Sorting Algorithms

Sorting

- The act of arranging data in groups according to some property of the data
- Sorting is a well-studied problem in CS
  - Many algorithms can be made faster if you first sort the data you operate on
- Examples in these slides will sort `ints`, but will work on other data types (unless indicated otherwise)

Determiing the Sort Order

- Values are sorted according to a “natural” ordering
  - Numerical values are sorted in ascending order
  - String sorted lexicographically
  - Objects in a class that implements the `Compariable` interface are sorted using the `compareTo` method

The Comparable Interface

- Has a single method: `compareTo`
  - The calling object is compared to the argument
  - Returns:
    - a negative integer if the calling object is less than the argument
    - 0 if the calling object is equal to the argument
    - a positive integer if the calling object is greater than the argument
  - Defines the "natural" ordering for objects in the class

Stable Sorts

- Stable Sort
  - A sort in which the relative order of equal elements is preserved
- Suppose you want to sort a deck of cards first by suit, then by value
  - A stable sort will ensure that the order of the suits will remain the same within the same value

3	2	4	2	4	3	→	2	2	3	3	4	4

Stable Sort

3	2	4	2	4	3	→	2	2	3	3	4	4

Non-Stable Sort

Sorting Performance

- Most single threaded sorts fall into one of two complexity categories
  - $O(n^2)$
  - $O(nlg(n))$
- All following examples assume and array named `arr` with length `n`
  - All examples will sort in ascending order

Selection Sort

- One of the simplest sorts
- Find the smallest value in the range `0` to `n-1` and put it in index 0 (swap the value at index 0 with the value at the found index)
- Repeat for index 1, index 2, ...
  - Essentially, select the element that should go at an index and put it there

Selection Sort Pseudocode

```
def find_min_index(arr, start):
    small_idx = start
    for (int i=start; i<arr.len; i++)
        if arr[i] < arr[small_idx]:
            small_idx = i
    return small_idx

def selection_sort(arr):
    for (int idx=0; i<arr.len; idx++):
        small_idx = find_min_index(arr, idx)
        swap(arr[idx], arr[small_idx])
```

Selection Sort Example

- Legend:
  - ⇑ `idx`
  - ↑ `small_idx`
- Original Array

  6 2 0 1 7
- Pass 1: `idx == 0`

6	2	0	1	7
⇑		↑		
0	2	6	1	7
⇑		↑		

- Pass 2: `idx == 1`

0	2	6	1	7
	⇑		↑	
0	1	6	2	7
	⇑		↑	

- Pass 3: `idx == 2`

0	1	6	2	7
		⇑	↑	
0	1	2	6	7
		⇑	↑	

Selection Sort Example

- Pass 4: `idx == 3`

$$\begin{array}{ccccc} 0 & 1 & 2 & 6 & 7 \\ & & & \Uparrow & \\ & & & \uparrow & \end{array}$$

- Pass 5: `idx == 4`

$$\begin{array}{ccccc} 0 & 1 & 2 & 6 & 7 \\ & & & & \Uparrow \\ & & & & \uparrow \end{array}$$

- Selection sort is not stable
  - Data gets additionally shuffled as the sort occurs
- Selection sort has very consistent speed
  - It's not very fast, but it will take roughly the same amount of time for a given array size

Selection Sort Complexity

- Call the length of the array n
- `find_min_index(arr, start)` takes, worst case, n times through the loop
- `small_idx = find_min_index(arr, i)` runs n times
- `swap`, while a function, is fast (4 operations ish)
  - Thus: Selection Sort is $O(n^2)$
  - As written (using indexing operations), this analysis assumes $O(1)$ random access; if using a LinkedList, `find_min_index` would be $O(n^2)$...

Insertion Sort

- Insertion sort works on the following principle:
  - A list of one element is always sorted with itself
  - Assume elements at indexes 0 through i are in sorted order
  - Take the element at index i+1 and insert it into its proper place
    - Swap the element back through the list as long as it is smaller than its predecessor
  - Start i at 1, and increment i until the entire list is sorted!

