of the maximum element between *values[0]* through *values[end]*
  - Swap *values[max_index]* and *values[end]*
  - Set *end* = *end* - 1

Selection sort algorithm

- *end* = n – 1
- while *end* > 0 :
  - Set *max_index* to the index of the maximum element between *values[0]* through *values[end]*
  - Swap *values[max_index]* and *values[end]*
  - Set *end* = *end* - 1

    - Set *max_index* to 0
    - Set *i* to 0
    - While *i* <= end:
      - If values[i] > values[max_index]*:*
        - set *max_index* to *i*
      - set *i* = *i +1*

Selection sort algorithm

- *end* = n – 1
- while *end* > 0 :
  - Set *max_index* to 0
  - Set *i* to 0
  - While *i* <= end:
    - If values[*i*] > values[*max_index*]*:*
      - set *max_index* to *I*
    - Set *i* = *i* + 1
  - Swap *values[max_index]* and *values[end]*
  - Set *end* = *end* - 1

Executed *n* – 1 times (while loop)

Executed up to *n* – 1 times each time (while loop)

This suggests an order of magnitude of n^2

Selection sort algorithm

- *end* = n – 1
- while *end* > 0 :
  - Set *max_index* to 0
  - Set *i* to 0
  - While *i* <= end:
    - If values[*i*] > values[*max_index*]*:*
      - set *max_index* to *I*
    - Set *i* = *i* + 1
  - Swap *values[max_index]* and *values[end]*
  - Set *end* = *end* - 1

Assume that *n* = 4

end	# iterations of inner while loop
3	4
2	3
1	2
0	-

In general, for a list of size *n*, the total number of inner loop iterations is:

2 + 3 + 4 + …. + *n*

This is $n(n+1)/2 - 1$, which has an order of magnitude of n^2.

Quicksort algorithm

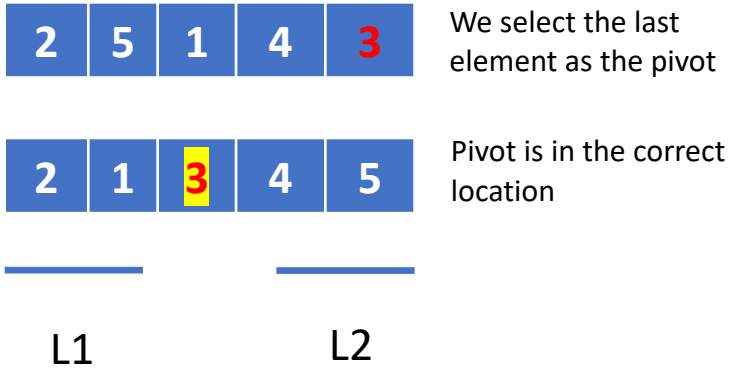
- Quicksort(arr, low, high) :
  - While low < high :
    - pi = partition (arr, low, high)
    - Quicksort(arr, low, pi – 1)
    - Quicksort(arr, pi + 1, high)

Call Quicksort on L1

Call Quicksort on L2

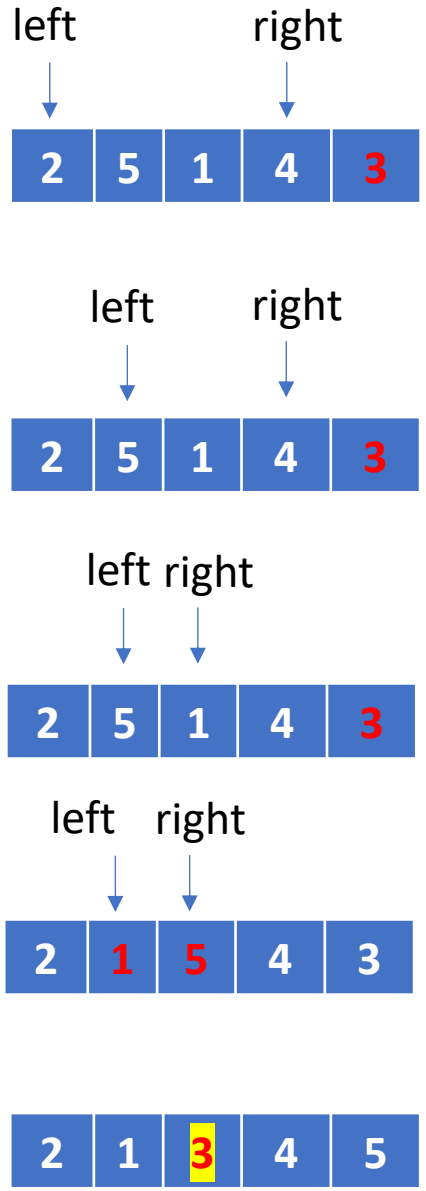
Partition step:

