Sorting

Kelly Rivers and Stephanie Rosenthal

15-110 Fall 2019

Announcements

- Homework 3 full is due next week!

Learning Objectives

- To trace different sorting algorithms as they sort
- To compare and contrast different algorithms for sorting based on runtime

Big Picture

If we can get a lot of runtime benefit of having lists sorted prior to searching, can we also sort efficiently?

- If we can, we stand a chance of doing fast search
- If we can't, then we will be stuck spending a lot of time searching

Activity

10 volunteers sort yourselves by birthdate

Move only one person at a time

What algorithms do you use to sort yourselves?

Selection Sort Algorithm

Find the person with the earliest birthday and move them to the front

Find the person with the next birthday and move them next

Repeat until the last person has been moved to place

Selection Sort Picture

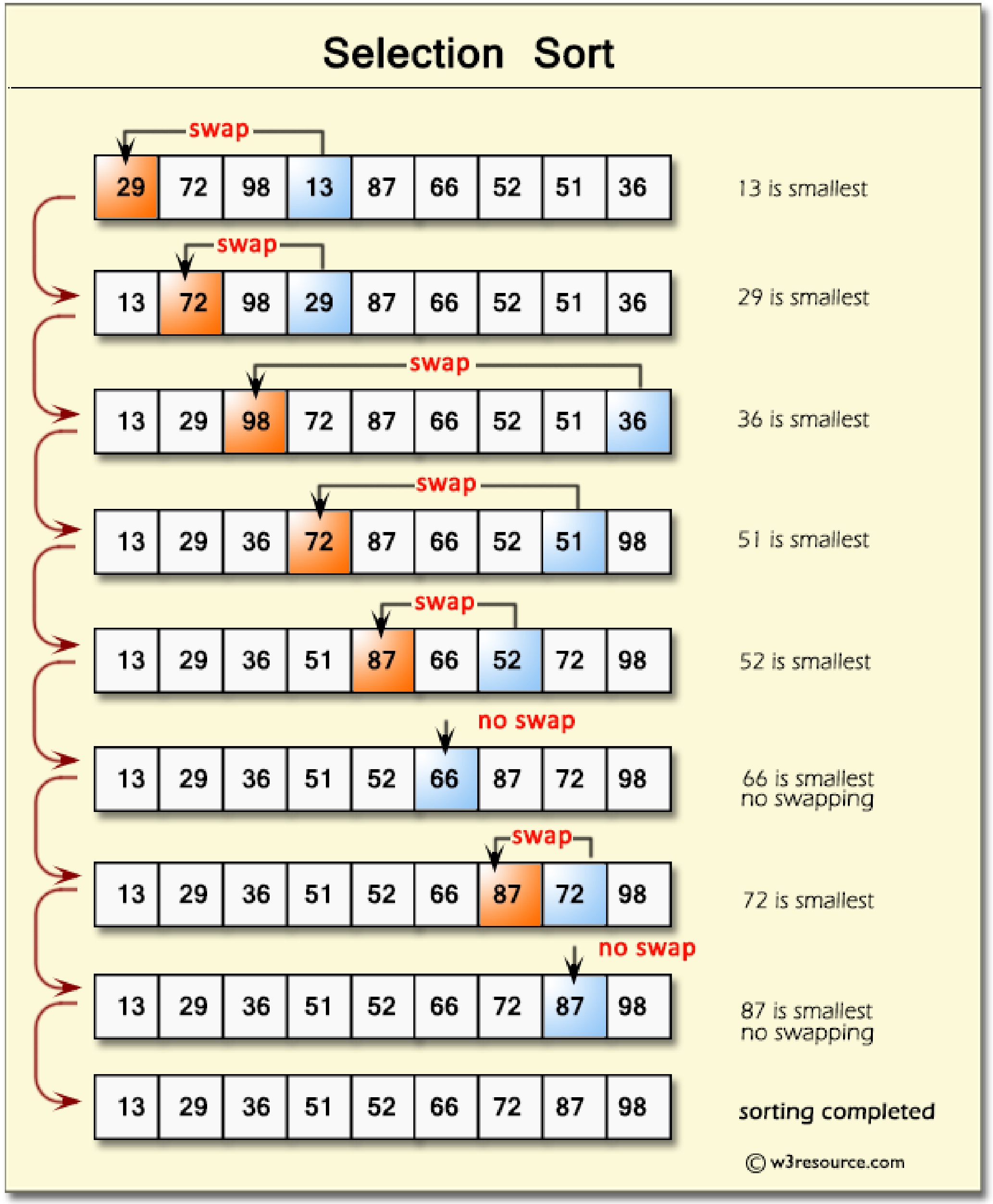
Find the smallest #, move to the front

Find the next smallest, move them next

Repeat until last has been moved

Note: swapping is faster than sliding all of the numbers down in the list

Why?



Selection Sort Code

```
def SelectionSort(L):
     for i in range(len(L)-1):
          # Find the minimum element in remaining
