CSCI 241

Lecture 7

Runtime Analysis

Announcements

- No quiz today!

Happenings

Monday, 1/28 – CS Faculty Candidate: Research Talk – 4 pm in CF 316

Tuesday, 1/29 – CS Faculty Candidate: Teaching Talk – 4 pm in CF 316

Wednesday, 1/30 – Peer Lecture Series: BASH Workshop – 5 pm in CF 420

Thursday, 1/31 – Group Advising to Declare the Major – 3 pm in CF 420

Thursday 1/31 – CS Faculty Candidate: Research Talk – 4 pm in CF 226

Friday, 2/1 – CS Faculty Candidate: Teaching Talk – 4 pm in CF 226

Goals:

- Know the runtime complexity of the sorting algorithms we've covered.
- Understand the basics of analyzing the runtime of recursive algorithms.
- Gain experience counting operations and determining big-O runtime of simple iterative and recursive algorithms.

Runtime Analysis: Overview

- Why? We want a measure of performance where

  - it is **independent** of what computer we run it on.

    Solution: count **operations** instead of clock time.

  - Dependence on **problem size** is made explicit.

    Solution: express runtime as a function of **n**
    (or whatever variables define problem size)

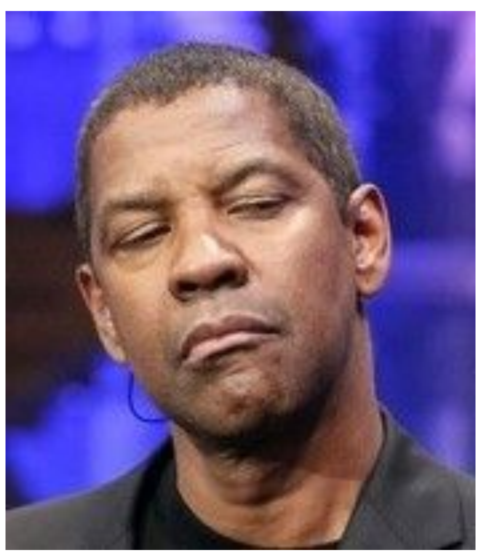
  - it is **simpler** than a raw count of operations and focuses on performance on **large problem sizes**.

    Solution: ignore constants, analyze **asymptotic** runtime.

Runtime Analysis: Overview

- How?

    1. Count the number of primitive (constant-time) operations that occur over the entire execution of the algorithm.



    2. Drop constants and lower-order terms to find the **asymptotic runtime class.**

Really? *any* constant?

A practical argument:

- My MacBook Pro from 2013: 3.17 **giga**FLOPs
- Fastest supercomputer as of June 2018: 200 **peta**FLOPs
- Supercomputer is 63,091,482 times faster.

Input interpretation:

