Lecture 14: Sorting Algorithms
Administrivia

Robbie’s office hours today are cancelled (we have to grade your midterms)

Project 3 Partner form is due tonight (P3 will be released Wednesday)

Project 2 due Wednesday
Exercise 3 due Friday
INEFFECTIVE Sorts

```plaintext
DEFINE HALHEARTEDMERGESORT(list):
  if length(list) < 2:
    return list
  pivot = int(length(list) / 2)
  a = HALHEARTEDMERGESORT(list[:pivot])
  b = HALHEARTEDMERGESORT(list[pivot:])
  // UMMMM
  return [a, b] // HERE.. SORRY.
```

```plaintext
DEFINE FIRSTBOGOSORT(list):
  // AN OPTIMIZED BOGOSORT
  // RUNS IN O(N LOG N)
  for n from 1 to log(length(list)):
    shuffle(list)
    if issorted(list):
      return list
  return "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```plaintext
DEFINE JOHNMENDELQUICKSORT(list):
  ok so you choose a pivot
  then divide the list in half
  for each half:
    check to see if it’s sorted
    no list, it doesn’t matter
    compare each element to the pivot
    the bigger ones go in a new list
    the equal ones go into uh
    the second list from before
    hang on, let me name the lists
    this is list a
    the new one is list b
    put the big ones into list b
    now take the second list
    call it list uh a2
    which one was the pivot in?
    scratch all that
    it just recursively calls itself
    until both lists are empty
    right?
    not empty, but you know what I mean
    am I allowed to use the standard libraries?
```

```plaintext
DEFINE PANCICSORT(list):
  if issorted(list):
    return list
  for n from 1 to 10000:
    pivot = random(0, length(list))
    list = list[:pivot] + list[pivot:]
    if issorted(list):
      return list
  return list
  if issorted(list):
    return list
  if issorted(list):
    // THIS CAN’T BE HAPPENING
    return list
  if issorted(list):
    // COME ON COME ON
    return list
  // OH JEEZ
  // I'M GONNA BE IN SO MUCH TROUBLE
  list = []
  system("shutdown -n +5")
  system("rm -rf .")
  system("rm -rf ~/*")
  system("rm -rf ~/")
  system("rd /s /q c: ") // PORTABILITY
  return [1, 2, 3, 4, 5]
```
Where are we?

This course is “data structures and algorithms”

Data structures
- Organize our data so we can process it effectively

Algorithms
- Actually process our data!

We’re going to start focusing on algorithms

We’ll start with sorting
- A very common, generally-useful preprocessing step
- And a convenient way to discuss a few different ideas for designing algorithms.
Types of Sorts

Comparison Sorts
Compare two elements at a time
General sort, works for most types of elements

What does this mean? `compareTo()` works for your elements
- And for our running times to be correct, `compareTo` must run in $O(1)$ time.

Niche Sorts aka “linear sorts”
Leverages specific properties about the items in the list to achieve faster runtimes

niche sorts typically run $O(n)$ time

For example, we’re sorting small integers, or short strings.

In this class we’ll focus on comparison sorts
Sorting Goals

**In Place sort**

A sorting algorithm is in-place if it allocates $O(1)$ extra memory

Modifies input array (can’t copy data into new array)

Useful to minimize memory usage

**Stable sort**

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort

Why do we care?
- “data exploration” Client code will want to sort by multiple features and “break ties” with secondary features

- \[(8, \text{“fox”}), (9, \text{“dog”}), (4, \text{“wolf”}), (8, \text{“cow”})\]

- \[(4, \text{“wolf”}), (8, \text{“fox”}), (8, \text{“cow”}), (9, \text{“dog”})\]  
  Stable

- \[(4, \text{“wolf”}), (8, \text{“cow”}), (8, \text{“fox”}), (9, \text{“dog”})\]  
  Unstable

**Speed**

Of course, we want our algorithms to be fast.

Sorting is so common, that we often start caring about constant factors.
SO MANY SORTS

Quicksort, Merge sort, in-place merge sort, heap sort, insertion sort, intro sort, selection sort, timsort, cubesort, shell sort, bubble sort, binary tree sort, cycle sort, library sort, patience sorting, smoothsort, strand sort, tournament sort, cocktail sort, comb sort, gnome sort, block sort, stackoverflow sort, odd-even sort, pigeonhole sort, bucket sort, counting sort, radix sort, spreadsort, burstsort, flashsort, postman sort, bead sort, simple pancake sort, spaghetti sort, sorting network, bitonic sort, bogosort, stooge sort, insertion sort, slow sort, rainbow sort...
Goals

Algorithm Design (like writing invariants) is more art than science.