          min_idx = i
          for j in range(i+1, len(L)):
               if L[min_idx] > L[j]:
                    min_idx = j
          # Swap the found minimum element with the ith
          swap(L, i, min_idx)
```

Selection Sort Code Runtime?

```
def SelectionSort(L):
    for i in range(len(L)-1):
        # Find the minimum element in remaining
        min_idx = i
        for j in range(i+1, len(L)):
            if L[min_idx] > L[j]:
                min_idx = j
        # Swap the found minimum element with the ith
        swap(L, i, min_idx)
```

What is the worst case list to sort?

Reverse sorted list. Each element has to be moved.

Selection Sort Code Runtime

```
def SelectionSort(L):
    for i in range(len(L)-1):
        # Find the minimum element in remaining
        min_idx = i
        for j in range(i+1, len(L)):
            if L[min_idx] > L[j]:
                min_idx = j
        # Swap the found minimum element with the ith
        swap(L, i, min_idx)
```

Loop runs len(L)-1 times

Loop runs up to len(L) times
It decreases each outer loop
1 compare in inner loop

1 swap in outer loop

n + n-1 + n-2 + n-3 + … + 2 + 1 = n(n+1)/2 = O(n^2)

Insertion Sort Algorithm

Start with the first two elements and sort them (swap if necessary)

