# MERGESORT & QUICKSORT

cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

# administrivia...

-assignment 4 due tonight at midnight

-assignment 5 is out

-midterm next Tuesday

# last time...

# re·cur·sion

[ri-**kur**-*zhuh* n]
***noun***

see *recursion.*

-**recursion** is a problem solving technique in which the solution is defined in terms of a simpler (or smaller) version of the problem

    -break the problem into smaller parts

        -*solve the smaller problems*

        -*combine the results*

-a recursive method calls itself

-some functions are easiest to define recursively

```
sum(N) = sum(N-1) + N
```

-there must be at least on *base case* that can be computed without recursion

    -any recursive call must make progress towards the base case!

# a simple example

**sum(N) = sum(N-1) + N**

```java
public static int sum(int n) {
  if(n == 1)
    return 1;
  return sum(n-1) + n;
}
```

FIX TO HANDLE ZERO OR NEGATIVE VALUES...

HOW CAN WE SOLVE THE SAME PROBLEM WITHOUT RECURSION?

WHICH IS BETTER, THE RECURSIVE SOLUTION OR THE ALTERNATIVE?

-recursive methods often have unusual parameters

-at the top level, we just want:

```
binarySearch(arr, item);
```

-but in reality, we have to call:

```
binarySearch(arr, item, 0, arr.length-1);
```

-**driver methods** are wrappers for calling recursive methods

-driver makes the initial call to the recursive method, knowing what parameters to use

-is *not* recursive itself

```
public static boolean binarySearch(arr, item){
    return binarySearchRecursive(
      arr, item, 0, arr.length-1);
}
```

# recursion, beware

**-do not use recursion when a simple loop will do**
  -growth rates may be the same, but...
  -...there is a lot of overhead involved in setting up the method frame
    -*way more overhead than one iteration of a for-loop*

**-do not do redundant work in a recursive method**
  -move validity checks to a driver method

**-too many recursive calls will overflow the call stack**
  -stack stores state from all preceding calls

# 4 recursion rules

1. always have at least one case that can be solved without using recursion

2. any recursive call must progress toward a base case

3. always assume that the recursive call works, and use this assumption to design your algorithms

4. never duplicate work by solving the same instance of a problem in separate recursive calls

# today...

-mergesort

-quicksort

-midterm stuff

# mergesort
divide and conquer

# but first, merging...

-say we have two sorted lists, how can we efficiently merge them?

-**idea:** compare the first element in each list to each other, take the smallest and add to the merged list
   -AND… repeat

## WHAT DOES THIS LOOK LIKE?

# mergesort

1) divide the array in half

2) sort the left half

3) sort the right half

HOW DO WE SORT?
CAN WE AVOID SORTING? HOW?

4) merge the two halves together

WHAT IS MISSING HERE?

# mergesort

1) divide the array in half

2) ~~sort the left half~~

3) ~~sort the right half~~

4) merge the two halves together

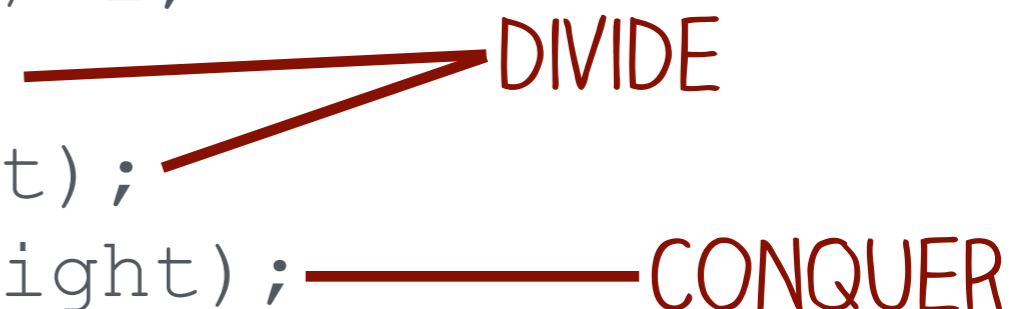2) take the left half, and go back to step 1   UNTIL???

3) take the right half, and go back to step 1   UNTIL???

## WHAT DOES THIS LOOK LIKE?

```
void mergesort(int[] arr, int left, int right)
{
  int mid = (left + right) / 2;
  mergsort(arr, left, mid);
  mergsort(arr, mid+1, right);
  merge(arr, left, mid+1, right);
}
```