- select a *pivot*
- move *pivot to* correct location
  - all elements less than pivot are moved to its left
  - all elements greater than pivot are moved to its right
- Return partition index

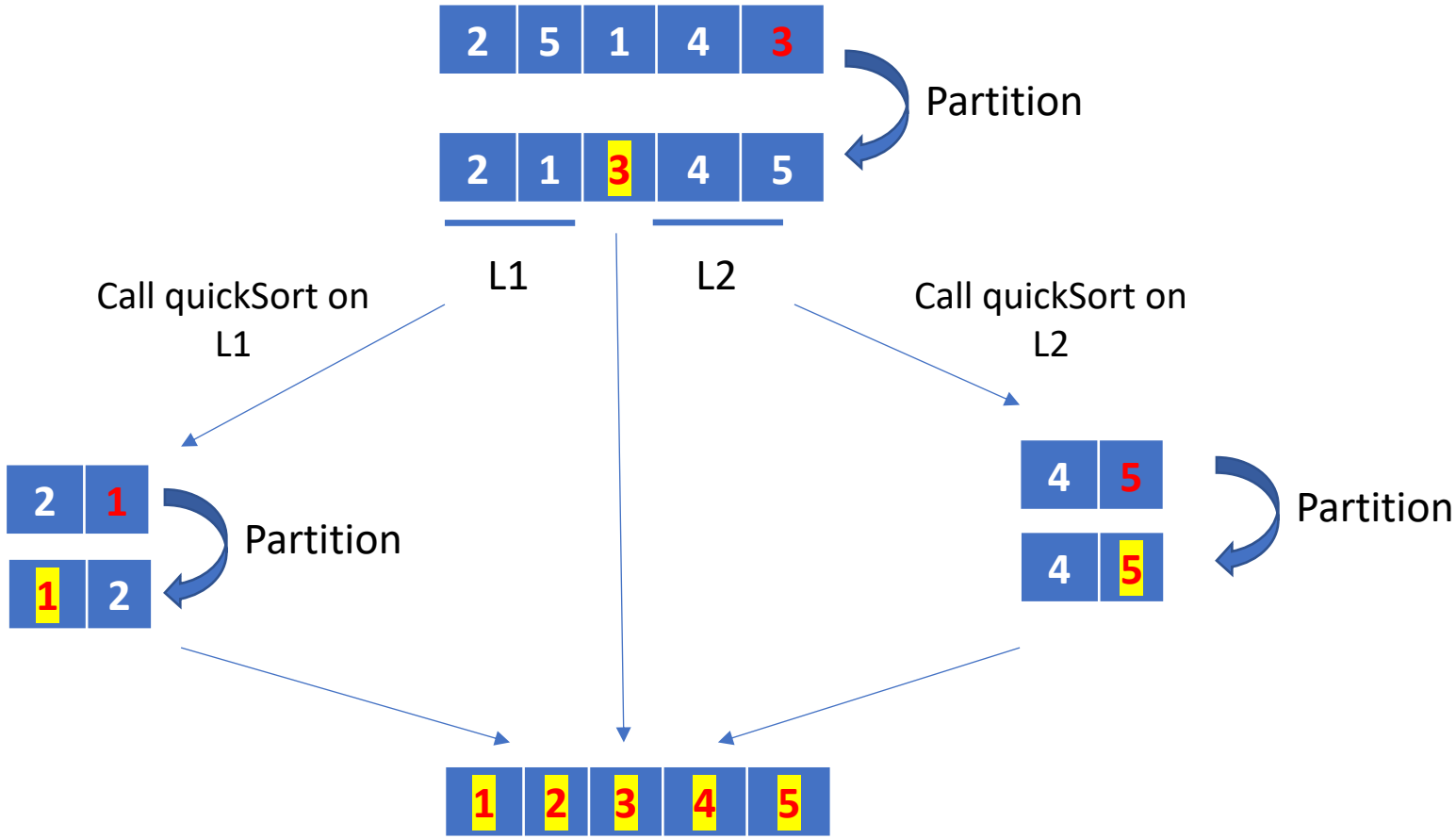


Partition algorithm

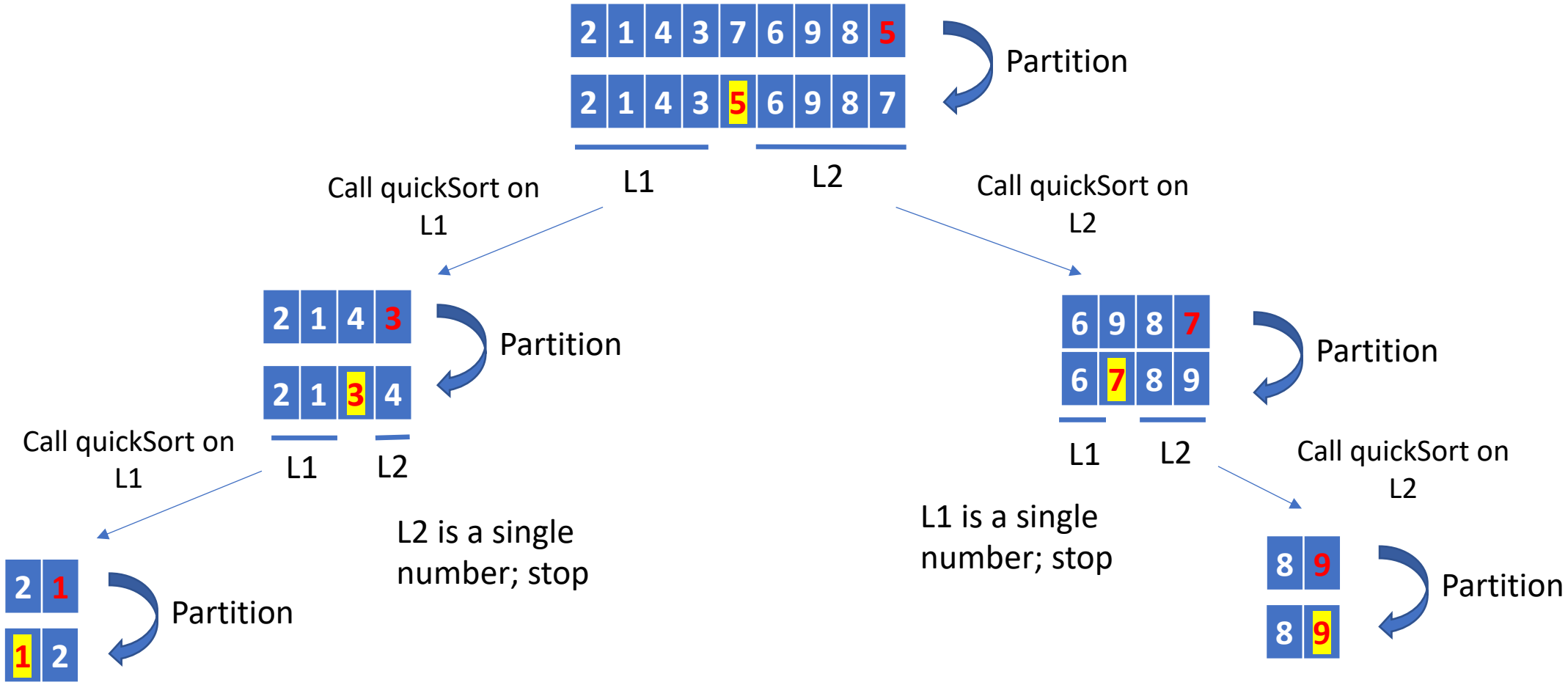
- Inputs:
  - *arr* (the list/array)
  - *low* (index of lower element),
  - *high* (index of last element, which will be the pivot)
- Set *left* = *low*
- Set *pivot* = *arr*[*high*]
- Set *right* = *high* – 1
- While *left* <= *right*:
  - Increase *left* by 1 until *arr*[*left*] > *pivot* (or left > right)
  - Decrease *right* by 1 until *arr*[*right*] <= *pivot* (or left > right)
  - If *left* < *right*, swap *arr*[*left*] and arr[*right*]
  - Increase *left* by 1
  - Decrease right by 1
- Swap arr[left] and arr[high]
- Return left



