Announcements

1. Midterm review options:
   1. Review sheet posted
   2. Review sessions:
      1. Thursday 5-6:20pm (Shari Haynes, in B00 lecture Pepper Canyon 109)
      2. Friday 10-10:50am (me, in this room)
Today’s topics

- Recursion review
- MergeSort
- Quicksort
Insertion sort: Best case analysis

Approximately how many steps does it take to insert the element into the sorted part each time through the loop (in the BEST case)?

A. 1
B. N
C. N^2
D. It depends on the length of the sorted part

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Element to insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Which algorithm could produce this list after 3 iterations:

4  5  12  62  45  21  47  30

A. Selection sort
B. Insertion sort
C. Both
D. Neither
Which algorithm could produce this list after 3 iterations:

4  15  12  58  45  62  68  75

A. Selection sort
B. Insertion sort
C. Both
D. Neither
Which algorithm could produce this list after 3 iterations:

14  15  12  53  45  95  64  59

A. Selection sort
B. Insertion sort
C. Both
D. Neither
Recursion: A review
Recursive vs. Normal functions

Here's a function $f$:

$$f(n) = \begin{cases} 1 & \text{if } n < 1, \\ n^3 + 5n + 7 & \text{if } n > 1. \end{cases}$$

What is $f(10)$?

$10 > 1$, so we apply the second part of the definition:

$$1000 + 50 + 7 = 1057$$
Recursive vs. Normal functions

Recursive Definition

\[ n! = \]
\[ \text{if } n \text{ is 1, then } n! = 1 \]
\[ \text{if } n > 1, \text{ then } n! = n \times (n - 1)! \]

What is 10! ?

10 > 1, so we apply the second part of the definition

10! = 10 \times 9!

*Well wait, we don’t have an answer yet!*
Recursive function

A function that calls itself directly or indirectly.

Solves a series of sub-problems that get smaller and smaller in size.

\[ n! \rightarrow (n-1)! \rightarrow (n-2)! \rightarrow \ldots \rightarrow 1! \]

Has a ‘base’ case, the smallest problem size for the given problem.
Recursion

Recursive definition

\[ n! = \begin{align*}
  & \text{if } n \text{ is } 1, \text{ then } n! = 1 \\
  & \text{if } n > 1, \text{ then } n! = n \times (n - 1)! 
\end{align*} \]

Recursive code

```java
long factorial ( int n ) {
    S.o.p(n);
    if (n==1) return 1;
    return n*factorial(n-1);
}
```

S.o.p is short for System.out.println
Recursion

Recursive definition

\[ n! = \]
  if \( n \) is 1, then \( n! = 1 \)
  if \( n > 1 \), then \( n! = n \times (n - 1)! \)

Recursive code

```c
long factorial ( int n ) {
    S.o.p(n);
    if (n==1) return 1;
    return n*factorial(n-1);
}
```

What is the \textbf{third} value printed when we call \texttt{factorial(10)}?
A. 2
B. 3
C. 7
D. 8
E. None of the above
Recursion

Recursive definition

\[ n! = \]
\[ \text{if } n \text{ is } 1, \text{ then } n! = 1 \]
\[ \text{if } n > 1, \text{ then } n! = n \times (n - 1)! \]

Recursive code

```c
long factorial ( int n ) {
    S.o.p(n);
    if (n==1) return 1;
    return n*factorial(n-1);
}
```

What is the first value ever returned when we call `factorial(10)`?
A. 1
B. 3
C. 5
D. 9
E. None of the above
Recursion

The sub-problems must get smaller. Otherwise, will end up with infinite recursion.

```java
1 long factorial ( int n ) {
2   if ( n == 1 ) return 1;
3   return n * factorial(n);
4 }
```
Recursion

How does all this look in memory?

Memory

Space for method calls, local variables

Current free space available for the application.