DIVIDE

CONQUER
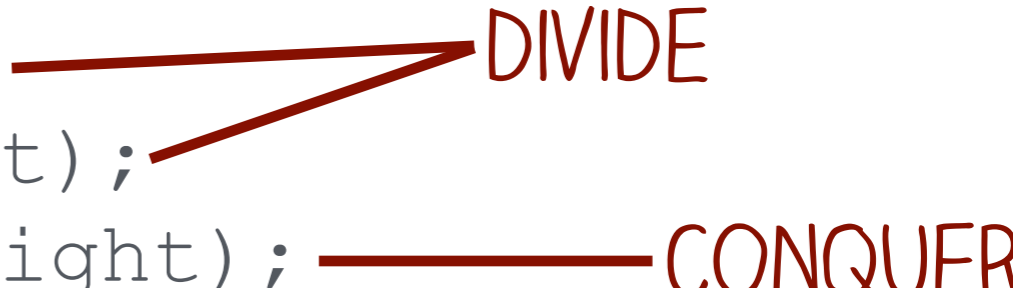
# WHAT ARE WE MISSING?

```
void mergesort(int[] arr, int left, int right)
{
  // arrays of size 1 are already sorted
  if(start >= end)
    return;

  int mid = (left + right) / 2;
  mergsort(arr, left, mid);        ──────── DIVIDE
  mergsort(arr, mid+1, right);     ────
  merge(arr, left, mid+1, right);  ──────── CONQUER
}
```

# WHAT IS THE COMPLEXITY OF THE DIVIDE STEP?

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) **N²**
F) **N³**

```
void mergesort(int[] arr, int left, int right)
{
  // arrays of size 1 are already sorted
  if(start >= end)
    return;

  int mid = (left + right) / 2;
  mergsort(arr, left, mid);          ──────────  DIVIDE
  mergsort(arr, mid+1, right);       ──────────
  merge(arr, left, mid+1, right);    ──────────  CONQUER
}
```

# WHAT IS THE COMPLEXITY OF THE CONQUER STEP?

A) $c$
B) $\log N$
C) $N$
D) $N \log N$
E) $N^2$
F) $N^3$

```
void mergesort(int[] arr, int left, int right)
{
  // arrays of size 1 are already sorted
  if(start >= end)
    return;

  int mid = (left + right) / 2;
  mergsort(arr, left, mid);          DIVIDE
  mergsort(arr, mid+1, right);
  merge(arr, left, mid+1, right);    CONQUER
}
```

# WHAT IS THE COMPLEXITY OF MERGESORT?

A) c            IS THIS THE WORST || AVERAGE || BEST-CASE?

B) log N

C) N

D) N log N

E) $N^2$

F) $N^3$

20

# merging sorted arrays

-easy concept, tricky code…

-lots of special cases:
- -keep track of two indices to step through both arrays (the "front" of each array)
    - *-indices do not necessarily move at the same speed*
- -have to stop the loop when either index reaches the end of their array
- -the two arrays are not necessarily the same size
- -what to do when you reach the end of one array but not the other?
- -copy from temp back into the array

```
void Merge(int[] arr, start, mid, end)
{
   // create temp array for holding merged arr
   int[] temp = new int[end - start + 1];

   int i1 = 0, i2 = mid;
   while(i1 < mid && i2 < end)
   {
      put smaller of arr[i1], arr[i2] into temp;
   }

   copy anything left over from larger half to temp;

   copy temp over to arr;
}
```

# IS THERE ANYTHING WE SHOULD DO DIFFERENT?

```
void Merge(int[] arr, int[] temp, start, mid, end)
{
   int i1 = 0, i2 = mid;
   while(i1 < mid && i2 < end)
   {
      put smaller of arr[i1], arr[i2] into temp;
   }

   copy anything left over from larger half to temp;

   copy temp over to arr;
}
```

# IS THERE ANYTHING WE SHOULD DO DIFFERENT?

# notes on merging

-the major disadvantage of mergesort is that the merging of two arrays requires an extra, temporary array

-this means that mergesort requires 2x as much space as the array itself

    -can be an issue if space is limited!

    -an *in-place* mergesort exists, but is complicated and has worse performance

-to achieve the overall running time of **O(N log N)** it is critical that the running time of the merge phase be linear

# mergesort variation

-it is a good idea to invoke insertion sort when the subarray size reaches a small enough threshold
   -why???
      -**HINT:** *what is the complexity of insertion sort? of mergesort? what are other runtime considerations?*

-the real threshold depends on several things:
   -hardware / OS / compiler
   -input characteristics

# quicksort
another divide and conquer

# quicksort

1) select an item in the array to be the **pivot**

2) **partition** the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) sort the left half

4) sort the right half

**NOTE: after partitioning, the pivot is in it's final position!**

WHAT DO YOU NOTICE?

# quicksort

1) select an item in the array to be the **pivot**

2) **partition** the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) ~~sort the left half~~

4) ~~sort the right half~~

3) take the left half, and go back to step 1   UNTIL???

4) take the right half, and go back to step 1   UNTIL???