Take the third element, and move it left until it is in place

Continue to take the i’th element and move it left until it is in place

Stop when the last item was moved into place

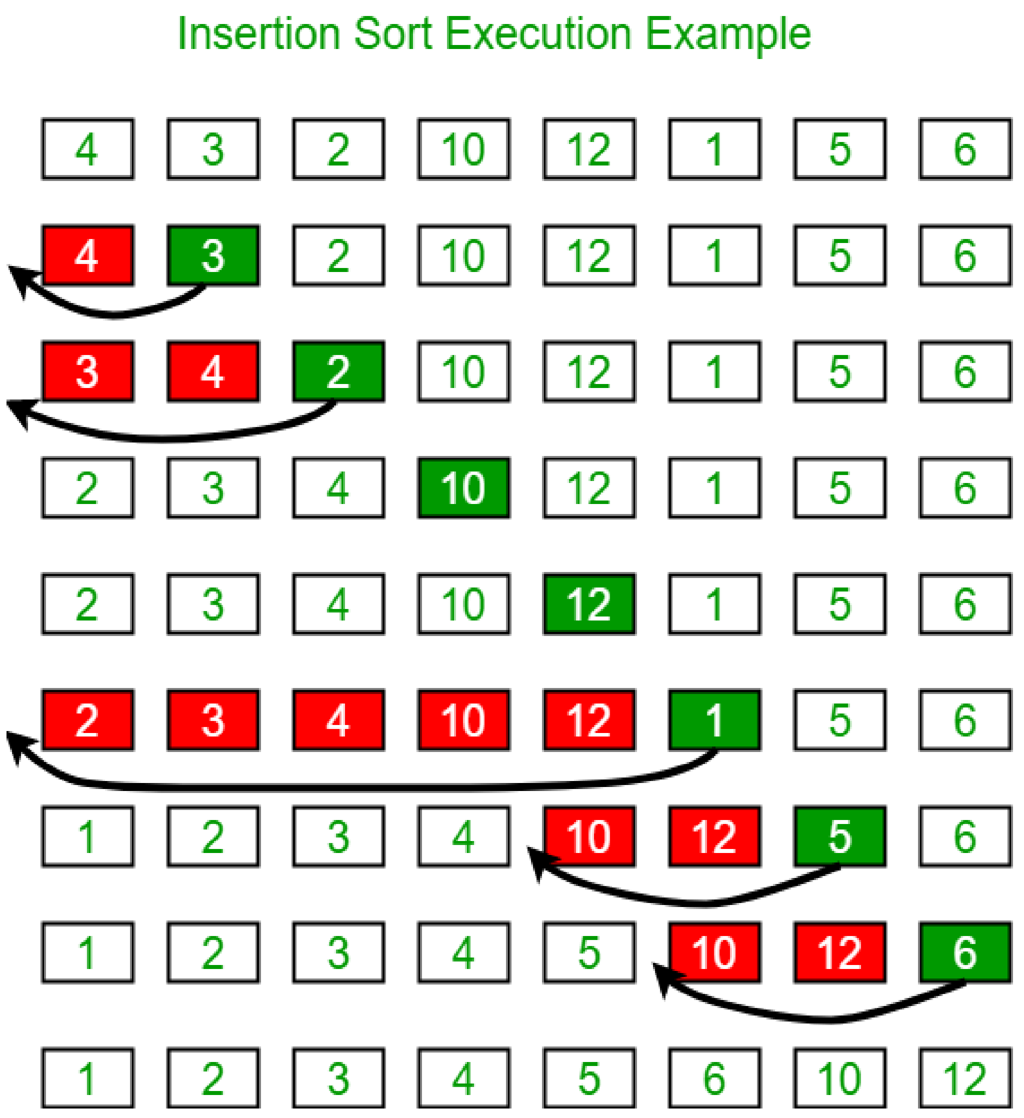
Insertion Sort Picture

Start with the first two elements and sort them (swap if necessary)

Take the third element, and move it left until it is in place

Continue to take the i'th element and move it left until it is in place

Stop when the last item was moved



Insertion Sort Code

```
def insertionSort(L):
    # Traverse through 1 to len(L)
    for i in range(1, len(L)):
        key = L[i]
        # Move elements of L[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < L[j]:
            L[j + 1] = L[j]
            j = j - 1
        L[j + 1] = key
```

Insertion Sort Code Runtime?

```
def insertionSort(L):
    # Traverse through 1 to len(L)
    for i in range(1, len(L)):
        key = L[i]
        # Move elements of L[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < L[j]:
            L[j + 1] = L[j]
            j = j - 1
        L[j + 1] = key
```

What is the worst case list to sort?

Reverse sorted list. Each element has to be moved.

Insertion Sort Code Runtime

```
def insertionSort(L):
    # Traverse through 1 to len(L)
    for i in range(1, len(L)):
        key = L[i]
        # Move elements of L[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < L[j]:
            L[j + 1] = L[j]
            j = j - 1
        L[j + 1] = key
```

Loop runs len(L)-1 times

Loop runs up to len(L) times

It increases each outer loop

1 swap in inner loop

1 + 2 + 3 + …+ n-1 + n = n(n+1)/2 = O(n^2)

MergeSort Algorithm

MergeSort the first half of the list

MergeSort the second half of the list

Merge the two sorted lists together

Recursive!

