Algorithms and Data Structures for Data Science

Search

CS 277
Brad Solomon

October 6, 2021

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Final Project Proposal

Proposal Deadline: October 20th

Submission: Commit proposal to GitHub.

`final_project/development/proposal.______`

Final Project Proposal Format

Leading Question: What problem are you trying to solve? How will you solve it for your given dataset / algorithm?

Dataset: What dataset are you using and how will it be used?

Algorithm: What algorithm are you implementing?

Timeline: When will you complete the stages of the project?

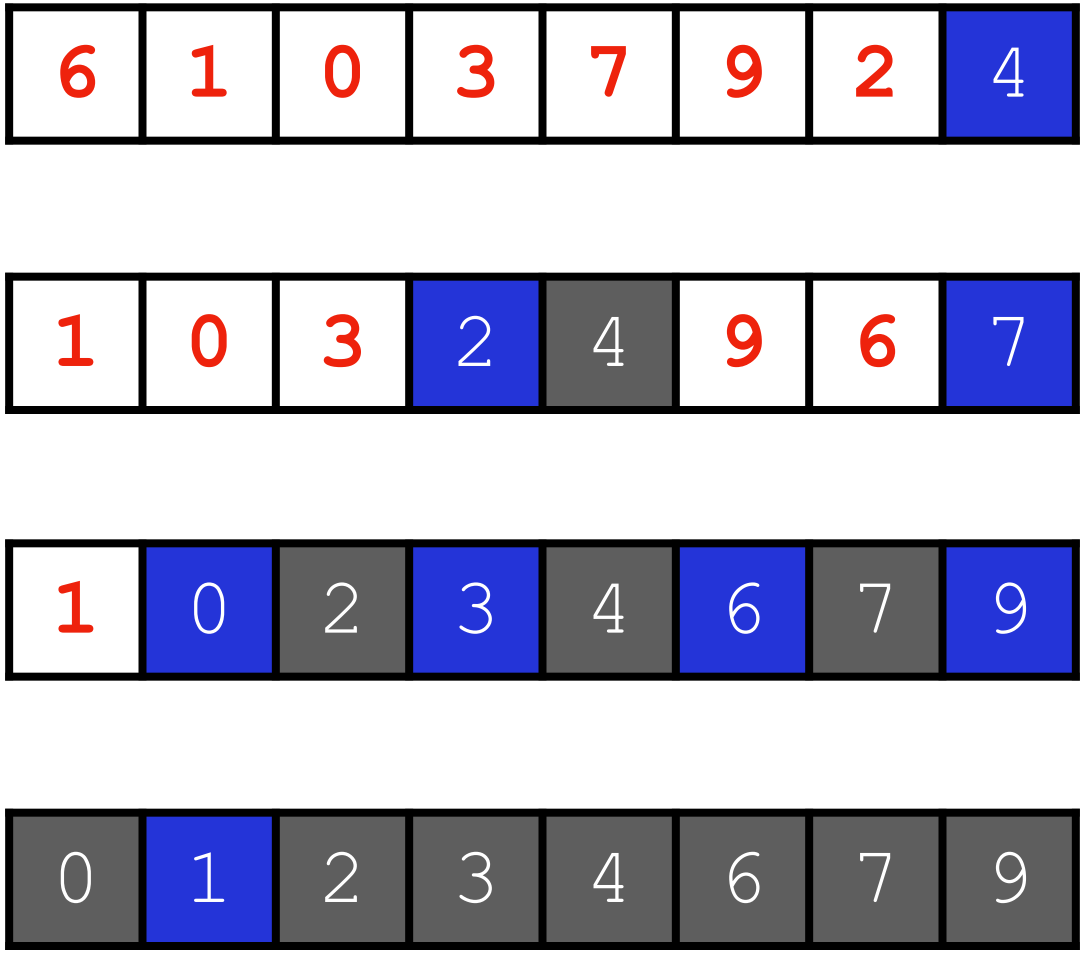
Learning Objectives

Discuss tradeoffs of sorting algorithms

Introduce the fundamental search problem

Introduce and implement binary search

QuickSort



1. Choose a *pivot* value
2. Divide the array into two partitions (larger and smaller than pivot)
3. Recursively QuickSort partitions

Selecting the pivot for quickSort

Can we do better than 'pick the last element in the list'?