Quicksort example

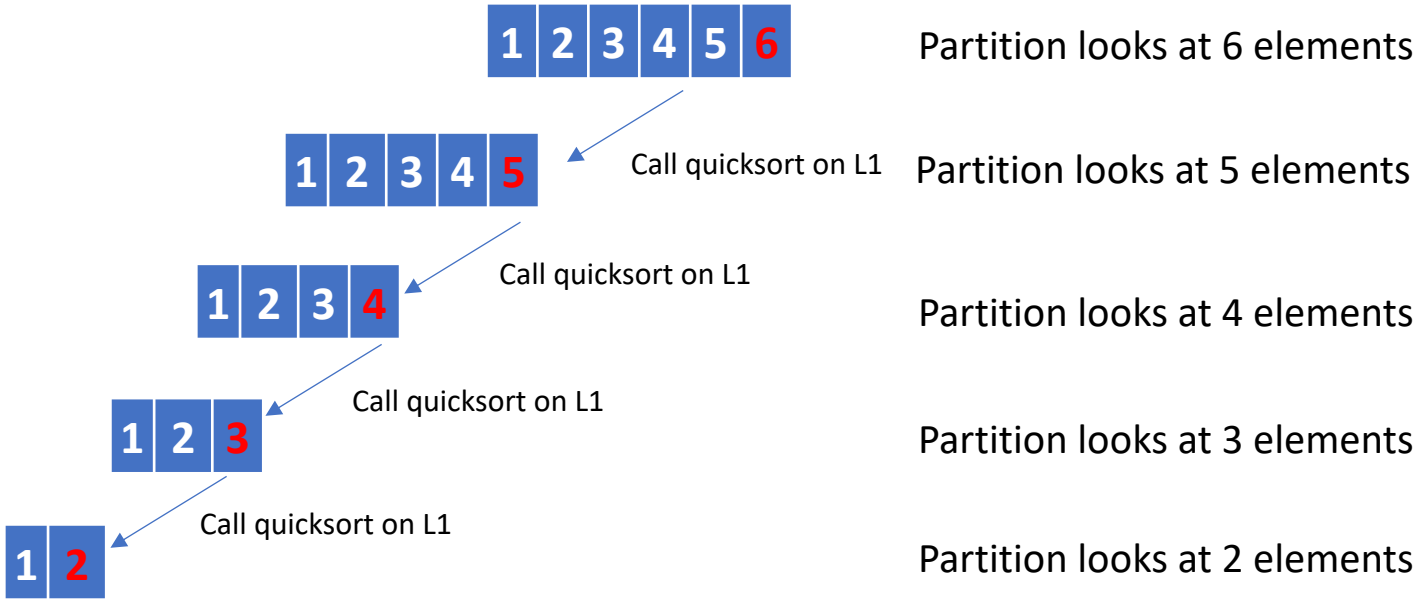


Quicksort example



Quicksort running times

- The *worst case* occurs if the original data is sorted, then the partition will keep the pivot in the last element and we will call quicksort on = a L1 = list containing all elements but the last one; L2 has no elements. The running time is $\theta(n^2)$
- Also see https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

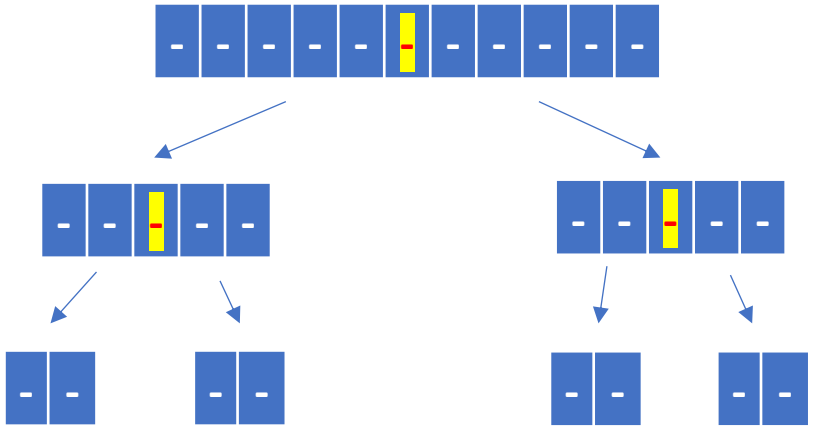


The total number of elements we look at is: 2 + 3 + …. + n

which is $\theta(n^2)$

Quicksort running times

- The *best case* occurs when the partitions are evenly balanced. In this case the number of sub-list pairs that get sorted is $\log n$. The running time is $\theta(n \log n)$.
- Also see https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort



Partition looks at 11 elements

Partition looks at 10 elements (~ 11 elements)

Partition looks at 8 elements (~ 11 elements)

