QUICKSORT

Admin

Compression assignment

MergeSort

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

MergeSort

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms(7 1 4 2 6 5 3 8)</td>
<td>ms(7 1 4 2)</td>
<td>ms(6 5 3 8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ms(7 1)</td>
<td>ms(4 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ms(7)</td>
<td>ms(1)</td>
<td>ms(4)</td>
<td>ms(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

merge!
MergeSort: implementation 1

```java
public static <E extends Comparable<E>> E[] mergeSort(E[] a) {
    if (a.length <= 1) {
        return a;
    } else {
        int mid = a.length/2;
        E[] left = Arrays.copyOfRange(a, 0, mid);
        E[] right = Arrays.copyOfRange(a, mid, a.length);
        E[] sortedLeft = mergeSort(left);
        E[] sortedRight = mergeSort(right);
        return merge(sortedLeft, sortedRight);
    }
}
```

MergeSort: implementation 1

```java
public static <E extends Comparable<E>> E[] mergeSort(E[] a) {
    if (a.length <= 1) {
        return a;
    } else {
        int mid = a.length/2;
        E[] left = Arrays.copyOfRange(a, 0, mid);
        E[] right = Arrays.copyOfRange(a, mid, a.length);
        E[] sortedLeft = mergeSort(left);
        E[] sortedRight = mergeSort(right);
        return merge(sortedLeft, sortedRight);
    }
}
```

requires copying the data

MergeSort: implementation 2

```java
mergeSortHelper(data, low, high)
if high-low > 1
    midPoint = low + (high-low)/2
    mergeSortHelper(data, low, mid)
    mergeSortHelper(data, mid, high)
    merge(data, low, mid, high)
```

How is this different?

```java
mergeSortHelper(data, low, high)
if high-low > 1
    midPoint = low + (high-low)/2
    mergeSortHelper(data, low, mid)
    mergeSortHelper(data, mid, high)
    merge(data, low, mid, high)
```

• doesn't require the extra copy!
• low/high specify the range we're sorting
• merge = mergeSortHelper(data, 0, data.length)
Merge:

merge(data, low, mid, high)

Assume:
- data starting at low up to, but not including, mid is sorted
- data starting at mid up to, but not including, high is sorted

Goal:
- data from low up to, but not including, high is sorted

Note: merge still requires an extra helper array!

MergeSort runtime

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

What is the runtime of mergeSort??

Merge runtime

```java
public static <E extends Comparable<? super E>> E[] merge(E[] left, E[] right) {
    E[] result = (E[]) new Objectized;
    int lIndex = 0;
    int rIndex = 0;
    for (int i = 0; i < size; i++) {
        if (lIndex == left.length) {
            result[i] = right[i];
        } else if (rIndex == right.length) {
            result[i] = left[i];
        } else if (left[lIndex].compareTo(right[rIndex]) <= 0) {
            result[i] = left[lIndex];
            lIndex++;
        } else {
            result[i] = right[rIndex];
            rIndex++;
        }
    }
    return result;
}
```

What is the runtime of merge?

```java
public static <E extends Comparable<? super E>> E[] merge(E[] left, E[] right) {
    E[] result = (E[]) new Objectized;
    int lIndex = 0;
    int rIndex = 0;
    for (int i = 0; i < size; i++) {
        if (lIndex == left.length) {
            result[i] = right[i];
        } else if (rIndex == right.length) {
            result[i] = left[i];
        } else if (left[lIndex].compareTo(right[rIndex]) <= 0) {
            result[i] = left[lIndex];
            lIndex++;
        } else {
            result[i] = right[rIndex];
            rIndex++;
        }
    }
    return result;
}
```

O(left.length + right.length) = O(n)
MergeSort runtime

<table>
<thead>
<tr>
<th>Divide the data in half</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call MergeSort on each half (resulting in two sorted halves)</td>
</tr>
<tr>
<td>Merge the two halves</td>
</tr>
</tbody>
</table>

Ignoring the cost of the recursive call, how much work is done per call of MergeSort?

O(n)

Mergesort runtime

<table>
<thead>
<tr>
<th>ms(length=n)</th>
<th>work done</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ms(n/2)</td>
</tr>
<tr>
<td>ms(n/2)</td>
<td>ms(n/2)</td>
</tr>
</tbody>
</table>

ms(length=n) work done
Mergesort runtime

<table>
<thead>
<tr>
<th>Work done</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms(length=n)</td>
<td>n</td>
</tr>
<tr>
<td>ms(n/2)</td>
<td>ms(n/2)</td>
</tr>
</tbody>
</table>

How much work is done for these calls?

