CSCI 241
Lecture 6
Quicksort
Stability; Non-Comparison Sorts
Radix Sort
Announcements

• Quiz 1 is out on Gradescope - take it between 10am and 10pm today.
Goals:

• Thoroughly understand the mechanism of mergesort and quicksort.

• Be prepared to implement merge and partition helper methods.

• Know what it means for a sorting algorithm to be stable.

• Understand the distinction between comparison and non-comparison sorts.

• Be prepared to implement radix sort.
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (end-start < 2):
        return
    mid = (end+start)/2
    Divide
    mergeSort(A,start,mid) Conquer (left)
    mergeSort(A,mid, end) Conquer (right)
    merge(A, start, mid, end) Combine

https://visualgo.net/bn/sorting
Mergesort vs Quicksort

/** mergesort A[st..end]*/
mergeSort(A, st, end):
  if (small):
    return
  mid = (end+st)/2
  mergeSort(A, st, mid)
  mergeSort(A, mid, end)
  merge(A, st, mid, end)

/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A, st, end)
  quickSort(A, st, mid)
  quickSort(A, mid+1, end)
Mergesort vs Quicksort

/** mergesort A[st..end]*/
mergeSort(A, st, end):
    if (small):
        return
    mid = (end+st)/2
    mergeSort(A, st, mid)
    mergeSort(A, mid, end)
merge(A, st, mid, end)

/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return
    mid = partition(A, st, end)
    quickSort(A, st, mid)
    quickSort(A, mid+1, end)
Mergesort vs Quicksort

/** mergesort A[st..end]*/
mergeSort(A, st, end):
  if (small):
    return
  mid = (end+st)/2
  mergeSort(A, st, mid)
  mergeSort(A, mid, end)
  merge(A, st, mid, end)

/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A, st, end)
  quickSort(A, st, mid)
  quickSort(A, mid+1, end)
Mergesort vs Quicksort

/** mergesort A[st..end]*
mergeSort(A, st, end):
  if (small):
    return
  mid = (end+st)/2
  mergeSort(A, st, mid)
  mergeSort(A, mid, end)
merge(A, st, mid, end)

/** quicksort A[st..end]*
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A, st, end)
  quickSort(A, st, mid)
  quickSort(A, mid+1, end)
Quicksort

/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return

    mid = partition(A,st,end)

    quickSort(A,st,mid)
    quickSort(A,mid+1, end)
/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return

    mid = partition(A, st, end)

    quickSort(A, st, mid)
    quickSort(A, mid+1, end)
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A,st,end)
  quickSort(A,st,mid)
  quickSort(A,mid+1, end)
/** quicksort A[st..end]**/
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A, st, end)
  quickSort(A, st, mid)
  quickSort(A, mid+1, end)
Quicksort

/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return

    mid = partition(A, st, end)

    quickSort(A, st, mid)

    quickSort(A, mid+1, end)
QuickSort

Key issues:

```python
/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return

    mid = partition(A, st, end)

    quickSort(A, st, mid)
    quickSort(A, mid+1, end)
```
Quicksort

Key issues:

1. Picking the pivot
   • First, middle, or last
   • Median of first, middle, or last

```c
/** quicksort A[st..end]*/
quickSort(A, st, end):
    if (small):
        return
    mid = partition(A, st, end)
    quickSort(A, st, mid)
    quickSort(A, mid+1, end)
```
QuickSort

/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
  mid = partition(A, st, end)
  quickSort(A, st, mid)
  quickSort(A, mid+1, end)

Key issues:

1. Picking the pivot
   • First, middle, or last
   • Median of first, middle, or last

2. Implementing partition
Implementing Partition

Pre: A p

Inv: \[ \begin{array}{c|c|c|c} \leq p & p & ? & \geq p \end{array} \]

Post: A \begin{array}{c|c|c} \leq p & p & \geq p \end{array}
Implementing Partition

Pre: \[ A \begin{array}{c|c|c|c} p & \_ & \_ & ? \end{array} \]

\[ \uparrow \]

Inv: \[ A \begin{array}{c|c|c|c} <= p & p & ? & >= p \end{array} \]

Post: \[ A \begin{array}{c|c|c|c} <= p & p & >= p \end{array} \]
Implementing Partition

Post: $A \leq p \mid p \mid p \geq p$
Implementing Partition

Inv: $\begin{array}{c}
A \\
\leq p \\
\text{?} \\
\geq p
\end{array}$

Post: $\begin{array}{c}
A \\
\leq p \\
\geq p
\end{array}$

While $i < j$:
  
  if $A[i] \leq p$:
    
    swap $(A, i, i-1)$
    
    $i++$
  
  else:
    
    swap $(A, i, j-1)$
    
    $j--$
Questions?
Stability

Objects can be sorted on keys - different objects may have the same value.