Insertion Sort Pseudocode

```
def insertion_sort(arr):
    for(i = 1; i < arr.len; i++):
        j = i
        while (j > 0 && arr[j] < arr[j-1])
            swap(arr[j], arr[j-1])
```

Insertion Sort Example

- Legend:

  $\Uparrow$ `i`

  $\uparrow$ `j-1`

- Original Array

  6 2 0 1 7

- Pass 1: `i == 1`

6	2	0	1	7
$\uparrow$	$\Uparrow$			
2	6	0	1	7
$\uparrow$	$\Uparrow$			

- Pass 2: `i == 2`

2	6	0	1	7
	$\uparrow$	$\Uparrow$		
2	0	6	1	7
$\uparrow$		$\Uparrow$		
0	2	6	1	7
		$\Uparrow$		

Insertion Sort Example

- Pass 3: `i == 3`

0	2	6	1	7
		$\uparrow$	$\Uparrow$	
0	2	1	6	7
	$\uparrow$		$\Uparrow$	
0	1	2	6	7
$\uparrow$			$\Uparrow$	

- Pass 4: `i == 4`

0	1	2	6	7
			$\uparrow$	$\Uparrow$

- Insertion sort, when implemented as above, is a stable sort
- Potentially improves on selection sort's performance by not sorting parts of the list that are already sorted

Insertion Sort Complexity

- To determine the complexity of insertion sort, we must count the number of times the inner loop runs
- When `i==1`, the inner loop runs at most once
- When `i==2`, the inner loop runs at most twice
- ...
- The inner loop runs a total of $1 + 2 + ... + (n-1) + n$ times (roughly)
  - In summation notation: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$
  - This reduces to $O(n^2)$

Comparing Insertion and Selection

Selection Sort

- Not stable
- Worst, average, and best time: $O(n^2)$
- In place sort (no new copies of data)

Insertion Sort

- Stable
- Worst time: $O(n^2)$
- Average time: $O(n^2)$
- Best time: $O(n)$ (list already sorted)
- Good for:
  - Short lists
  - Nearly sorted lists
- In place sort (no new copies of data)

Faster Sorting

- Selection and insertion sort are *comparison-based* sorting algorithms
  - Items are sorted by comparing elements to each other
- For any comparison-based sorting algorithm, it can be rigorously shown that:
  - The worst time is *at least* $O(nlg(n))$
  - The average time is *at least* $O(nlg(n))$
- Merge Sort and Quick Sort both have anaverage time of $O(nlg(n))$
  - linear-logrithmic
  - Both selection and insertion sort are quadratic ($O(n^2)$)

Merge Sort

- Use *Divide and Conquer* strategies to solve the problem
- Note that with selection and insertion sorts, halving the problem size quarters the runtime, so...
- Divide the array in half, and recursively apply merge sort to each half
  - There are variations on this
  - Some implementations continue until the list size is < 7, then use insertion sort
  - Some implementations keep splittng until the list size is 1, which is inherently sorted
- Once each half is sorted, merge the sorted arrays

Two Way Merge

- Given two sorted lists, combine them into a single sorted list
- Continually take the smallest element from each list and add it to the new list
- Best and worst case is $O(n)$

Merge Example

Merge the following arrays:

1 3 5 7   0 2 4 6

Merge Example

1	3	5	7

$\uparrow$

0	2	4	6

$\uparrow$

$$0 < 1$$

0

1	3	5	7

$\uparrow$

0	2	4	6

$\uparrow$

$$1 < 2$$

0	1

Merge Example

1	3	5	7
	$\uparrow$		

0	2	4	6
	$\uparrow$		

$$2 < 3$$

0	1	2			

1	3	5	7
	$\uparrow$		

0	2	4	6
		$\uparrow$	

$$3 < 4$$

0	1	2	3		

Merge Example

1	3	5	7		0	2	4	6
		$\uparrow$					$\uparrow$	

$$4 < 5$$

0	1	2	3	4		

1	3	5	7		0	2	4	6
		$\uparrow$						$\uparrow$

$$5 < 6$$

0	1	2	3	4	5		

Merge Example

1	3	5	7

0	2	4	6

$\uparrow$ $\uparrow$

$6 < 7$

0	1	2	3	4	5	6	

1	3	5	7

0	2	4	6

$\uparrow$

Only 7 left

0	1	2	3	4	5	6	7

Merge Pseudocode

```
def merge(A, B):
    int idxA = 0, idxB = 0
    merged = list()
    while (idxA < A.len && idxB < B.len):
        if (A[idxA] <= B[idxB]):
            merged.add(A[idxA])
            idxA++
        else:
            merged.add(B[idxB])
            idxB++
    # grab any remaining values from the nonempty List
    while (idxA < A.len):
        merged.add(A[idxA])
    while (idxB < B.len):
        merged.add(B[idxB])
    return merged
```

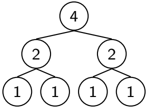
Merge Sort Pseudocode

```
def merge_sort(arr):
    if (arr.len == 1):
        return arr # single value sorted
    A = arr.subList(0,arr.len/2)
    B = arr.subList(arr.len/2, arr.len)
    merge_sort(A)
    merge_sort(B)
    sorted = merge(A, B)
    return sorted
```

