CSCI 241
Lecture 5
Recursive Sorting:
Mergesort and Quicksort
Announcements

• Quiz 0 scores are out.

• I will post a video going over it (Q1.mp4) later today.
Goals:

• Be able to **understand** and **develop** recursive methods *without* thinking about the details of how they are executed.

• Know the generic steps of a **divide-and-conquer** algorithm.

• Thoroughly understand the mechanism of **mergesort** and **quicksort**.

• Be prepared to implement **merge** and **partition** helper methods.
How do we execute recursive methods?

```python
/** return n!; pre: n >= 0 */
fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)

fact(3)
=> 3 * fact(2)
=> 2 * fact(1)
=> 1 * fact(0)
=> 1
```
How do we understand recursive methods?
How do we understand recursive methods?

1. Make sure it has a precise specification.
How do we understand recursive methods?

1. Make sure it has a precise specification.
2. Make sure it works in the base case.
How do we understand recursive methods?

1. Make sure it has a **precise specification**.

2. Make sure it works in the **base case**.

3. Ensure that each recursive call makes **progress** towards the base case.
How do we understand recursive methods?

1. Make sure it has a precise specification.

2. Make sure it works in the base case.

3. Ensure that each recursive call makes progress towards the base case.

4. Replace each recursive call with the spec and verify overall behavior is correct.
How do we understand recursive methods?

```python
/** returns # of ‘e’ in string s */
def count_e(s):
    if len(s) == 0:
        return 0
    first = 0
    if s[0] == 'e':
        first = 1
    return first + count_e(s[1:end])
```
How do we understand recursive methods?

/** returns # of ‘e’ in string s */
def count_e(s):
    if len(s) == 0:
        return 0
    first = 0
    if s[0] == ‘e’:
        first = 1
    return first + count_e(s[1..end])
How do we understand recursive methods?

```python
/** returns # of ‘e’ in string s */
def count_e(s):
    if len(s) == 0:
        return 0
    first = 0
    if s[0] == ‘e’:
        first = 1
    return first + count_e(s[1..end])
```

1. spec
2. base case
How do we understand recursive methods?

```python
/** returns # of ‘e’ in string s */
def count_e(s):
    if len(s) == 0:
        return 0
    first = 0
    if s[0] == ‘e’:
        first = 1
    return first + count_e(s[1:end])
```

1. **spec**
2. **base case**
3. **progress**
How do we understand recursive methods?

```python
/** returns # of 'e' in string s */
def count_e(s):
    if len(s) == 0:
        return 0
    first = 0
    if s[0] == 'e':
        first = 1
    return first + count_e(s[1:end])
```

1. spec
2. base case
3. progress
4. recursive call —> spec
Got it?

This code has **at least one** bug:

1. Spec
2. Base case
3. Progress
4. Recursive call
   
   \(\iff\) spec
Got it?

This code has at least one bug:

dup(String s):
    if s.length == 0:
        return s

    return s[0] + s[0] + dup(s)
Got it?

```java
/** return a copy of s with each * character repeated */
dup(String s):
    if s.length == 0:
        return s
    return s[0] + s[0] + dup(s)
```

1. Spec
2. Base case
3. Progress
4. Recursive call

<=> spec!
/** return a copy of s with each
 * character repeated */

dup(String s):
  if s.length == 0:
    return s
  return s[0] + s[0] + dup(s)

1. Spec
2. Base case
3. Progress
4. Recursive call
   ⇔ spec
/** return a copy of s with each * character repeated */
dup(String s):
    if s.length == 0:
        return s
    
    return s[0] + s[0] + dup(s[1..s.length])

3. progress!
How do we develop recursive methods?

1. Write a **precise specification**.

2. Write a **base case** without using recursion.

3. Define all other cases in terms of **subproblems** of the same kind.

4. Implement these definitions using the **recursive call** to compute solutions to the subproblems.
Examples:

- civic
- radar
- deed
- racecar

**Recursive** definition: A string s is a palindrome if

- \( s\text{.length} < 2 \), OR
- \( s[0] == s[\text{end}-1] \) AND \( s[1..\text{end}-2] \) is a palindrome
Recursive definition: A string so is a palindrome if

- \( s.\text{length} < 2 \), OR
- \( s[0] == s[\text{end}-1] \) AND \( s[1..\text{end}-2] \) is a palindrome

\[ \text{racecar} \]

\[ \text{palindrome} \]
Recursive definition: A string so is a palindrome if
• $s \cdot \text{length} < 2$, OR
• $s[0] == s[end-1]$ AND $s[1..end-2]$ is a palindrome

**Problem 3:** Write a recursive palindrome checker:

```java
/** return true iff $s[start..end]$ is a palindrome */
public boolean isPal(s, start, end) {
    // your code here
}
```
Incremental Algorithms

solve a problem a little bit at a time.
Incremental Algorithms

solve a problem a little bit at a time.
Incremental Algorithms
solve a problem a little bit at a time.

Natural programming mechanism: loops
Incremental Algorithms

solve a problem a little bit at a time.

Natural programming mechanism: loops
Incremental Algorithms
solve a problem a little bit at a time.