We’ll do a little bit of designing our own algorithms
- Take CSE 417 (usually runs in Winter) for more

Mostly we’ll understand how existing algorithms work

Understand their pros and cons
- Design decisions!

Practice how to apply those algorithms to solve problems
Algorithm Design Patterns

Algorithms don’t just come out of thin air.

There are common patterns we use to design new algorithms.

Many of them are applicable to sorting (we’ll see more patterns later in the quarter)

Invariants/Iterative improvement
  - Step-by-step make one more part of the input your desired output.

Using data structures
  - Speed up our existing ideas

Divide and conquer
  - Split your input
  - Solve each part (recursively)
  - Combine solved parts into a single
Principle 1

Invariants/Iterative improvement
- Step-by-step make one more part of the input your desired output.

We’ll write iterative algorithms to satisfy the following invariant:
After $k$ iterations of the loop, the first $k$ elements of the array will be sorted.
Insertion Sort

Sorted Items

Unsorted Items

Current Item

Sorted Items

Unsorted Items

Current Item

Sorted Items

Unsorted Items

Current Item

https://www.youtube.com/watch?v=ROalU379I3U
public void insertionSort(collection) {
    for (entire list)
        if (currentItem is smaller than largestSorted)
            int newIndex = findSpot(currentItem);
            shift(newIndex, currentItem);
}
public int findSpot(currentItem) {
    for (sorted list going backwards)
        if (spot found) return
}
public void shift(newIndex, currentItem) {
    for (i = currentItem > newIndex)
        item[i+1] = item[i]
        item[newIndex] = currentItem
}
Satisfying the invariant

We said this would be our invariant:

After $k$ iterations of the loop, the first $k$ elements of the array will be sorted.

But that wasn’t a full description of what happens

Insertion sort:

After $k$ iterations of the loop, the elements that started in indices 0, ..., $k - 1$ are now sorted

Selection sort:

After $k$ iterations of the loop, the $k$ smallest elements of the array are (sorted) in indices 0, ..., $k - 1$
Selection Sort

0 1 2 3 4 5 6 7 8 9
2 3 6 7 18 10 14 9 11 15

Sorted Items

Current Item

Unsorted Items

0 1 2 3 4 5 6 7 8 9
2 3 6 7 9 10 14 18 11 15

Sorted Items

Current Item

Unsorted Items

0 1 2 3 4 5 6 7 8 9
2 3 6 7 9 10 18 14 11 15

Sorted Items

Current Item

Unsorted Items

https://www.youtube.com/watch?v=Ns4TPTC8whw
Selection Sort

```java
public void selectionSort(collection) {
    for (entire list)
        int newIndex = findNextMin(currentItem);
        swap(newIndex, currentItem);
}

public int findNextMin(currentItem) {
    min = currentItem
    for (unsorted list)
        if (item < min)
            min = currentItem
    return min
}

public int swap(newIndex, currentItem) {
    temp = currentItem
    currentItem = newIndex
    newIndex = currentItem
}
```

<table>
<thead>
<tr>
<th>Sorted Items</th>
<th>Current Item</th>
<th>Unsorted Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

Worst case runtime? \(O(n^2)\)

Best case runtime? \(O(n^2)\)

In-practice runtime? \(O(n^2)\)

Stable? No

In-place? Yes
Principle 2

Selection sort:
After $k$ iterations of the loop, the $k$ smallest elements of the array are (sorted) in indices $0, \ldots, k - 1$
Runs in $\Theta(n^2)$ time no matter what.

Using data structures
- Speed up our existing ideas

If only we had a data structure that was good at getting the smallest item remaining in our dataset...
- We do!
Heap Sort

1. run Floyd’s buildHeap on your data
2. call removeMin n times

```java
public void heapSort(collection) {
    E[] heap = buildHeap(collection)
    E[] output = new E[n]
    for (n)
        output[i] = removeMin(heap)
}
```

Worst case runtime?  $O(n \log n)$
Best case runtime?  $O(n \log n)$
In-practice runtime?  $O(n \log n)$
Stable?  No
In-place?  