Merge Sort Analysis

- Merge Sort is $O(nlg(n))$
  - Where does the $lg(n)$ come from?
- Recall that merge sort splits the list in half every time
  - Let's visualize the sort

Merge Sort Visual Breakdown



- The node values are the size of the list each call to the merge sort is operating on
- On the last level of the tree, there are n leaves, so there are roughly $2n$ nodes in the tree
  - This is a full tree!
  - The height is roughly $lg(n)$
- At each level, we are doing a merge worth $O(n)$ (on level 1, we are merging $\frac{n}{2}$ twice...)
- Thus, we do approximately n operations $lg(n)$ times: $O(nlg(n))$
- $lg(n)$ crops up frequently in divide and conquer operations

Merge Sort Summary

- Worst case (pivot is always largest or smallest): $O(nlg(n))$
- Average case: $O(nlg(n))$
- Not in place sort
- Not stable

Quick Sort

- Quick Sort is like Merge Sort in that it is a divide and conquer algorithm
- Quick Sort works as follows:
  - Choose a *pivot* value
  - Partition the array so everything to the left of the pivot is less than it, and everything to the right of the pivot is greater than it
  - Recursively Quick Sort the partitions

Choosing the Pivot

- A good pivot value is the median (as it divides the array in half)
- Calculating the exact median can be slow, so approximate it
  - Choose three random numbers from the data, and choose the median of the three

Partition Pseudocode

```
def partition(arr, start, end):
    choose pivot value
    b = start, c = end - 1 // assumes you're sorting
                           // to end (exclusive)
    while (b < c):
        while (arr[b] < pivot):
            b++
        while (arr[c] > pivot)
            c--;
        if (b <= c):
            swap(arr[b], arr[c])
    return index of partition (b)
```

Quick Sort Pseudocode

```
def quicksort(arr, start, end):
    if start >= end:
        return;
    pivot = partition(arr, start, end);
    quicksort(arr, start, pivot)
    quicksort(arr, pivot+1, end)
```

Quick Sort Summary

- The partitioning scheme presented can get hairy when you have duplicate elements
- Worst case (pivot is always largest or smallest): $O(n^2)$
- Average case: $O(nlg(n))$
- In place sort
- Not stable

Comparing Merge and Quick

Merge Sort

- Not stable
- Worst, average, and best time: $O(nlg(n))$
- Not in place

Quick Sort

- Not stable
- Worst time: $O(n^2)$
- Average time: $O(nlg(n))$
- Though poor worst time behavior, considered one of the most efficient sorting algorithms and is widely used
- In place sort (no new copies of data)

Radix Sort

- All sorts we've discussed have been comparison based
  - Worst time can be no better than $O(nlg(n))$
- What if we could sort without comparing?
- Radix Sort is not based on comparisons
  - Cannot be used to sort arbitrary data sets
  - Usually used for sorting unsigned integers

Radix Sort

- Radix sort uses “bucketing”
  - Group the numbers based on individual digits
- Example: Sort the following:

```
3 17 12 98 101 207 8 200 45 57 73 87 41 58
```

Radix Sort Example

`3 17 12 98 101 207 8 200 45 57 73 87 41 58`

- Consider each number from left to right
- Place each number in a list based on the ones digit
- Go through the buckets in order, and rewrite the list with the numbers in order

0:	200
1:	100 41
2:	12
3:	3 73
4:	
5:	45
6:	
7:	17 207 57 87
8:	98 8 58
9:	

`200 101 41 12 3 73 45 17 207 57 87 98 8 58`

Radix Sort Example

`200 101 41 12 3 73 45 17 207 57 87 98 8 58`

- Repeat the process using the tens digit
  - Single digit numbers have a 0 in the tens digit

0:	200 101 3 207 8
1:	12 17
2:	
3:	
4:	41 45
5:	57 58
6:	
7:	73
8:	87
9:	98

`200 101 3 207 8 12 17 41 45 57 58 73 87 98`

Radix Sort Example

`200 101 3 207 8 12 17 41 45 57 58 73 87 98`

- Repeat the process using the hundreds digit
  - Single and double digit numbers have a 0 in the hundreds digit
- After rewriting the list, the numbers are sorted
  - No number has 4 or more digits

0: 3 8 12 17 41 45 57 58 73 87 98
1: 101
2: 200 207
3:
4:
5:
6:
7:
8:
9:

`3 8 12 17 41 45 57 58 73 87 98 101 200 207`

Radix Sort analysis

- Stable Sort
- Not in place sort
- Average and worst time is $O(n \log_{10}(N))$
  - N is the largest integer in the array