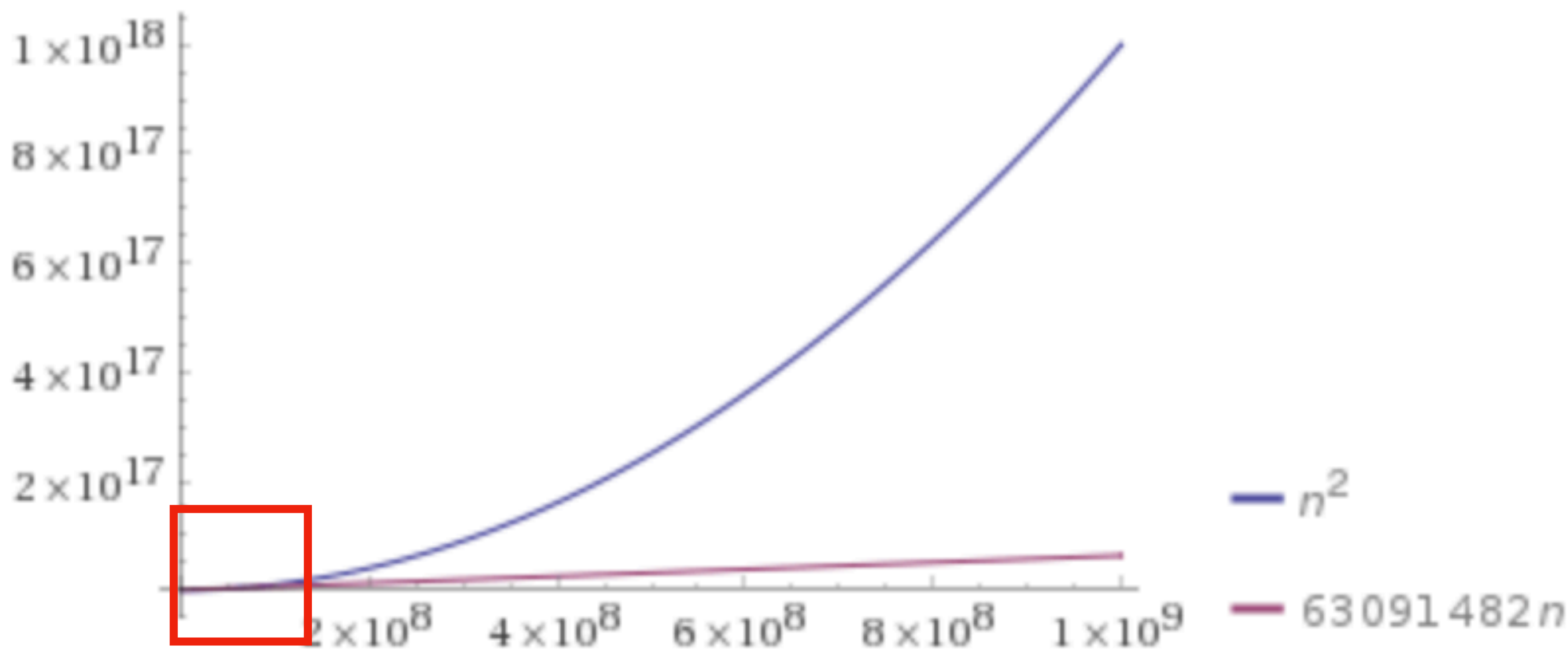
n² on a supercomputer

plot	n^2	$n = 0$ to $1\,000\,000\,000$
	$63\,091\,482\,n$	

n on my macbook

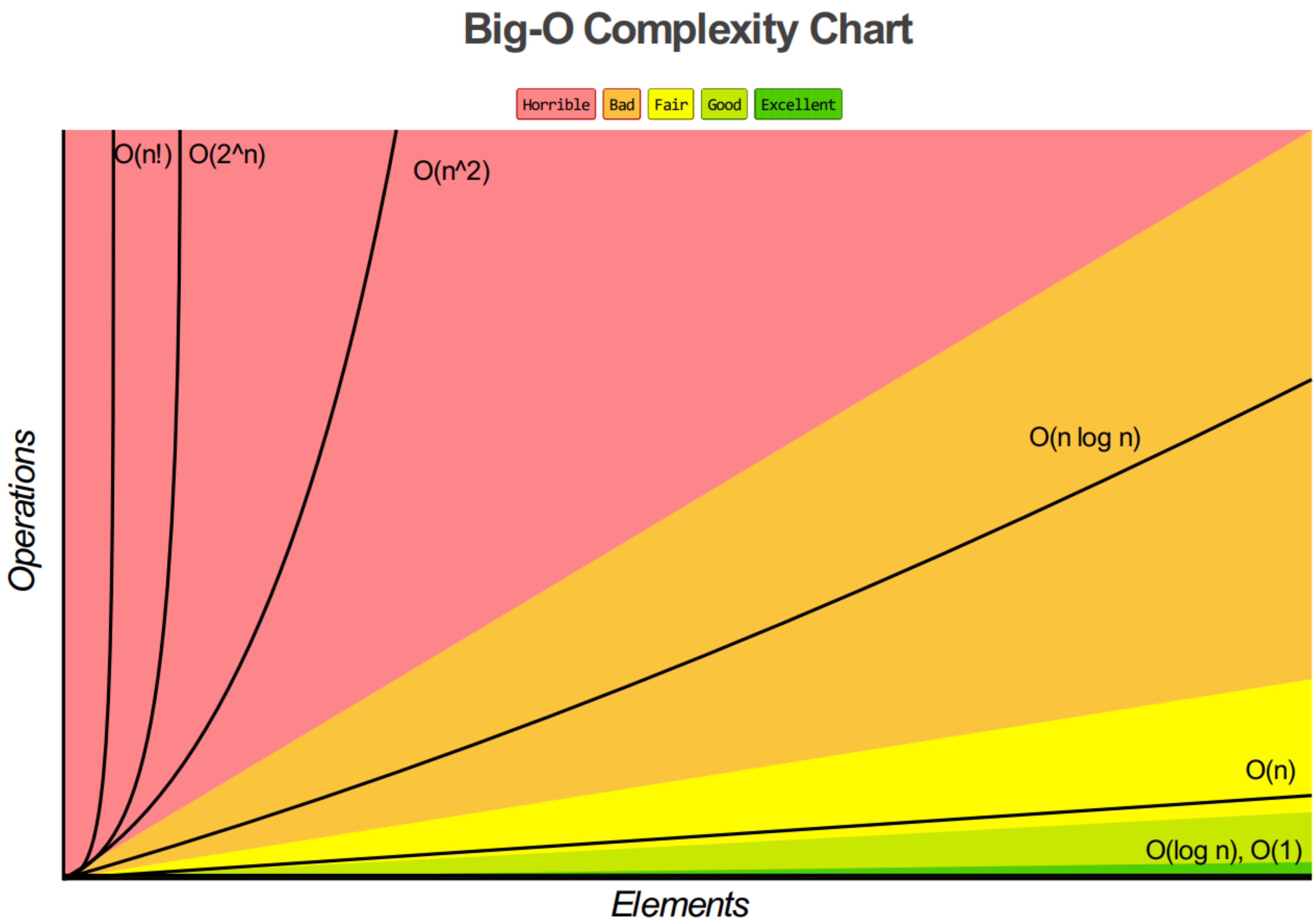
Enlarge | Data | Customize | Plaintext | Interactive

Plot:



n² algorithm may be faster here!

Common Complexities



Counting Operations

What’s a constant-time operation?

- Anything that **doesn’t** depend on the input size:
    - Reading/writing from/to a variable or array location.
    - Evaluating an arithmetic or boolean expression.
    - Returning from a method.

Counting Operations

What's a constant-time operation?

- Anything that doesn't depend on the input size:
  - Reading/writing from/to a variable or array location.
    `int i = 2; int b = 4; a[i] = b;`
  - Evaluating an arithmetic or boolean expression.
    `int i = 0; int j = i+4; int k = i*j;`
  - Returning from a method. `return k;`

Key intuition:

- These don't take identical amounts of time, but the times are within a **constant factor** of each other.
- Same for running the **same** operation on a **different** computer.

Counting Operations

What's **not** a constant-time operation?

- Anything that **does** depend on the input size, e.g.:
  - Looping over all values in an array of size n.
  - Recursively checking whether a string is a palindrome
  - Sorting an array
  - Most nontrivial algorithms / data structure operations we'll cover in this class.

Counting Operations

What happens when the number of times executed is variable / depends on the data?

- We have to specify whether we want worst-case, average-case (aka expected-case), or best-case runtime.

```
public int findMax(int[] a) {
  int currentMax = a[0];
  for (int i = 1; i < a.length; i++) {
    if (currentMax < a[i]) {
      currentMax = a[i];
    }
  }
}
```

times executed depends on contents of a!

Counting Operations

What happens when the number of times executed is variable / depends on the data?