https://www.youtube.com/watch?v=Xw2D9aJRBY4
In Place Heap Sort

Heap Sorted Items

Current Item

Heap

Sorted Items

percolateDown(22)
In Place Heap Sort

```java
public void inplaceHeapSort(collection) {
    E[] heap = buildHeap(collection)
    for (n)
        output[n - i - 1] = removeMin(heap)
}
```

Complication: final array is reversed! Lots of fixes:
- Run reverse afterwards \(O(n)\)
- Use a max heap
- Reverse compare function to emulate max heap

- Worst case runtime? \(O(n \log n)\)
- Best case runtime? \(O(n \log n)\)
- In-practice runtime? \(O(n \log n)\)
- Stable? No
- In-place? Yes
Principle 3: Divide and Conquer

1. Divide your work into smaller pieces recursively
   - Pieces should be smaller versions of the larger problem

2. Conquer the individual pieces
   - Recursion!
   - Until you hit the base case

3. Combine the results of your recursive calls

```java
divideAndConquer(input) {
    if (small enough to solve)
        conquer, solve, return results
    else
        divide input into a smaller pieces
        recurse on smaller piece
        combine results and return
}
```
Merge Sort

https://www.youtube.com/watch?v=XaqR3G_NVoo

Sort the pieces through the **magic** of recursion
mergeSort(input) {
    if (input.length == 1)
        return
    else
        smallerHalf = mergeSort(new [0, ..., mid])
        largerHalf = mergeSort(new [mid + 1, ...])
    return merge(smallerHalf, largerHalf)
}

Worst case runtime? $T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} = O(n \log n)$

Best case runtime? Same

Average runtime? Same

Stable? Yes

In-place? No
There’s more than one way to divide!

Mergesort:

Split into two arrays.
- Elements that just happened to be on the left and that happened to be on the right.

Quicksort:

Split into two arrays.
- Elements that are “small” and elements that are “large”
- What do I mean by “small” and “large”?

Choose a “pivot” value (an element of the array)

One array has elements smaller than pivot, other has elements larger than pivot.
Quick Sort v1

### Divide

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

### Combine (no extra work if in-place)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>22</td>
<td>57</td>
<td>91</td>
</tr>
</tbody>
</table>

Sort the pieces through the **magic** of recursion

https://www.youtube.com/watch?v=ywWB6J5gz8
Quick Sort v1

```java
quickSort(input) {
    if (input.length == 1)
        return
    else
        pivot = getPivot(input)
        smallerHalf = quickSort(getSmaller(pivot, input))
        largerHalf = quickSort(getBigger(pivot, input))
        return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? \( T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + T(n - 1) & \text{otherwise} \end{cases} = O(n^2) \)

Best case runtime? \( T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + 2T(n/2) & \text{otherwise} \end{cases} = O(n \log n) \)

In-practice runtime? Just trust me \( O(n \log n) \)

Stable? No

In-place? Can be done
Quick Sort v2 (in-place)
Quick Sort v2 (in-place)

```java
quickSort(input) {
    if (input.length == 1)
        return
    else
        pivot = getPivot(input)
        smallerHalf = quickSort(getSmaller(pivot, input))
        largerHalf = quickSort(getBigger(pivot, input))
    return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + T(n - 1) & \text{otherwise} \end{cases}$ = $O(n^2)$

Best case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + 2T(n/2) & \text{otherwise} \end{cases}$ = $O(n\log n)$

In-practice runtime? Just trust me = $O(n\log n)$

Stable? No

In-place? Yes
Can we do better?

We’d really like to avoid hitting the worst case.

Key to getting a good running time, is always cutting the array (about) in half.

How do we choose a good pivot?

Here are four options for finding a pivot. What are the tradeoffs?
- Just take the first element
- Take the median of the first, last, and middle element
- Take the median of the full array
- Pick a random element as a pivot
Pivots

Just take the first element
- fast to find a pivot
- But (e.g.) nearly sorted lists get $\Omega(n^2)$ behavior overall

Take the median of the first, last, and middle element
- Guaranteed to not have the absolute smallest value.
- On real data, this works quite well...
- But worst case is still $\Omega(n^2)$

Take the median of the full array
- Can actually find the median in $O(n)$ time (google QuickSelect). It’s complicated.
- $O(n \log n)$ even in the worst case....but the constant factors are awful. No one does quicksort this way.

Pick a random element as a pivot
- somewhat slow constant factors
- Get $O(n \log n)$ running time with probability at least $1 - 1/n^2$
- “adversaries” can’t make it more likely that we hit the worst case.