17

Mergesort runtime

<table>
<thead>
<tr>
<th>Work done</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms(length=n)</td>
<td>n</td>
</tr>
<tr>
<td>ms(n/2)</td>
<td>ms(n/2)</td>
</tr>
</tbody>
</table>

n/2 + n/2 = n

How much work is done for these calls?

18

Mergesort runtime

<table>
<thead>
<tr>
<th>Work done</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms(length=n)</td>
<td>n</td>
</tr>
<tr>
<td>ms(n/2)</td>
<td>ms(n/2)</td>
</tr>
<tr>
<td>ms(n/4)</td>
<td>ms(n/4)</td>
</tr>
</tbody>
</table>

How much work is done for these calls?

19

Mergesort runtime

<table>
<thead>
<tr>
<th>Work done</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms(length=n)</td>
<td>n</td>
</tr>
<tr>
<td>ms(n/2)</td>
<td>ms(n/2)</td>
</tr>
<tr>
<td>ms(n/4)</td>
<td>ms(n/4)</td>
</tr>
</tbody>
</table>

How much work is done for these calls?

20
Mergesort runtime

How much work is done for these calls?

Mergesort runtime

How many levels are there?

Mergesort runtime

How many levels are there?

We've seen this before...

Each level down we decrease the size by 2.

\[
\text{length} = \frac{n}{2^{\text{level}}}
\]
We've seen this before...

Each level down we decrease the size by 2:

\[ \text{length} = \frac{n}{2^{\text{level}}} \]

We stop when the length is 1:

\[ 1 = \frac{n}{2^{\text{level}}} \]

\[ 2^{\text{level}} = n \]

\[ \text{level} = \log_2 n \]

MergeSort running time

O(n) work at each level

log n levels

Overall runtime: O(n log n) (best, worst, average)

MergeSort properties

Stable?

In-place?

MergeSort properties

Stable: yes!
MergeSort properties

Stable: yes

In-place: sort of
- easy way to implement is not
- very hard to do (but possible)
- in practice, variants of merge sort that combine with insertion sort are in-place

what does this method do?

```java
public static <E extends Comparable<? super E>> int partition(E[] vals, int start, int end)
   { int lessThanIndex = start-1;
     for (int i = start; i < end; ++i)
     { if (vals[i].compareTo(vals[lessThanIndex]) <= 0) {
         lessThanIndex++;
         swap(vals, start, lessThanIndex, i);
       }
     } swap(vals, start, lessThanIndex+1, end);
     return lessThanIndex+1;
   }
```

```java
... 5  7  1  2  8  4  3  6 ...  
      ^     ^     
     start   end
```

```java
public static <E extends Comparable<? super E>> int partition(E[] vals, int start, int end)
   { int lessThanIndex = start-1;
     for (int i = start; i < end; ++i)
     { if (vals[i].compareTo(vals[lessThanIndex]) == 0) {
         lessThanIndex++;
         swap(vals, lessThanIndex, i);
       }
     } swap(vals, lessThanIndex+1, end);
     return lessThanIndex+1;
   }
```

```java
... 5  7  1  2  8  4  3  6 ...  
      ^     ^     
     start   end
```
public static <E extends Comparable<? super E>> int partition(E[] vals, int start, int end)
    int lessThanIndex = start-1;
    for (int i = start; i < end; ++i)
        if (vals[i].compareTo(vals[lessThanIndex]) == 0)
            swap(vals, lessThanIndex, i);
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;

public static <E extends Comparable<?>> int partition(E[] vals, int start, int end) {
    int lessThanIndex = start - 1;
    for (int i = start; i < end; ++i) {
        if (vals[i].compareTo(vals[end]) < 0) {
            swap(vals, lessThanIndex, i);
            lessThanIndex++;
        }
    }
    swap(vals, lessThanIndex + 1, end);
    return lessThanIndex + 1;
}
What's happening?

```
public static <E extends Comparable<? super E>> int partition(E[] vals, int start, int end){
  let lessThanIndex = start-1;
  for let i = start; i < end; ++i{  
    if vals[i].compareTo(vals[lessThanIndex]) == 0 } {  
      let temp = vals[i];  
      swap(vals, lessThanIndex, i);  
    }
  swap(vals, lessThanIndex+1, end);  
  return lessThanIndex+1;
}
```
Partition running time?

vals[end] is called the pivot

Partitions the elements nums[start...end-1] into two sets, those ≤ pivot and those > pivot

Operates in place

Final result:
Partition running time?