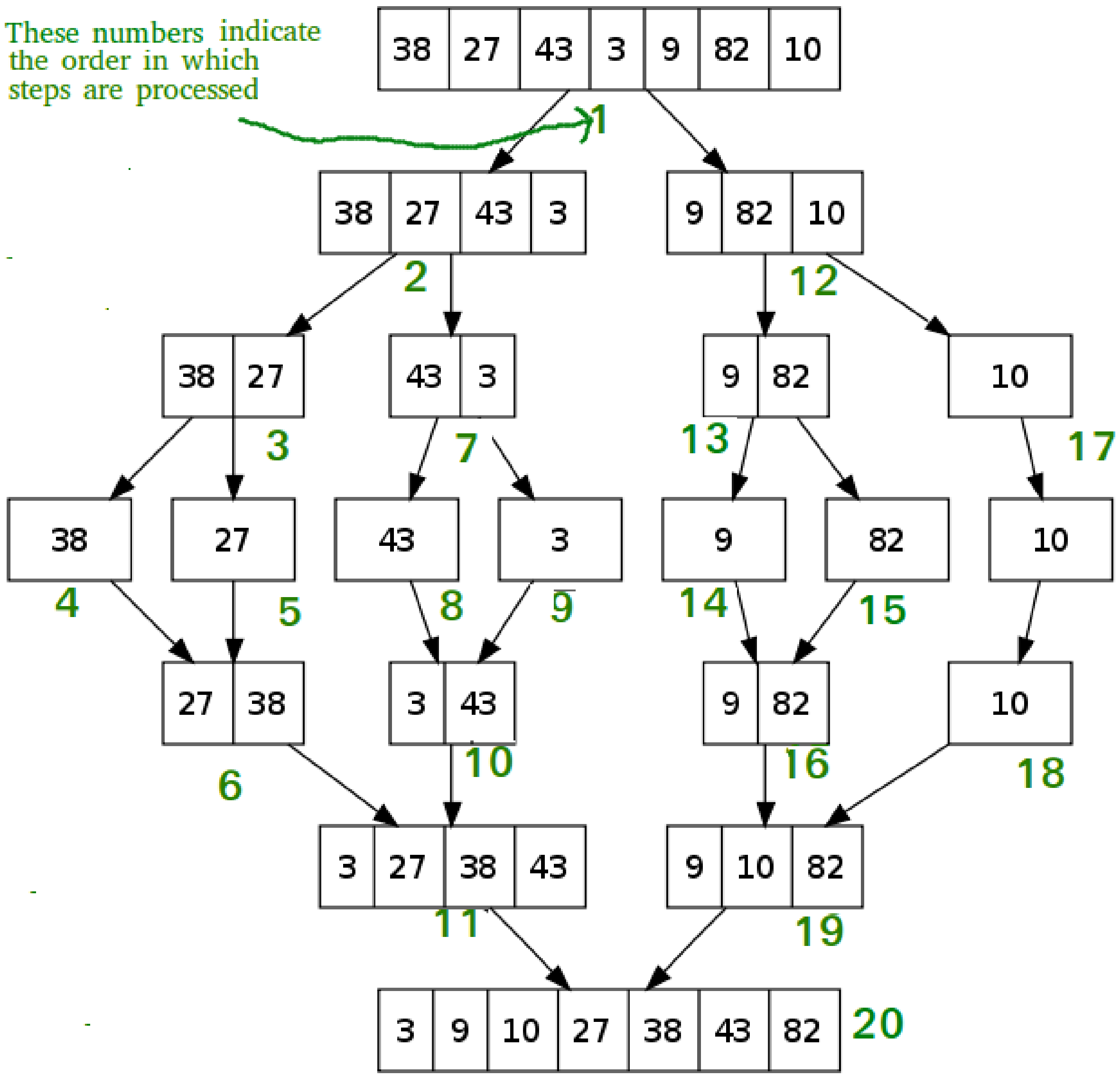
MergeSort Picture

MergeSort the first half of the list

MergeSort the second half of the list

Merge the two sorted lists together

Recursive!



MergeSort Runtime

```
def mergeSort(A):
    if len(A) <= 1: return A
    mid = len(A)//2 #Finding the mid of the array
    L = A[:mid] # Dividing the array elements
    R = A[mid:] # into 2 halves

    mergeSort(L) # Sorting the first half
    mergeSort(R) # Sorting the second half
    A = merge(L,R)
```

MergeSort Code

```
def mergeSort(A):
      if len(A) <= 1: return A
       mid = len(A)//2 #Finding the mid of the array
      L = A[:mid] # Dividing the array elements
      R = A[mid:] # into 2 halves

      mergeSort(L) # Sorting the first half
      mergeSort(R) # Sorting the second half
      A = merge(L,R)
```

Copy n/2 elements

Copy n/2 elements

Compare and Copy
n elements

3n copy/compares at each level * log(n) levels to divide len(A) by 2 repeatedly
= O(nlogn)

Takeaways

We can compare runtimes of different sorting algorithms

We can sort faster than O(n^2) using a divide/conquer approach

As you think about your code, try to imagine if you could change it to run faster than it currently does