Natural programming mechanism: loops

<table>
<thead>
<tr>
<th>A</th>
<th>sorted</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

insertion sort
Divide-and-Conquer Algorithms

solve a problem by breaking it into smaller problems.

https://upload.wikimedia.org/wikipedia/commons/f/fe/Quicksort.gif
Divide-and-Conquer Algorithms solve a problem by breaking it into smaller problems. (easier!)
Divide-and-Conquer Algorithms solve a problem by breaking it into smaller problems. (easier!)

https://upload.wikimedia.org/wikipedia/commons/f/fe/Quicksort.gif
Divide-and-Conquer Algorithms

solve a problem by breaking it into smaller problems.

Natural programming mechanism: recursion

(easier!)

https://upload.wikimedia.org/wikipedia/commons/f/fe/Quicksort.gif
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 2):
        return
    mid = (end+start)/2
    mergeSort(A,start,mid)
    mergeSort(A,mid, end)
    merge(A, start, mid, end)
An example of Divide-and-Conquer

/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 2):
        return
    mid = (end+start)/2  1. Divide
    mergeSort(A,start,mid)
    mergeSort(A,mid, end)
    merge(A, start, mid, end)
An example of Divide-and-Conquer

/** sort A[start..end] using mergesort */
mergeSort(A, start, end):

if (A.length < 2):
    return

mid = (end+start)/2  

1. Divide

mergeSort(A,start,mid)
mergeSort(A,mid, end)

2. Conquer

merge(A, start, mid, end)
An example of Divide-and-Conquer

```python
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 2):
        return
    mid = (end+start)/2
    mergeSort(A, start, mid)
    mergeSort(A, mid, end)
    merge(A, start, mid, end)

1. Divide
2. Conquer
3. Combine
```

1. Divide
2. Conquer
3. Combine
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end+start)/2

mergeSort(A, start, mid)

mergeSort(A, mid, end)

merge(A, start, mid, end)
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 2):
        return
    mid = (end+start)/2
    Divide

mergeSort(A,start,mid)

mergeSort(A,mid, end)

merge(A, start, mid, end)
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 2):
        return
    mid = (end+start)/2
    mergeSort(A, start, mid)
    mergeSort(A, mid, end)
    merge(A, start, mid, end)
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 2):
        return
    mid = (end+start)/2
    mergeSort(A, start, mid)    Conquer (left)
    mergeSort(A, mid, end)       Conquer (right)
merge(A, start, mid, end)
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 2):
        return
    mid = (end+start)/2
    mergeSort(A, start, mid)
    mergeSort(A, mid, end)
    merge(A, start, mid, end)
Merge Step

- Merge two halves, each of which is sorted.
Merge Step

- Merge two halves, each of which is sorted.

https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci241_18f/img/merge.gif
Merge step: Loop Invariant

Precondition:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>sorted</td>
<td>sorted</td>
</tr>
<tr>
<td>A</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

Postcondition:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>sorted</td>
<td>sorted</td>
</tr>
<tr>
<td>A</td>
<td>merged</td>
<td></td>
</tr>
</tbody>
</table>
## Merge step: Loop Invariant

**Precondition**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>sorted</td>
<td>sorted</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Invariant**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>copied</td>
<td>not yet copied</td>
<td>copied</td>
</tr>
<tr>
<td>A</td>
<td>merged</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Postcondition**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>sorted</td>
<td>sorted</td>
</tr>
<tr>
<td>A</td>
<td>merged</td>
<td></td>
</tr>
</tbody>
</table>
Merge step: Loop Invariant

\[ \text{Post} \]

\[ \begin{array}{c|c|c}
B & \text{sorted} & \text{sorted} \\
\hline
A & \text{merged} & \\
\end{array} \]

mid
Merge step: Loop Invariant

Inv

Post

sorted | sorted
merged
Merge step: Loop Invariant

Inv

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>i</td>
</tr>
<tr>
<td>j</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>merged</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>not yet copied</td>
<td>copied</td>
</tr>
<tr>
<td>copied</td>
<td>not yet copied</td>
</tr>
</tbody>
</table>

Post

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td>sorted</td>
</tr>
</tbody>
</table>

A | merged
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
    if (A.length < 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 A[st..end]*/
mergeSort(A, st, end):
    if (small):
        return
    mid = (end−start)/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, end)
Quicksort

/** mergesort A[st..end]*/
mergeSort(A, st, end):
    if (small):
        return
    mid = (end - start) / 2
    Divide
    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, end)
Quicksort

/** mergesort A[st..end]*/
mergeSort(A, st, end):
  if (small):
    return

  mid = (end-start)/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, end)
/** mergesort A[st..end]*/
mergeSort(A, st, end):
    if (small):
        return
    mid = (end - start)/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
    Divide mid = partition(A, st, end)
    Conquer quickSort(A, st, mid)
    quickSort(A, mid, end)
Combine NO OP

Quicksort
Quicksort

```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)
```

Unsorted:
/** 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

```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)
```

Unsorted:

Small things left, big things right:

Sort left things:
Unsorted:

Small things left
big things right:

Sort left things:

Sort right things:

/** 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

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

Key issues:

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

2. Implementing `partition`

```cpp
/** 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)
```