O(n)

```java
public static <E extends Comparable<? super E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for(int i = start; i < end; ++i){
        if(vals[i].compareTo(vals[end]) < 0) {
            swap(vals, lessThanIndex, i);
            lessThanIndex++;
        }
    }
    swap(vals, lessThanIndex+1, end); return lessThanIndex+1;
}
```

Quicksort

How can we use this method to sort?

```java
public static <E extends Comparable<? super E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for(int i = start; i < end; ++i){
        if(vals[i].compareTo(vals[end]) < 0) {
            swap(vals, lessThanIndex, i);
            lessThanIndex++;
        }
    }
    swap(vals, lessThanIndex+1, end); return lessThanIndex+1;
}
```

Quicksort

```java
void quicksortHelper(E[] vals, int start, int end){
    if(start < end){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

```java
public static <E extends Comparable<? super E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for(int i = start; i < end; ++i){
        if(vals[i].compareTo(vals[end]) < 0) {
            swap(vals, lessThanIndex, i);
            lessThanIndex++;
        }
    }
    swap(vals, lessThanIndex+1, end); return lessThanIndex+1;
}
```

```java
void quicksortHelper(E[] vals, int start, int end){
    if(start < end){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

8 5 1 3 6 2 7 4
Quicksort

void quicksortHelper(int[] vals, int start, int end) {
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition - 1);
        quicksortHelper(vals, partition + 1, end);
    }
}

Quicksort

void quicksortHelper(int[] vals, int start, int end) {
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition - 1);
        quicksortHelper(vals, partition + 1, end);
    }
}

Quicksort

void quicksortHelper(int[] vals, int start, int end) {
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition - 1);
        quicksortHelper(vals, partition + 1, end);
    }
}

Quicksort

void quicksortHelper(int[] vals, int start, int end) {
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition - 1);
        quicksortHelper(vals, partition + 1, end);
    }
}
void quicksortHelper(int[] vals, int start, int end){
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}

void quicksortHelper(int[] vals, int start, int end){
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
Quicksort

```
void quicksortHelper(E[] vals, int start, int end){
  if (start < end){
    int partition = partition(vals, start, end);
    quicksortHelper(vals, start, partition-1);
    quicksortHelper(vals, partition+1, end);
  }
}
```

What happens here?

```
void quicksortHelper(E[] vals, int start, int end){
  if (start < end){
    int partition = partition(vals, start, end);
    quicksortHelper(vals, start, partition-1);
    quicksortHelper(vals, partition+1, end);
  }
}
```

Quicksort

```
void quicksortHelper(E[] vals, int start, int end){
  if (start < end){
    int partition = partition(vals, start, end);
    quicksortHelper(vals, start, partition-1);
    quicksortHelper(vals, partition+1, end);
  }
}
```

Quicksort

```
void quicksortHelper(E[] vals, int start, int end){
  if (start < end){
    int partition = partition(vals, start, end);
    quicksortHelper(vals, start, partition-1);
    quicksortHelper(vals, partition+1, end);
  }
}
```
Running time of Quicksort?

Worst case?

Each call to Partition splits the array into an empty array and n-1 array

\[
n-1 + n-2 + n-3 + ... + 1 = O(n^2)
\]

When does this happen?
- sorted
- reverse sorted
- near sorted/reverse sorted
Quicksort best case?

Each call to Partition splits the array into two equal parts

... How much work is done at each "level", i.e. running time of a level?

\[ O(n) \]

73

Quicksort best case?

Each call to Partition splits the array into two equal parts

... How many levels are there?

Similar to mergesort, each call to Partition will throw away half the data until we're down to one element: \( \log_2 n \) levels

74

Quicksort best case?

Each call to Partition splits the array into two equal parts

... Overall runtime?

\[ O(n \log n) \]

75

Quicksort Average case?

Two intuitions

- As long as the Partition procedure always splits the array into some constant ratio between the left and the right, say 1-to-\( R \), e.g. 9-to-1, then we maintain \( O(n \log n) \)

- As long as we only have a constant number of "bad" partitions intermixed with a "good partition" then we maintain \( O(n \log n) \)

76
How can we avoid the worst case?

Inject randomness into the data

```c
void randomizedPartition(E [] nums, int start, int end){
    int i = randomInt(start, end);
    swap(nums, i, end);
    return partition = partition(nums, start, end);
}
```

Randomized quicksort is average case $O(n \log n)$

What is the worst case running time of randomized Quick sort?

$O(n^2)$

We could still get very unlucky and pick “bad” partitions at every step

Quicksort properties

Stable?

- No

In-place?

- Yes