0	1	2	3	4	5	6

Sorting Algorithm Tradeoffs

	Best Case Time	Worst Case time	Best Case Space	Worst Case Space
SelectionSort				
InsertionSort				
MergeSort				
QuickSort				

What sorting algorithm would you use…?

0	1	2	3	4	5	6	7	8	9

What sorting algorithm would you use…?

9	8	7	6	5	4	3	2	1	0

What sorting algorithm would you use…?

2	3	4	1	6	7	8	5	9	0

TimSort

An *adaptive* sort — adjusts behavior based on input data

Take advantage of *runs* of consecutive ordered elements

Order the merge steps to merge roughly equal sized lists together

When merging, reuse the allocated memory for each list

Speed up merging by ‘galloping’

The Search Problem

Given a collection of objects, C, with comparable values and an object of interest, q, find the first instance of $q \in C$.

Input:

4	5	6	7	8	9	10	11	12	13

Output: Index of q if it exists, -1 otherwise

Naive Linear Search

```python
def naive_linear(inList, val):

    for i, obj in enumerate(inList):

        if val == obj:

            return i
```

6	1	0	3	7	9	2	4

Naive Sorted Search

`Find(3)`

0	1	2	4	5	6	7	8	9	10

Naive Sorted Search

```
def naive_sorted(inList, val):

    for i, obj in enumerate(inList):

        if val == obj:

            return i

        elif val > obj:

            return -1
```

0	1	2	3	4	6	7	9

Optimal Sorted Searching

`Find(7)`

0	1	2	3	4	5	6	7	8	9

Binary Search

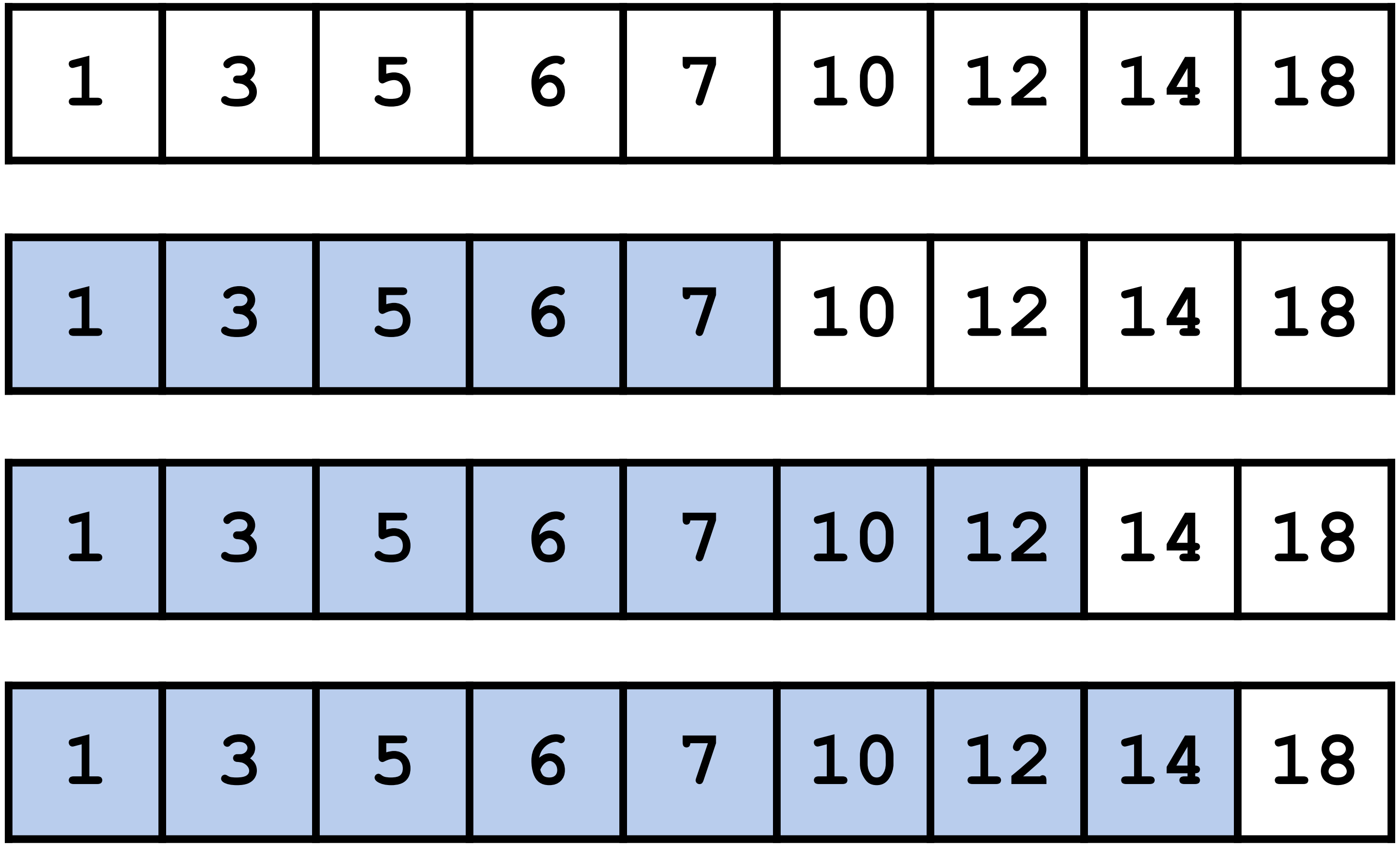
Find(8)