All things ‘new’ - Objectville
Recursion

Example code

```java
Integer[] myarr;
myarr = new Integer[100];
```

Space allotted for the references in the array
Recursion

Example code

```java
public void add(int a, int b){
    int c = a + b;
    return c
}
```
Recursion

Memory After

Main and other methods
mymethod()
x, xfac

Recursive code

```c
long factorial ( int n ) {
    S.o.p(n);
    if (n==1) return 1;
    return n*factorial(n-1);
}

void mymethod() {
    int x = 10;
    long xfac = 0;
    nfac = factorial(10);
}
```
Recursion

### Memory After

<table>
<thead>
<tr>
<th>Main and other methods</th>
<th>mymethod()</th>
<th>x, xfac</th>
<th>factorial()</th>
<th>n = 10</th>
</tr>
</thead>
</table>

### Recursive code

```c
long factorial ( int n ) {
    S.o.p(n);
    if (n==1) return 1;
    return n*factorial(n-1);
}
```

```c
void mymethod () {
    int x = 10;
    long xfac = 0;
    nfac = factorial(10);
}
```
Recursion

Memory After

<table>
<thead>
<tr>
<th>Main and other methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>mymethod()</td>
</tr>
<tr>
<td>x, xfac</td>
</tr>
<tr>
<td>factorial()</td>
</tr>
<tr>
<td>n = 10</td>
</tr>
<tr>
<td>factorial()</td>
</tr>
<tr>
<td>n = 9</td>
</tr>
</tbody>
</table>

Recursive code

```c
long factorial ( int n ) {
    S.o.p(n);
    if (n==1) return 1;
    return n*factorial(n-1);
}

void mymethod()
{
    int x = 10;
    long xfac = 0;
    nfac = factorial(10);
}
```
**Recursion**

**Recursive code**

```c
long factorial ( int n ) {  
    S.o.p(n);  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}  

void mymethod()  
{  
    int x = 10;  
    long xfac = 0;  
    nfac = factorial(10);  
}  
```

**Memory After**

<table>
<thead>
<tr>
<th>Main and other methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>mymethod()</td>
</tr>
<tr>
<td>x, xfac</td>
</tr>
<tr>
<td>factorial()</td>
</tr>
<tr>
<td>n = 10</td>
</tr>
<tr>
<td>factorial()</td>
</tr>
<tr>
<td>n = 9</td>
</tr>
<tr>
<td>factorial()</td>
</tr>
<tr>
<td>n = 8</td>
</tr>
</tbody>
</table>
Recursive code

```c
long factorial ( int n ) {
    S.o.p(n);
    if (n==1) return 1;
    return n*factorial(n-1);
}
```

How many different values of n will be on the stack at the same time, at most, during the execution of factorial(10)?

(A) 1  (B) 2  (C) 9  (D) 10  (E) other/none
Recursion

Memory

Main and other methods
blahblah()
blahblah()
blahblah()
blahblah()

If you run out of stack space, any further requests for stack space will result in StackOverflowError.
MergeSort: The Magic of Recursion

Consider this magical way of sorting lists:

```
  12   4   9   3   15   8   19   2
```

Split the list in half:

```
  12   4   9   3
```

```
  15   8   19   2
```

Magically sort each list

```
  3   4   9   12
```

```
  2   8   15   19
```

Merge the two lists back together

```
  2   3   4   8   9   12   15   19
```
public int[] merge(int[] a, int[] b) {
    int indexA = 0;
    int indexB = 0;
    int indexRet = 0;
    int[] ret = new int[a.length+b.length];

    while (indexA < a.length && indexB < b.length) {
        // Fill in code here
        indexRet++;
    }

    for (; indexA < a.length; indexA++, indexRet++)
        ret[indexRet] = a[indexA];
    for (; indexB < b.length; indexB++, indexRet++)
        ret[indexRet] = a[indexA];
    return ret;
}
public int[] merge(int[] a, int[] b) {
    int indexA = 0;
    int indexB = 0;
    int indexRet = 0;
    int[] ret = new int[a.length+b.length];

    while (indexA < a.length && indexB < b.length) {
        if (a[indexA] < b[indexB]) {
            ret[indexRet] = a[indexA];
            indexA++;
        } else {
            ret[indexRet] = b[indexB];
            indexB++;
        }
        indexRet++;
    }
    for (; indexA < a.length; indexA++, indexRet++)
        ret[indexRet] = a[indexA];
    for (; indexB < b.length; indexB++, indexRet++)
        ret[indexRet] = a[indexA];
    return ret;
}
public int[] merge( int[] a, int[] b )
{
    int indexA = 0;
    int indexB = 0;
    int indexRet = 0;
    int[] ret = new int[a.length+b.length];

    while ( indexA < a.length && indexB < b.length ) {
        if ( a[indexA] < b[indexB] ) {
            ret[indexRet] = a[indexA];
            indexA++;
        }
        else {
            ret[indexRet] = b[indexB];
            indexB++;
            indexRet++;
        }
        for ( ; indexA < a.length; indexA++, indexRet++ )
            ret[indexRet] = a[indexA];
        for ( ; indexB < b.length; indexB++, indexRet++ )
            ret[indexRet] = a[indexA];
    }
    return ret;
}