- Worst-case is usually the important one, with notable exceptions for algorithms that beat asymptotically faster algorithms in practice.
- Quicksort is worst-case $O(n^2)$ but often beats MergeSort in practice

Counting Strategies:
1. Simple counting

```
/** A singly linked list node */
public class Node {
   int value;
   Node next;
   public Node(int v) {
      value = v;
   }
}
```

```
/** Insert val into the list in after pred.
 * Precondition: pred is not null */
public void addAfter(Node pred, int val) {
   Node newNode = new Node(val);          1
   new_node.next = pred.next;             1
   pred.next = newNode;                   1
}
```

Counting Strategies:
1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}

// is equivalent to:

int i = 0;               1
while (i < n) {          1 per iteration
   loopBody(i);          1 per iteration
   i++;                  1 per iteration
}
```

How many iterations?

i takes on values 0..n, of which there are n.

Counting Strategies:
1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}
```

Total runtime:

```
// is equivalent to:
```

1 + 2n + n*[runtime of loopBody]

```
int i = 0;              —— 1
while (i < n) {         —— n
   loopBody(i);         —— n * runtime of loopBody
   i++;                 —— n
}
```

How many iterations?

i takes on values 0..n, of which there are n.

Counting Strategies: 2. Aggregate Analysis

Not as easy case:

1. Identify all primitive operations

2. Trace through the algorithm, reasoning about the loop bounds in order to count the worst-case number of times each operation happens.

Counting Strategies: 2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] < A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

Invariant: A

	sorted	i ?

AT MOST How many times do we call swap() during iteration i?

Counting Strategies: 2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] < A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

Invariant: A

	i
sorted	?

AT MOST How many times do we call swap() during iteration i?

j begins at i and could go as far as 1: that's as many as i swaps at iteration i

Number of swaps: 1 in 1st iteration + 2 in 2nd iteration + … + n in nth iteration

1 + 2 + 3 + … + n-1 + n = (n * (n-1)) / 2 = (n^2 - n) / 2

Counting Strategies: 2. Aggregate Analysis

```
// Sorts A using insertion sort
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] < A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

AT MOST How many times do we call swap() during iteration i?

j begins at i and could go as far as 1: that's as many as i swaps at iteration i

Number of swaps: 1 in 1st iteration + 2 in 2nd iteration + … + n in nth iteration

$1 + 2 + 3 + \ldots + n-1 + n = (n * (n-1)) / 2 = (n^2 - n) / 2$

$(n^2 - n)/2 \Rightarrow n^2 / 2 - n / 2 \Rightarrow n^2 - n \Rightarrow O(n^2)$

What about recursion?

Much like loops:

1. How much work is actually done per call?
2. How many calls are made?
   - This is simpler when the work per call is the same.
   - Sometimes the work per call depends on n.

Operation Counting in Recursive Methods: Example

```
/** Prints the linked list starting at head */
printList(Node head):

  if head != null:
    print(head)
    printList(head.next)
```

Runtime Analysis: MergeSort

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return                        O(1)
  mid = (end-start)/2  O(1)

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

  merge(A, start, mid, end) O(??)
```

Runtime Analysis: MergeSort

```
/** 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)
```

O(1)

O(1)

O(?)

O(?)

O(??)

1. How much work is actually done per call?

Merge step

```
merge(A, start, mid, end):
  B = a deep copy of A
  i = start
  j = mid
  k = 0
  while i < mid and j < end:
    if B[i] < B[j]:
      A[k] = B[i]
      i++
    else:
      A[k] = B[j]
      j++
    k++

  while i < mid:
    A[k] = B[i]
    i++, k++

  while j < end:
    A[k] = B[j]
    j++, k++
```

O(1)

O(1)