1	3	5	6	7	8	9

1	3	5	6	7	8	9

Binary Search

Find(18)



Binary Search

Find(4)

1	3	5	6	7	10	12	14	18

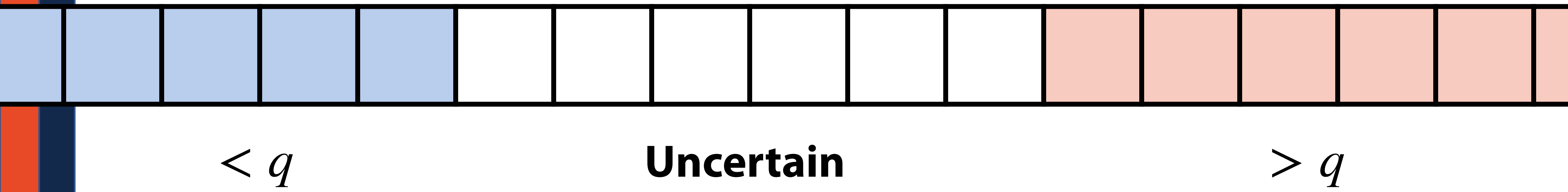
1	3	5	6	7	10	12	14	18

1	3	5	6	7	10	12	14	18

1	3	5	6	7	10	12	14	18

Binary Search

A binary search (for object q) partitions the search space into three regions



How can we track this information?

Binary Search

```
def binary_search(inList, q):



def recursive_BS(inList, q, start, end):
```

0	1	2	3	4	6	7	9

Binary Search

```
def binary_search(inList, q):

    start = 0
    end = len(inList) - 1

    while start <= end:
        mid = (start+end)//2

        pivot = inList[mid]
        if pivot == q:
            return mid
        elif pivot > q:
            end = mid - 1
        else:
            start = mid + 1

    return -1
```

0	1	2	3	4	6	7	9

Binary Search Efficiency

0	1	2	6	7	8	9	10	11

0	1	2	6

2	6

6

Logarithmic Efficiency

There are between 10^{78} and 10^{82} atoms in the universe. If we had a $O(log\ n)$ search tool, how many steps would it take to search?

Logarithmic Efficiency

We like data structures (and algorithms) that operate around a factor of $O(log\ n)$.