## WHAT DOES THIS LOOK LIKE?

```
void quicksort(int[] arr, int left, int right)
{
  int pivot_index = partition(arr, left, right);
  quicksort(arr, left, pivot_index-1);
  quicksort(arr, pivot_index+1, right);
}
```

# WHAT ARE WE MISSING?

```
void quicksort(int[] arr, int left, int right)
{
   // arrays of size 1 are already sorted
   if(start >= end)
      return;

   int pivot_index = partition(arr, left, right);
   quicksort(arr, left, pivot_index-1);
   quicksort(arr, pivot_index+1, right);
}
```

# WHAT IS THE DIVIDE STEP?

# WHAT IS THE CONQUER STEP?

# quick note...

-a huge benefit of quicksort is that it can be done **in-place**

　　-ie. you can do the sort within the original array

-mergesort requires an extra, temporary array for merging

-however, in-place partitioning for quicksort requires some careful thought...

# in-place partitioning

1) select an item in the array to be the *pivot*

2) swap the pivot with the last item in the array *(just get it out of the way)*

3) step from left to right until we find an item > pivot
   *-this item needs to be on the **right** of the partition*

4) step from right to left until we find an item < pivot
   *-this item needs to be on the **left** of the partition*

5) swap items

6) continue until left and right stepping cross

7) swap pivot with left stepping item

## WHAT DOES THIS LOOK LIKE?

```
find pivot, swap with right_bound;

L = left_bound, R = right_bound - 1;
while(L <= R)
{
  if(arr[L] <= pivot)
  {
    L++; continue; // find next item > pivot
  }

  if(arr[R] >= pivot)
  {
    R--; continue; // find its "swapping partner"
  }

  swap(arr, L, R); // partners found, swap them
  L++; R--;
}

// point where L met R is the pivot location
swap(arr, L, right_bound); // put pivot back
```

# choosing a pivot

-the median of all array items is the best possible choice… why?
    -is time-consuming to compute
    -finding true median is **O(N)**

-it is important that we avoid the worst case
    -what IS the worst case(s)?

-middle array item is a safe choice… why?

-*median-of-three:* pick a few random items and take median
    -why not the first, middle, and last items?

-*random pivot:* faster than median-of-three, but lower quality

# choosing a pivot

-any nonrandom pivot selection has some devious input that causes $O(N^2)$

-trade-off between quality of pivot and time to select

-selection cost should always be $O(c)$
  -ie. it should not depend on $N$!

# quicksort complexity

-performance of quick sort heavily depends on which array item is chosen as the pivot

-**best case:** pivot partions the array into two equally-sized subarrays *at each stage* — $O(N \log N)$

-**worst case:** partition generates an empty subarray *at each stage* — $O(N^2)$

-**average case:** bound is $O(N \log N)$
   -proof is quite involved, see the textbook if you are curious

# quicksort vs mergesort

-both are **O(N log N)** in the average case

-mergesort is also **O(N log N)** in the *worst* case
  -so, why not always use mergesort?

-mergesort requires **2N** space
  -and, copying everything from the merged array
  back to the original takes time

-quicksort requires no extra space
  -thus, no copying overhead!
  -but, in **O(N²)** worst case <wha wha>

-both are divide and conquer algorithms (recursive)

-mergesort sorts "on the way up"
   -after the base case is reached, sorting is done as the calls return and merge

-quicksort sorts "on the way down"
   -once the base case is reached, that part of the array is sorted

-though quicksort is more popular, it is not always the right choice!

# sorting summary

| | Best | Average | Worst | Notes |
|---|---|---|---|---|
| **Selection Sort** | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | Never used in practice |
| **Insertion Sort** | $\mathbf{O(N)}$ | $O(N^2)$ | $O(N^2)$ | Takes advantage of "sortedness" |
| **Shellsort** | $O(N \log N)$ | $O(N^{1.25})$ | $O(N^{1.5})$ | Depends on gap sizes |
| **Mergesort** | $O(N \log N)$ | $\mathbf{O(N \log N)}$ | $\mathbf{O(N \log N)}$ | 2x space overhead, guaranteed $O(N \log N)$ |
| **Quicksort** | $O(N \log N)$ | $\mathbf{O(N \log N)}$ | $O(N^2)$ | Depends on pivot |

# midterm

# topics

-Java basics
   -variables, types
   -control flow
   -reference types
   -classes, methods

-OOP
   -inheritance
   -polymorphism
   -interfaces
   -`super`

# topics

-generics
   -wild cards
   -generic classes

-comparators

-algorithm analysis
   -growth rates
   -Big-O
   -determining complexity of loops and algorithms

# topics

-Collections

-Iterators

-recursion

-selection sort
-insertion sort
-shellsort
-mergesort
-quicksort

BE ABLE TO REASON ABOUT
PERFORMANCE AND BEHAVIOR OF EACH!

# test format

-problems may be of the following types:
    -short answer
    -determining output of code
    -writing code
    -filling in missing code
    -multiple choice
    -true / false

# next time…

-no lab on Monday

**-midterm on Tuesday in class**

-reading
  -chapter 17
  -chapter 3
    *-http://opendatastructures.org/ods-java/*

**-homework**
  -assignment 5 due Thursday