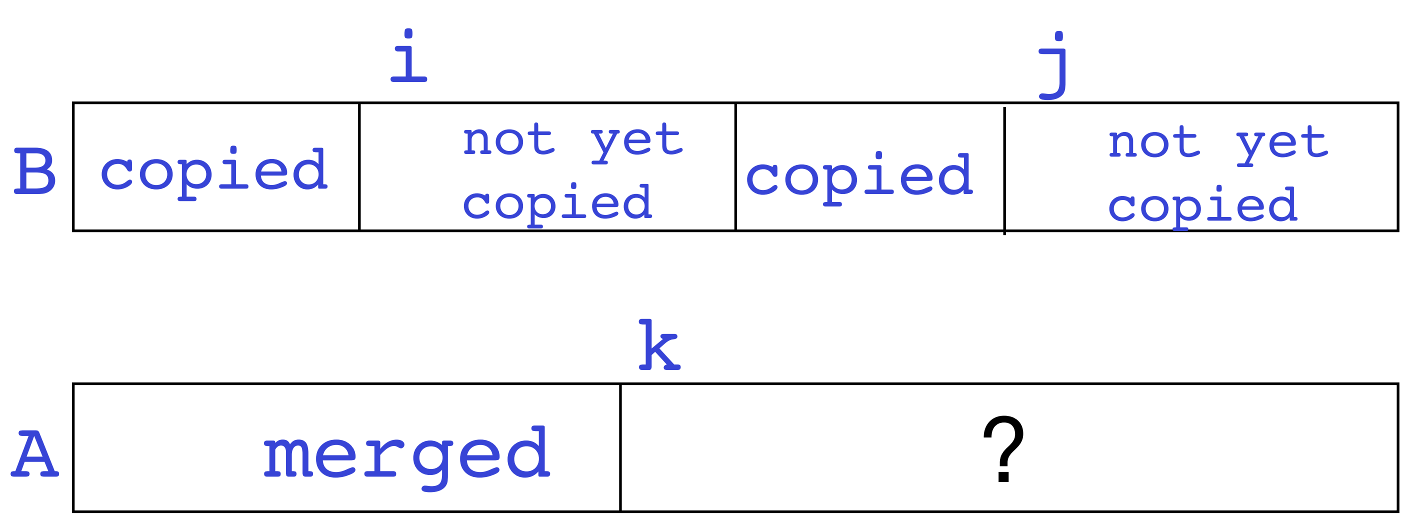
O(1)

Smaller thing goes first

Ran out of things in one list or the other

Copy remaining things from nonempty half

Inv



Merge step

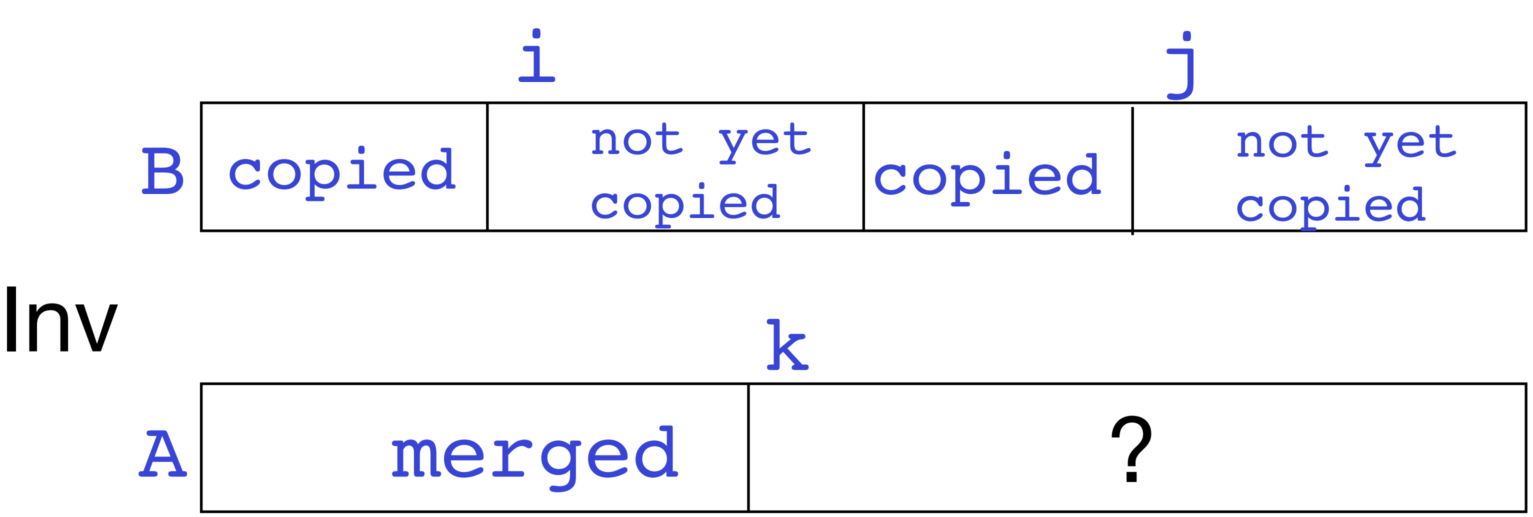
```
merge(A, start, mid, end):
  B = a deep copy of A
  i = start
  j = mid
  k = 0
  while i < mid and j < end:
    if B[i] < B[j]:
      A[k] = B[i]
      i++
    else:
      A[k] = B[j]
      j++
    k++

  while i < mid:
    A[k] = B[i]
    i++, k++

  while j < end:
    A[k] = B[j]
    j++, k++
```

