Algorithms and Data Structures for Data Science

Sorting

CS 277
Brad Solomon

April 15, 2024

Department of Computer Science
Exam Information

Exam 3 (4/23 — 4/25)

Covering all material up to last Wednesday (April 10th)

Final Exam (05/02 — 05/06)

Expected time: 1 hour exam in 1 hour, 50 minute time block

50 minute makeup exams *during* final exam time!

Must take 3 makeup exams; take one of them.
Submit topics or concepts you want reviewed

Google form linked through Prairielearn

> Concepts to review

> Topics not seen that you want to see
We’ve seen most core data structures

Lists

Trees

Graphs

Hash Tables

But we haven’t seen a great deal of algorithms!

For the rest of the class, review core concepts…

And apply them to new problems!
Learning Objectives

Review fundamentals of lists and Big O

Introduce common sorting algorithms

Practice analyzing the Big O of different methods
List Implementations

Array List
Set of data in an ordered context
- Memory is continuous

Linked List
- Individual objects
- Growth allowed
Lists are a great way to store **data structures**.
Lists are a great way to store **data structures**

$$H = \{ h_1, h_2, \ldots, h_k \}$$
## Array Implementation

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Singly Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look up given an input <strong>index</strong></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Search given an input <strong>value</strong></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insert/Remove at <strong>front</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert/Remove at <strong>arbitrary</strong> location</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

*Assume array doesn't need to be resized.*
The Sorting Problem

Given a collection of objects, $C$, with comparable values, order the objects such that $\forall x \in C, x_i \leq x_{i+1}$
SelectionSort

1. Find the $i$-th smallest value
2. Place it at position $i$ via swap
3. Repeat for $0 \leq i \leq n - 1$

B: $0 \times n \times [O(n) + O(1)] = O(n^2)$

Expected swap:

$\text{tmp} = L[i]$, $O(1)$

$L[i] = L[0]$, $O(1)$

$L[0] = \text{tmp}$, $O(1)$
SelectionSort Efficiency

\[ \sum + (n-1) + 4(n-2) + \ldots + 1 \]

\[ = \frac{n(n-1)}{2} \approx O(n^2) \]
InsertionSort

1. Assume first value is ‘sorted’

2. Loop through remaining values:

3. Insert value into the ‘sorted’ array

Key trick: Insert by swapping!

Worst case performance:

\[ O(n^2) = O(n) \]
InsertionSort Efficiency

Worst case
Swap #

\[ 1 + 2 + 3 + \ldots + (n-1) \]

\( (large \ n) \)
Best and Worst Case insertionSort

Given the numbers 0 — 6, what is the **best** possible insertionSort input?

```
0 1 2 3 4 5 6
```

6 swaps

(Can run in \(O(n)\) time in the best case)

Given the numbers 0 — 6, what is the **worst** possible insertionSort input?

```
6 5 4 3 2 1 0
```
Recursive Array Sorting

Base Case: Array with one item is sorted

Recursive Step: Split list in half, sort both halves

Combining: Merge sorted lists
MergeSort

4 3 6 7 1
MergeSort

\[ \text{n} \leq 5 \]

4 3 6 7 1

4 3 6

4 3

4 3

Binary tree

Binary search tree

slow \rightarrow fast

O(n)
MergeSort

4 3 6 7 1

4 3 6

4 3

4 3

3 4

3 4 6

1 3 4 6 7

Split

Moise
Recursive MergeSort

1) Input list recursively split to a collection of “sorted” base cases

2) Sorted lists are merged back together
Recursive MergeSort Efficiency

1) Input list split in half

How many times do we have to split a list in half?

\[ \text{Height of recursion tree is } \log(n) \]
Recursive MergeSort Efficiency

2) Sorted lists are merged back together

3 2 1 4 ➔ 1 2 3 4
Recursive MergeSort Efficiency

2) Sorted lists are merged back together

Claim: Merging two sorted arrays can be done in $O(n + m)$ time
Recursive MergeSort Efficiency

\[ T(n) = 2 \ T(n/2) + C \ast n \leq \ \left( \frac{n}{2} + \frac{n}{2} \right) \]

- \( \text{Split list in half} \)
- \( \text{Merge on halves} \)

\[ \Rightarrow \ 0(n \ \log n) \]
Best and Worst Case mergeSort

Given the numbers 0 — 6, what is the **best** possible mergeSort input?

Given the numbers 0 — 6, what is the **worst** possible mergeSort input?
Optimal Sorting

**Claim:** Any deterministic comparison-based sorting algorithm must perform $O(n \log n)$ comparisons to sort $n$ objects.
## Sorting Algorithm Tradeoffs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case Time</th>
<th>Worst Case Time</th>
<th>Best Case Space</th>
<th>Worst Case Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionSort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertionSort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
# Sorting Algorithm Tradeoffs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case Time</th>
<th>Worst Case Time</th>
<th>Best Case Space</th>
<th>Worst Case Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionSort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Bonus Content: TimSort (Python’s built-in sort)

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

Take advantage of *runs* of consecutive ordered elements

Start by using insertionSort to build sorted lists of <= 64 elements

Use MergeSort once all sub-arrays are ordered

Additional heuristics speed up merging in practice
QuickSort

1. Choose a *pivot* value

6 1 0 3 7 9 2 4
QuickSort

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

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions
QuickSort

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions
QuickSort

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

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions
QuickSort

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

6 1 0 3 7 9 2 4 9 6 7

1 0 3 2 4 9 6 7

1 0 3 2 4 9 6 7

1 0 3 2 4 9 6 7

1 0 3 2 4 9 6 7

0 1 2 3 4 6 7 9
Recursive Quicksort

Base Case: A list of size 2 is sorted

Recursive Step: Quicksort on both partitions around pivot

Combining: Nothing! (Put pivot in between partitions)