A stable sort maintains the order of distinct elements with the same key.

- Example: sort a list of Student objects by first name only
- Example: sorting numbers on 10’s place only

\[
[ 6^* \ 2^* \ 6^+ \ 2^+ \ 3 \ 4 ]
\]
Stability

Objects can be sorted on **keys** - different objects may have the same value.

A **stable** sort maintains the order of distinct elements with the same key.

- Example: sort a list of Student objects by first name only
- Example: sorting numbers on 10’s place only

\[
\begin{array}{ccccccc}
6^* & 2^* & 6^+ & 2^+ & 3 & 4 \\
61 & 21 & 63 & 23 & 35 & 48 \\
\end{array}
\]
Stability

A **stable** sort maintains the order of elements with the same value.

Stably sorted: \[[ 6^* 2^* 6^+ 2^+ 3 4 ]\]

Unstably sorted: \[[ 2^+ 2^* 3 4 6^* 6^+ ]\]
Stability

A **stable** sort maintains the order of elements with the same value.

**In groups**: Sort this list using insertion and selection sort. Is either algorithm stable?

\[ 6 \,* \, 2 \,* \, 6 \,+ \, 2 \,+ \, 3 \, \, 4 \, \]

**insertionSort(A)**:

```python
i = 0;
while i < A.length:
    // push A[i] to its sorted position in A[0..i]
    // increment i
```

**selectionSort(A)**:

```python
i = 0;
while i < A.length:
    // find min of A[i..A.length]
    // swap it with A[i]
    // increment i
```
Comparison sorts operate by comparing pairs of elements.

Examples: all four sorts we've seen so far!
Comparison sorts operate by comparing pairs of elements.

Examples: all four sorts we've seen so far!

...is there any other way to do it?
Comparison sorts operate by comparing pairs of elements.

Examples: all four sorts we've seen so far!

...is there any other way to do it?

How do you sort without comparing elements?
How do you sort things without comparing them?

Suppose I gave you 10 sticky notes with the digits 0 through 9. What algorithm would you use to sort them?
How do you sort things without comparing them?

Suppose I gave you 10 sticky notes with the digits 0 through 9. What algorithm would you use to sort them?

How many times did you need to look at each sticky note?
How do you sort things without comparing them?

Suppose I gave you 10 sticky notes with the digits 0 through 9. What algorithm would you use to sort them?

How many times did you need to look at each sticky note?

What if there are duplicates?
/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
    do a stable sort of A on the dth least significant digit

// A is now sorted(!)
LSD Radix Sort

/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
    do a stable sort of A on the dth least significant digit

// A is now sorted(!)
LSD Radix Sort

/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
    do a stable sort of A on the dth least significant digit

// A is now sorted(!)

Don’t believe me? https://visualgo.net/en/sorting
LSD Radix Sort using queue buckets

Pseudocode from visualgo.net:

LSDRadixSort(A):
    create 10 buckets (queues) for each digit (0 to 9)
    for each digit (least- to most-significant):
        for each element in A:
            move element into its bucket based on digit
        for each bucket, starting from smallest digit
            while bucket is non-empty
                restore element to list
LSD Radix Sort
using queue buckets

Pseudocode from visualgo.net:

LSDRadixSort(A):
create 10 buckets (queues) for each digit (0 to 9)
for each digit (least- to most-significant):
    for each element in A:
        move element into its bucket based on digit
    for each bucket, starting from smallest digit
        while bucket is non-empty
            restore element to list
LSD Intuition: sort on most-significant digit last; if tied, yield to the next most significant digit, and so on. Only works because stability preserves orderings from less significant digits (previously sorted).
Exercise: Radix sort this

[  7, 19, 21, 11, 14, 54, 1, 8]

Hint: [07, 19, 21, 11, 14, 54, 01, 08]

LSDRadixSort(A):
create 10 buckets (queues) for each digit (0 to 9)
for each digit (least- to most-significant):
   for each element in A:
      move element into its bucket based on digit
   for each bucket, starting from smallest digit
      while bucket is non-empty
         restore element to list

Exercise: Radix sort this

\[07, 19, 61, 11, 14, 54, 01, 08\]

<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><strong>Buckets</strong> on 1’s place:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sorted on 1’s place:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Buckets</strong> on 10’s place:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sorted on 10’s place:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>