O(1)

Smaller thing goes first

Ran out of things in one list or the other

Copy remaining things from nonempty half



Merge step

```
merge(A, start, mid, end):
  B = a deep copy of A
  i = start
  j = mid
  k = 0
  while i < mid and j < end:
    if B[i] < B[j]:
      A[k] = B[i]
      i++
    else:
      A[k] = B[j]
      j++
    k++

  while i < mid:
    A[k] = B[i]
    i++, k++

  while j < end:
    A[k] = B[j]
    j++, k++
```

at most n
=> O(n) iterations

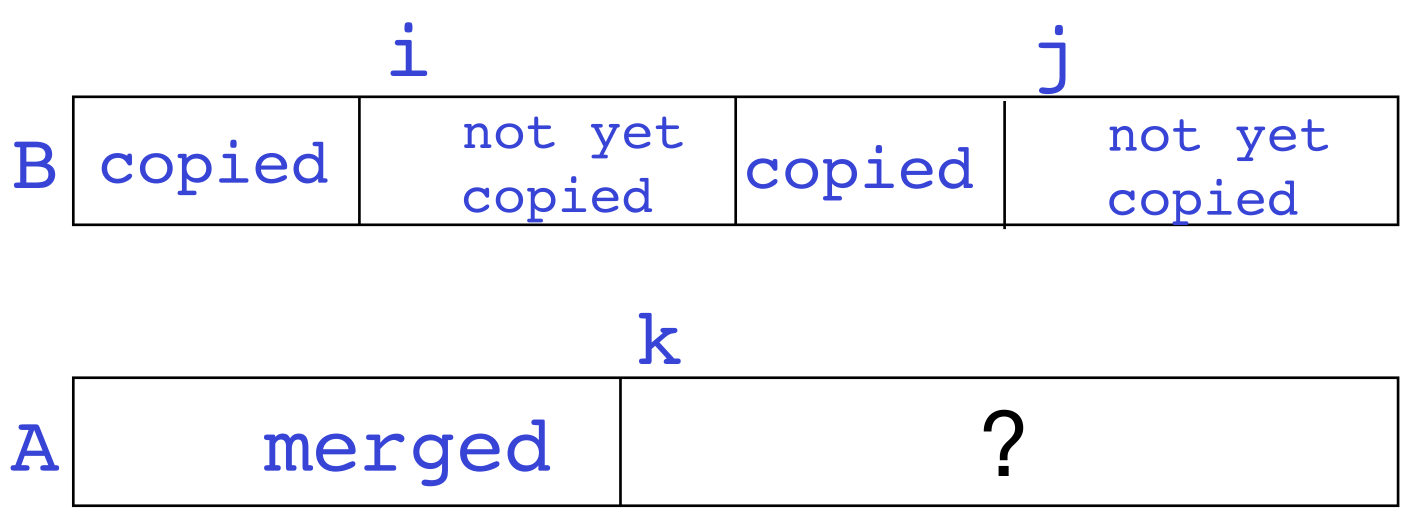
O(1)

Smaller thing goes first

Ran out of things in one list or the other

Copy remaining things from nonempty half

Inv



Merge step

```
merge(A, start, mid, end):
  B = a deep copy of A
  i = start
  j = mid
  k = 0
  while i < mid and j < end:
    if B[i] < B[j]:
      A[k] = B[i]
      i++
    else:
      A[k] = B[j]
      j++
    k++

  while i < mid:
    A[k] = B[i]
    i++, k++

  while j < end:
    A[k] = B[j]
    j++, k++
```

