Announcements

1. Midterm review options:
   1. Review sheet coming soon!
   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

- Selection sort and Insertion sort
- Recursion review... moving toward Quicksort and Mergesort
Sorting Algorithms
the fruit flies of complexity theory... (Big-O)

Checksort
check permutations until sorted

List of length n

Check:
3 1 2
3 2 1
1 2 3
1 3 2
2 1 3
2 3 1

What is the best-case running time of checksort?
A. $O(1)$
B. $O(\log(n))$
C. $O(n)$
D. $O(n \log(n))$
E. $O(n^2)$
Sorting Algorithms

the *fruit flies* of complexity theory... (Big-O)

**Checks out**

check permutations until sorted

<table>
<thead>
<tr>
<th>List of length n</th>
<th>Check:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 2</td>
<td>3 1 2</td>
</tr>
<tr>
<td></td>
<td>3 2 1</td>
</tr>
<tr>
<td></td>
<td>1 2 3</td>
</tr>
<tr>
<td></td>
<td>1 3 2</td>
</tr>
<tr>
<td></td>
<td>2 1 3</td>
</tr>
<tr>
<td></td>
<td>2 3 1</td>
</tr>
</tbody>
</table>

What is the worst-case running time of checksort?

A. $O(1)$
B. $O(n \cdot \log(n))$
C. $O(n^2)$
D. $O(2^n)$
E. $O(n!)$
Sorting Algorithms

the fruit flies of complexity theory... (Big-O)

Checksort
check permutations until sorted

List of length $n$

Check:

3 1 2
3 2 1
1 2 3
1 3 2
2 1 3
2 3 1
Practical sorting algorithms: Selection sort

**Pseudocode:** selectionSort( comparableType [] array )

1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1
   // invariant: all elements from the first position to the last position of the
   // sorted part are in non-decreasing order
Selection Sort: The Picture

(a) Initial configuration for selection sort. The input array is logically split into an unsorted part and a sorted part (initially empty).

(b) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (first pass).

(c) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (second pass).

(d) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (third pass).
Pseudocode: selectionSort( comparableType [] array )
1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1
   // invariant: all elements from the first position to the last position of the
   // sorted part are in non-decreasing order

- Approximately How many times does the algorithm have to select the largest element from the unsorted part of the list? (i.e. how many times does the outer loop run?)
  A. 1 time
  B. N times
  C. N^2 times
Selection sort: Running time

- Approximately how many comparisons does it take to select the largest element from the unsorted part of the list?
  - A. 1 comparison
  - B. N-1 comparisons
  - C. At most N-1, but usually less than N-1

Pseudocode: selectionSort( comparableType [] array )
1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1
   // invariant: all elements from the first position to the last position of the sorted part are in non-decreasing order
Approximately how many comparisons does it take to select the largest element from the unsorted part of the list?

A. 1 comparison
B. N-1 comparisons
C. At most N-1, but usually less than N-1

We could overestimate use N, but we might not get the tightest bound possible.
Pseudocode: selectionSort( comparableType [] array )
1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1
   // invariant: all elements from the first position to the last position of the
   // sorted part are in non-decreasing order

1st iter  2nd iter  3rd iter
# comparisons:  (N-1)  +  (N-2)  +  (N-3)  +  …  +  2  +  1
Insertion sort: Pseudo code

Pseudocode: insertionSort ( primitiveType [] array )

1. while the size of the unsorted part is greater than 0
2. let the target element be the first element in the unsorted part
3. find target’s insertion point in the sorted part
4. insert the target in its final, sorted position
   // invariant: the elements from position 0 to (size of the sorted part – 1)
   // are in nondecreasing order

Pseudocode finding the insertion point for the target in the sorted part

1. get a copy of the first element in the unsorted part
2. while ( there are elements in the unsorted part to examine AND
   we haven’t found the insertion point for the target )
3. move the element up a position  // make room for the target
(a) Initial configuration for insertion sort. The input array is logically split into a sorted part (initially containing one element) and an unsorted part.

(b) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (first pass).

(c) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (second pass).

(d) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (third pass).
Insertion sort: Worst case analysis

- Approximately how many times must you insert the next element into the sorted part of the array? (Outer loop)
  A. 1
  B. N
  C. \( N^2 \)
Insertion sort: Worst case analysis

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

A. 1
B. N
C. N²
D. It depends on the length of the sorted part
Insertion sort: Worst case analysis

<table>
<thead>
<tr>
<th>1\textsuperscript{st} iter</th>
<th>2\textsuperscript{nd} iter</th>
<th>3\textsuperscript{rd} iter</th>
<th># comparisons:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>$+ \cdots + (N-2) + (N-1)$</td>
</tr>
</tbody>
</table>
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

2 4 9 13 15 17 19 20
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
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! ?

\[ 10 > 1, \text{ so we apply the second part of the definition} \]
\[ 10! = 10 \times 9! \]

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

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

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

\[ n! = \begin{cases} 1 & \text{if } n = 1, \\ n\cdot(n-1)! & \text{if } n > 1. \end{cases} \]

What is the **third** value printed when we call `factorial(10)`?

A. 2  
B. 3  
C. 7  
D. 8  
E. None of the above

Recursive code

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

Recursive definition

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

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.

```c
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?

- 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];
```
Recursion

Memory After

- Main and other methods
- add()  a,b,c

Example code

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

- Main and other methods
- mymethod()
  - x, xfac
- factorial()
  - n = 10
Recursion

Memory After

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

Memory After

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

Recursive code

```cpp
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>
</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

```java
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`
Consider this magical way of sorting lists:

Split the list in half:

Magically sort each list

Merge the two lists back together
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;
}
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 – 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 start, int end )
{
    int mid = start + ((end - start) / 2);
    mergeSort( toSort, start, mid );
    mergeSort( toSort, mid+1, end );
    merge( toSort, 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, start, mid );
        mergeSort( toSort, mid+1, end );
        merge( toSort, start, mid, end );
    }
}

This mergeSort works!!
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.
Merge Sort: An Example

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: An Example

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

How many merge steps are there?
A. \( \log(N) \)
B. \( N \)
C. \( N^2 \)
How many comparisons are made at each merge? (total)
A. \(\log(N)\)
B. \(N\)
C. \(N^2\)
D. Other

Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions
Running time for Mergesort:
log(N)*1 + log(N)*N = O(N*log(N))