CSCI 241

Lecture 9

Runtime of Quick, Merge, and Radix

How's A1 going?

Announcements

- Feedback survey out, please submit by Monday
- Quiz today: same as usual
- In-class problems for today are posted on the course webpage (schedule table, 4/24): https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci241_20s/lectures/L09/sort_runtimes.html
    - Please pull them up so you can refer to them while in breakout rooms.

Goals

- Get more practice analyzing runtimes.
- Know how logarithms end up in runtime counts.
- Know the runtime complexity of all the sorting algorithms we've covered.

Asymptotic Runtime Class

or, "big-O" runtime

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**.

- Tells us how the runtime **grows** as the input size grows.

- Doesn't tell us *anything* about runtime when the input is small!

0. Warmup: What's the runtime of `mins`?

$\to \{ O(n^2), O(n) \}$

```
/** Return the max value in A[start..end] */
public int findMax(int[] a, int start, int end) {
  int currentMax = a[start];   O(1)
  for (int i = start+1; i < end; i++) {
    if (currentMax < a[i]) { O(1)
      currentMax = a[i]; O(1)
    }
  }
  return currentMax;
}
```

O(1)

O(1)

end - (start + 1) = n

O(n)

$1+2+3+\dots+49$

max

```
/** Print the min of several subarrays of A
 * Precondition: A.length >= 50. */
public static void mins(A) {
  for (int i = 1; i < 50; i++) {
    System.out.println(findMin(A, 0, i));
  }
}
```

Max

n = A.length

49

0..1 → 1
0..2 → 2
⋮
0..49 → 49

2. Something new...

```
public int f(int n) {
  while (n > 0) {
    System.out.println(n);
    n = n/2;
  }
}
```

$\log_2 n$

$O(1)$

18
9
4
2
1

128
64
32
16
8
4
2
1

How many times can n be divided by 2 before becoming 0?

$\frac{n}{2^x} = 1$

$n = 2^x$

$\boxed{\log_2 n = x}$

$\boxed{O(\log n)}$

$\log_2 n = k \log_{10} n$

Recursive methods:

1. How much work is actually done per call?
   not counting the recursive calls

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.

Runtime Analysis: MergeSort

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

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

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

O(1)

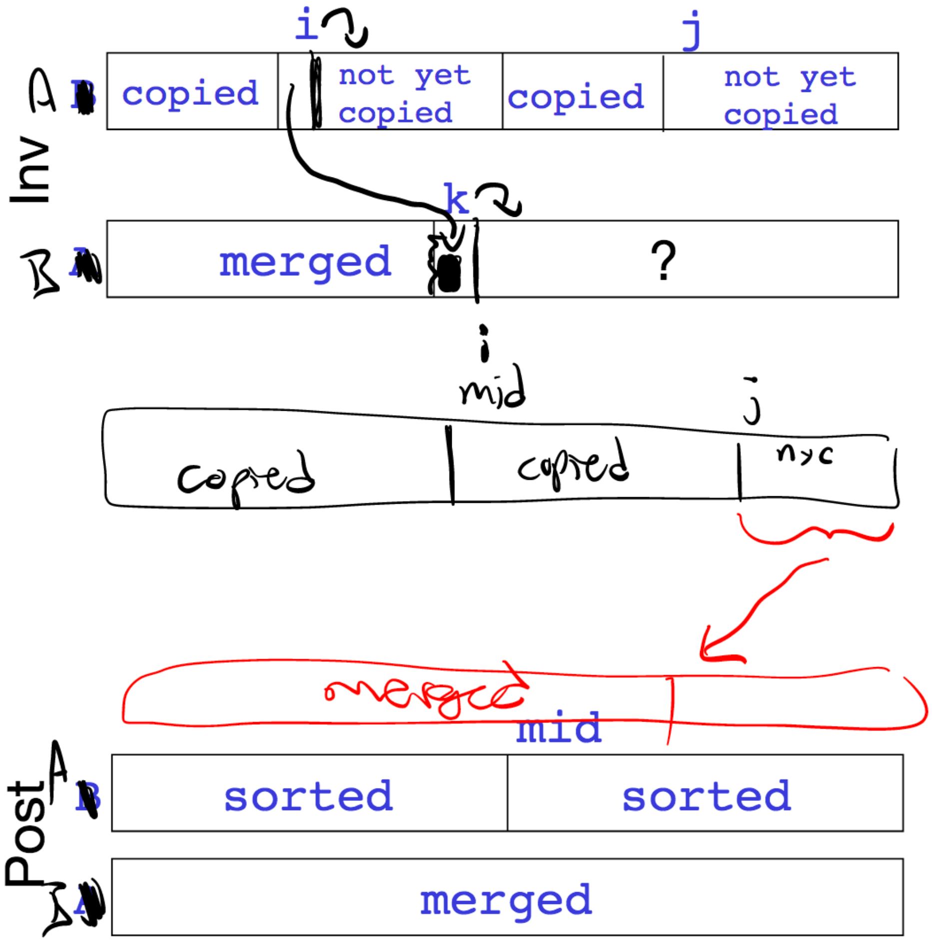
O(1)

O(1)

O(?)

1. How much work is actually done per call?

Merge step: Loop Invariant



set i, j

make new array A

while neither uncopied section is empty:

- copy smaller of A[i], A[j] into B[k]
- increment i, k
- or j

copy remaining values from left or right half

2. Runtime of merge

```
initialize i, j
B = deep copy of A
while neither uncopied segment is empty:
    copy the smaller of B[i], B[j]
         into A[k]
    increment i or j
    increment k
while one uncopied segment is empty:
    copy the next element in the nonempty
        segment into A[k]
    increment i or j
    increment k
```

$1 + O(n) + O(n) + O(n) = \cancel{3}\, \boxed{O(n)}$

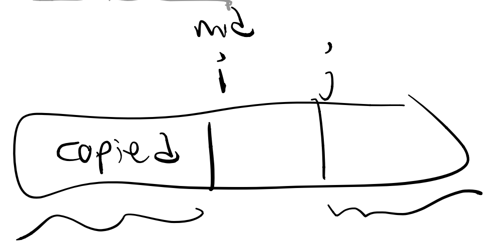
2. Runtime of merge

```
initialize i, j                                         O(1)
B = deep copy of A                                      O(n)
while neither uncopied segment is empty:                } O(n)
    copy the smaller of B[i], B[j]          O(1)  }
        into A[k]                                 } O(1)
    increment i or j                        O(1)  }
    increment k                             O(1)  }
while one uncopied segment is empty:                    } Σ^n O(n)
    copy the next element in the nonempty
       segment into A[k]              O(1)  }
    increment i or j                  O(1)  } O(1)
    increment k                       O(1)  }
```

	i		j	
B	copied	not yet copied	copied	not yet copied

Invariant

	k	
A	merged	?



Runtime Analysis: MergeSort

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
   if (end-start < 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(n)
```

1. How much work is actually done per call? $O(n)$

Runtime Analysis: MergeSort

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (end-start < 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?

$O(n \log n)$

How many calls to mergesort?