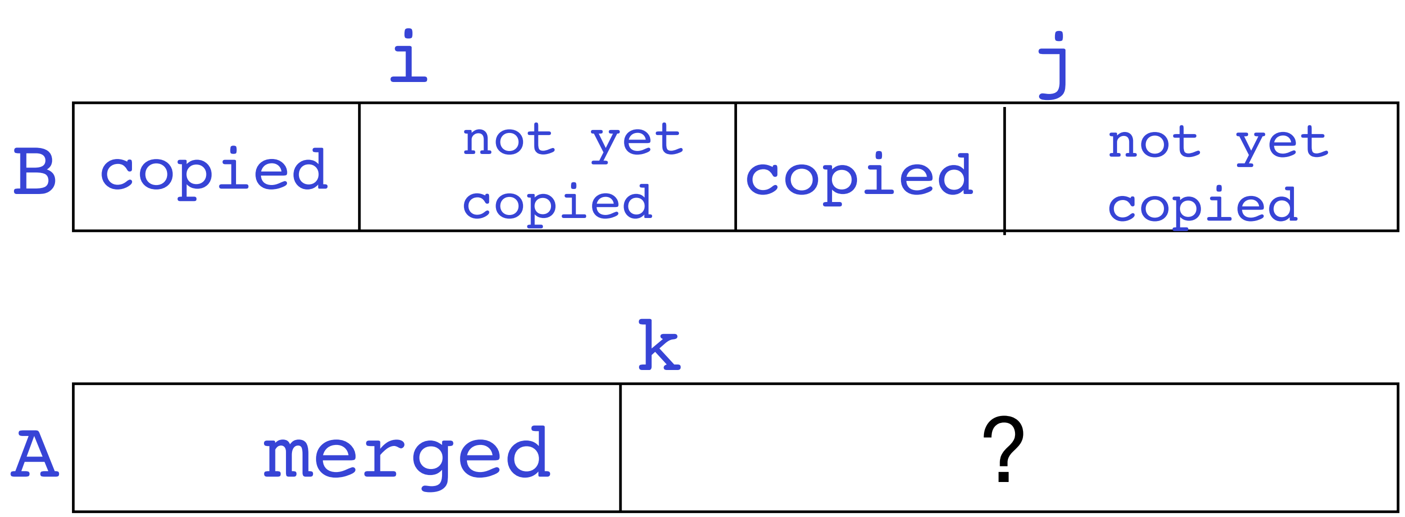
O(n)

Smaller thing goes first

Ran out of things in one list or the other

Copy remaining things from nonempty half

Inv



Merge step

```
merge(A, start, mid, end):
  B = a deep copy of A
  i = start
  j = mid
  k = 0
  while i < mid and j < end:
    if B[i] < B[j]:
      A[k] = B[i]
      i++
    else:
      A[k] = B[j]
      j++
    k++
```

O(n)

Smaller thing goes first

by a similar argument:

Ran out of things in one list or the other

```
  while i < mid:
    A[k] = B[i]
    i++, k++
```

at most n/2 => O(n)

Copy remaining things from nonempty half

```
  while j < end:
    A[k] = B[j]
    j++, k++
```

at most n/2 => O(n)

Runtime Analysis: MergeSort

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):          O(1)
    return
  mid = (end-start)/2         O(1)

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

  merge(A, start, mid, end)   O(??)
```

1. How much work is actually done per call?

Runtime Analysis: MergeSort

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return                  O(1)
  mid = (end-start)/2       O(1)

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

  merge(A, start, mid, end) O(n)
```

1. How much work is actually done per call?

Runtime Analysis: MergeSort

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

O(1)

O(1)

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

O(?)

O(?)