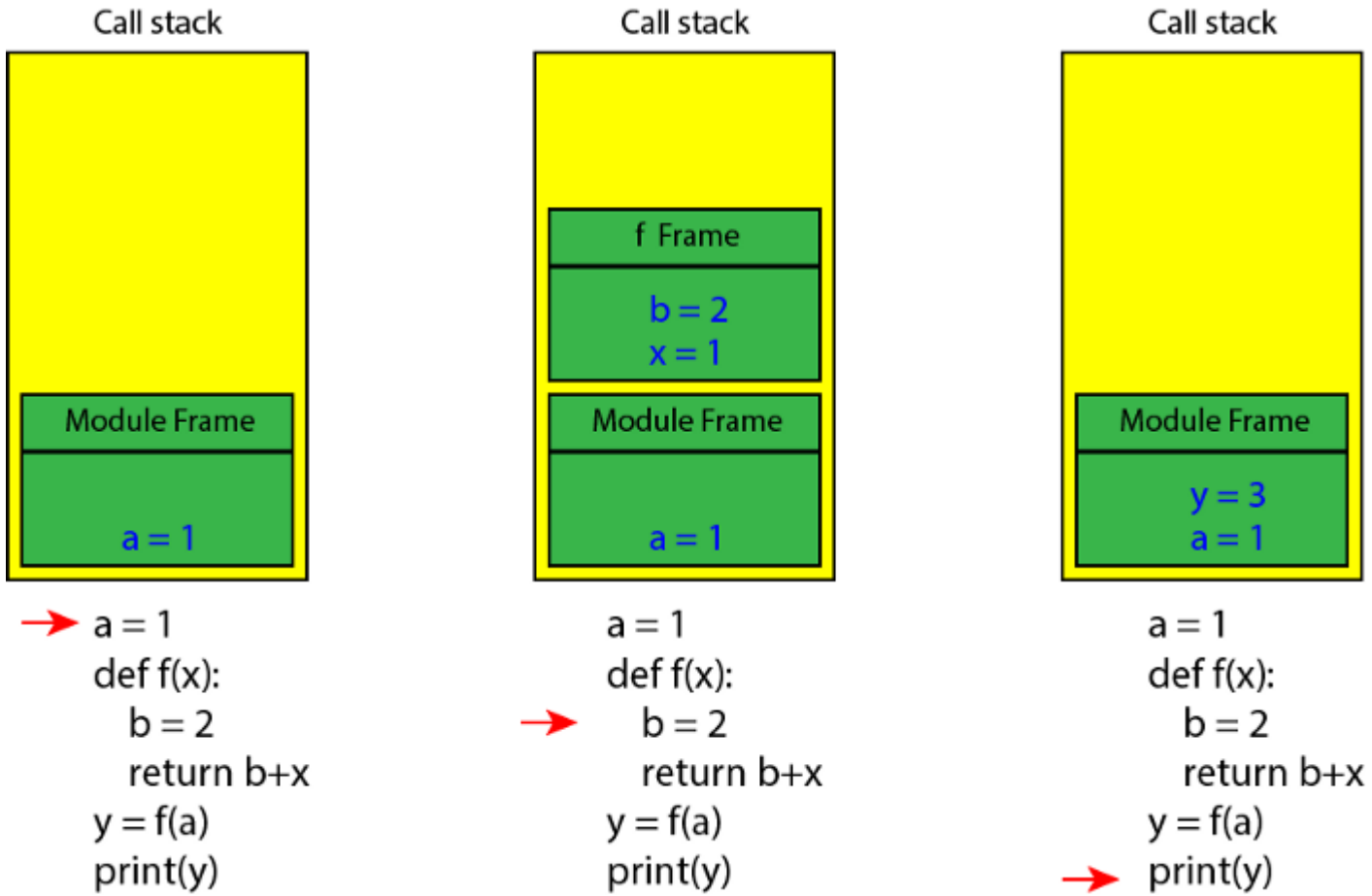
The total number of elements we look at is approximately 11 x $\lfloor \log n \rfloor$ which is $\theta(n \log n)$

Selection sort and Quicksort algorithms

	Selection sort		Quicksort	
	Time	Additional Space	Time	Additional Space
Best	$\theta(n^2)$	$\theta(1)$	$\theta(n \log n)$	$\theta(\log n)$
Worst	$\theta(n^2)$	$\theta(1)$	$\theta(n^2)$	$\theta(n)$
Average	$\theta(n^2)$	$\theta(1)$	$\theta(n \log n)$	$\theta(\log n)$

- Which algorithm is the *best*?

When a function is called, information is stored in the *call stack*



In quicksort, recursive function calls are stored on the stack,

- *n* times in the worst case
- roughly *log n* times in the best/average case.