Important differences between this code and HW5:
1. Your merge will use a “work/scratch array” that you pass in and merge in place, instead of creating a new array inside the method.
2. You will be sorting (and merging) objects of type Comparable. See the book and Java’s Comparable interface for details on how to compare objects using the compareTo method.
Consider this magical way of sorting lists:

| 12 | 4  | 9  | 3  | 15 | 8  | 19 | 2  |

Split the list in half:

| 12 | 4  | 9  | 3  |     |     |     |    |
| 15 | 8  | 19 | 2  |     |     |     |    |

Magically sort each list – using the same sorting method we are implementing!

| 3  | 4  | 9  | 12 |     |     |     |    |
| 2  | 8  | 15 | 19 |     |     |     |    |

Merge the two lists back together
public void mergeSort( int[] toSort, int[] workArray, int start, int end )
{
    int mid = start + ((end - start) / 2);
    mergeSort( toSort, workArray, start, mid );
    mergeSort( toSort, workArray, mid+1, end );
    merge( toSort, workArray, start, mid, end );

    "In place" merge method.
    Assume it works
}

Does this mergeSort method work?
A. Yes
B. No
public void mergeSort( int[] toSort, int[] workArray, int start, int end )
{
    int mid = start + ((end - start) / 2);
    mergeSort( toSort, workArray, start, mid );
    mergeSort( toSort, workArray, mid+1, end );
    merge( toSort, workArray, start, mid, end );

    "In place" merge method.
    Assume it works
}

Does this mergeSort method work?
A. Yes
B. No
public void mergeSort( int[] toSort, int start, int end )
{
    if ( start < end ) {
        int mid = start + ((end - start) / 2);
        mergeSort( toSort, workArray, start, mid );
        mergeSort( toSort, workArray, mid+1, end );
        merge( toSort, workArray, start, mid, end );
    }
}

This mergeSort works!!
Merge Sort: An Example

Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
Merge Sort: An Example

Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.

How many split steps are there?
A. \( \log(N) \)
B. \( N \)
C. \( N^2 \)
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.

About how instructions are executed at each split?
A. 1
B. log(N)
C. N
D. \( N^2 \)
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
Running time for Mergesort:
$\log(N) \times 1 + \log(N) \times N = O(N \times \log(N))$
Quicksort: Another magical (recursive) algorithm

Select a **pivot** element:

```
 12  4  9  14  15  8  19  2
```

“Partition” the elements in the array (**smaller**, **pivot**, **larger**)

```
  8  4  9  2  12  15  19  14
```

Magically sort the smaller elements and the larger elements (Quicksort)

```
  2  4  8  9  12  15  19  21
```
Quicksort: Another magical (recursive) algorithm

Select a pivot element:

```
12  4  9  14  15  8  19  2
```

“Partition” the elements in the array (smaller, pivot, larger)

```
 8  4  9  2  12  15  19  14
```

We won’t cover how the partition step works, but see if you can figure it out! (or Google it)
Quick Sort: Using a “good” pivot

How many levels will there be if you choose a pivot that divides the list in half?
A. 1
B. log(N)
C. N
D. N*log(N)
E. N^2
Quick Sort: Using a “good” pivot

If the time to partition on each level takes N comparisons, how long does Quicksort take with a good partition?

A. $O(1)$
B. $O(\log(N))$
C. $O(N)$
D. $O(N \log(N))$
E. $O(N^2)$
Quick Sort: Using a “good” pivot

If the time to partition on each level takes N comparisons, how long does Quicksort take with a good partition?

A. \( O(1) \)
B. \( O(\log(N)) \)
C. \( O(N) \)
D. \( O(N \times \log(N)) \)
E. \( O(N^2) \)

Space complexity: \( O(\log_2 n) \) for the runtime stack activation records
Which of these choices would be the worst choice for the pivot?

A. The minimum element in the list
B. The last element in the list
C. The first element in the list
D. A random element in the list
Quick sort with a bad pivot

If the pivot always produces one empty partition and one with \( n - 1 \) elements, there will be \( n \) levels, each of which requires \( O(n) \) comparisons: \( O(n^2) \) time complexity.
Which of these choices is a better choice for the pivot?

A. The first element in the list
B. A random element in the list
C. They are about the same