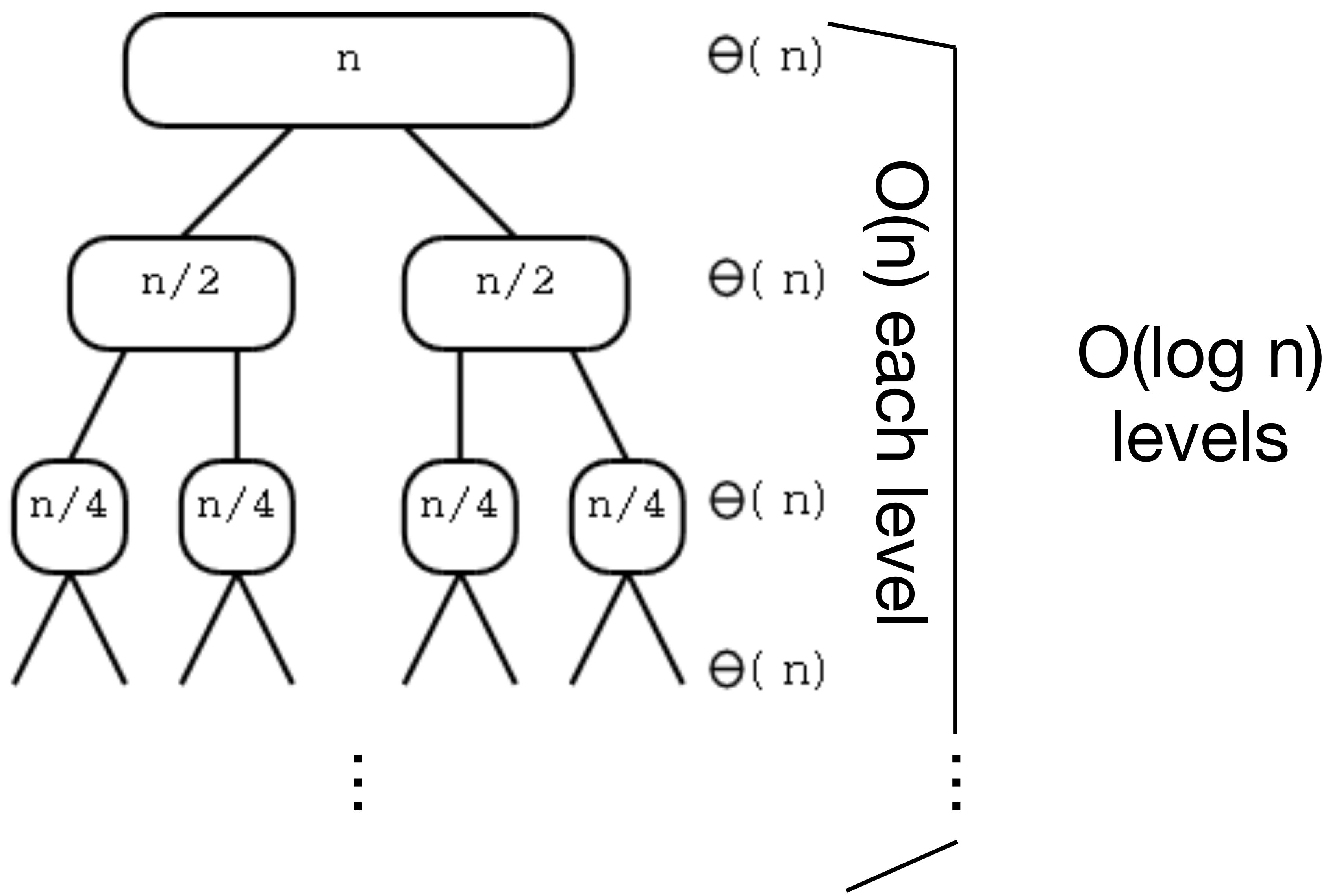
```
  merge(A, start, mid, end)
```

O(n)

2. How How many calls are made?

How many times can we divide n by 2 before we hit 1?

$$n/2^x = 1$$
$$n = 2^x$$
$$x = \log_2 n$$



Runtime Analysis: Quicksort

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):          O(1)
    return

  mid = partition(A,st,end) O(n)

  quickSort(A,st,mid)
  quickSort(A,mid, end)      ??
```

Runtime Analysis: Quicksort

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)
  quickSort(A,mid, end)
```

O(1)

O(n)

??

If pivot splits array approximately in half each time, O(log n) levels of recursion just like mergesort.

average (expected) case

If pivot is the min or max each time, O(n) levels of recursion, for a total runtime of $O(n^2)$!

worst case