Algorithms and Data Structures for Data Science

Sorting

CS 277
Brad Solomon

March 6, 2023
Reminder: mp_sketching due 3/20

MP is shorter than the first — can reasonably completed this week.
Exam 1 Retake

Retake exam much better than first!

Average: 76%      Median: 83%

Overall exam seemed more ‘fair’

If you are concerned about your grade, talk to me!
Exam 2 and Final Exam Registration Open

Exam 2 is **immediately after spring break** (sorry)

Must take exam between March 21-23

Content includes everything prior to spring break:

- Hashing (Hash tables, bloom filters, k-minima, minhash)
- Sorting (Selection, Insertion, Merge, and Quicksort)
- Search (Binary search and binary range search)
Informal Early Feedback

Form available here: Informal Early Feedback

Form is entirely anonymous — please give constructive criticism!

If majority of class fill it out, everyone gets bonus points!
Learning Objectives

Conceptualize mergeSort

Introduce and implement quickSort

Discuss sorting algorithm tradeoffs and real world sorting
Recursive Array Sorting

0 3 7 5 8 9 2 1 4 6

Base Case:

Recursive Step:

Combining:
MergeSort
Recursive MergeSort

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

2) Sorted lists are merged back together
Recursive MergeSort

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

```
1) def splitList(inList):
2)     if len(inList) > 1:
3)         mid = len(inList) // 2
4)         front = inList[:mid]
5)         back = inList[mid:]
6)         return front, back
7) # Add cases for empty / single value
```
Recursive MergeSort

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

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

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

3 2 1 4

Time:

Space:
MergeSort Efficiency

2) Sorted lists are merged back together
MergeSort Efficiency

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

2) Sorted lists are merged back together

1 3 8 5 0 7 6 2
1 3 5 8 0 7 2 6
1 3 5 8 0 2 6 7
0 1 2 3 4 5 6 7
MergeSort Efficiency

2) Sorted lists are merged back together

Time:

Space:
MergeSort

4 3 6 7 1

4 3 6

4 3

4 3

7 1

7 1

7 1
MergeSort

4 3 6 7 1

4 3 6

4 3

4 3

3 4

3 4 6

7 1

7 1

7 1

1 7

1 3 4 6 7
MergeSort by Index

More efficient (in space) if we recurse without allocating new lists

```python
def splitList_coords(inList):
    if len(inList) > 1:
        mid = len(inList) // 2
        frontCoords = (0, mid - 1)
        backCoords = (mid, len(inList) - 1)
        return frontCoords, backCoords
    # Add cases for empty / single value
```
MergeSort by Index

We still need $O(n)$ extra space to merge

0 3 5 2

0 3

2 5

0 2 3 5
<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Space:</strong></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertionSort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MergeSort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QuickSort

1. Choose a pivot value
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 → 1 0 3 2 4 9 6 7

1 0 3 2 4 9 6 7 → 1 0 2 3 4 6 7 9

1 0 2 3 4 6 7 9 → 0 1 2 3 4 6 7 9
Recursive Quicksort

Base Case:

Recursive Step:

Combining:
QuickSort is a ‘sort in place’ algorithm

1) quickSort recursion uses only a single array

```python
def quickSort(inList, f, e):
    if f < e:
        p = partition(inList, f, e)
        quickSort(inList, f, p - 1)
        quickSort(inList, p + 1, e)
```

6 1 0 3 7 9 2 4
QuickSort is a ‘sort in place’ algorithm

1) quickSort recursion uses only a single array

```python
def quickSort(inList, f, e):
    if f < e:
        p = partition(inList, f, e)
        quickSort(inList, f, p - 1)
        quickSort(inList, p + 1, e)
```

Input array: [6, 1, 0, 3, 7, 9, 2, 4]

Sorted array: [1, 0, 3, 2, 4, 9, 6, 7]
QuickSort is a ‘sort in place’ algorithm

1) quickSort recursion uses only a single array

```python
def quickSort(inList, f, e):
    if f < e:
        p = partition(inList, f, e)
        quickSort(inList, f, p - 1)
        quickSort(inList, p + 1, e)
```

...
QuickSort is a ‘sort in place’ algorithm

2) quickSort partitions the array into four regions

2 1 3 8 7 5 0 4

\[ \leq p \quad > p \quad ??? \quad p \]
QuickSort is a ‘sort in place’ algorithm

2) quickSort partitions the array into four regions

Uncertain
QuickSort is a ‘sort in place’ algorithm

2) quickSort partitions the array into four regions

2 1 3 8 7 5 0 4

2 1 3 0 7 5 8 4
QuickSort is a ‘sort in place’ algorithm

3) The last step in partition swaps the pivot

```
2 1 3 0 7 5 8 4
```

```
   |   |   |   |   |   |   |   |
```
def partition(inList, f, e):
    pivot = inList[e]
    i = f
    for j in range(f, e):
        if inList[j] <= pivot:
            inList[i], inList[j] = inList[j], inList[i]
            i+=1
    inList[i], inList[e] = inList[e], inList[i]
    return i
Best Case quickSort

Given the numbers 0 — 6, what is the best possible quickSort input?
Worst Case quickSort

Given the numbers 0 — 6, what is the worst possible quickSort input?
## Sorting Algorithm Tradeoffs

<table>
<thead>
<tr>
<th></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><strong>SelectionSort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>InsertionSort</strong></td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>MergeSort</strong></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>QuickSort</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selecting the pivot for quickSort

Can we do better than ‘pick the last element in the list’?
Next Class: Search

A sorted list is more efficient to search than a list — but how?

Consider: When do you want to search with a hash table vs a list?
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