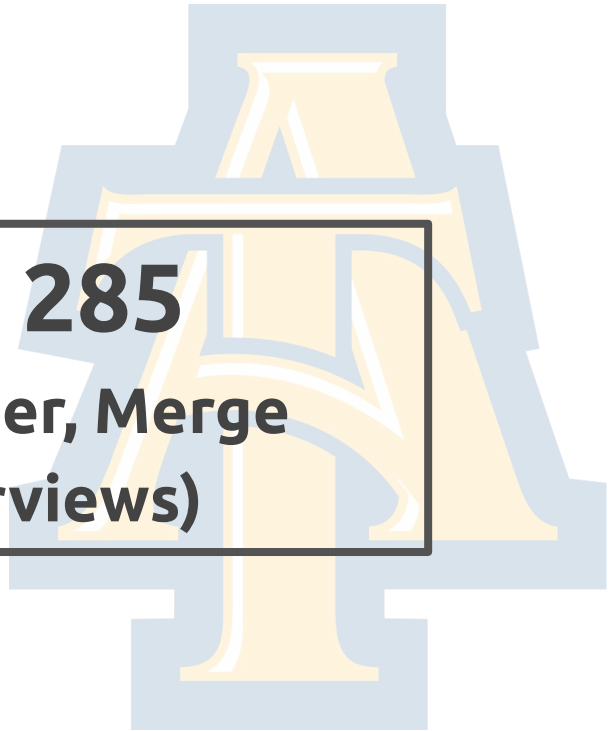
COMP - 285

Advanced Algorithms

Welcome to COMP 285

Lecture 7: Divide and Conquer, Merge Sort + CS Job Hunting (Interviews)



Lecturer: Chris Lucas (cflucas@ncat.edu)

HW2 was released!

Due 09/15 @ 11:59PM ET

HW2 was released!

With video walkthrough! (pilot)

HW1 grades by EoW!

Solutions will be released as well.

Career Opportunities!

CAREERS

- Blizzard Entertainment SWE internship
- Nintendo IT Software Internship
- DoD Counterintelligence internship *not necessarily technical*
- Siemens cybersecurity internship

www.comp285-fall22.ml

piazza

CAREERS

- Grant Thornton (Cybersecurity, specifically mentions ethical hacking!) link
- MLB Cybersecurity internship link
- IBM hardware engineering internship link

Quiz!

www.comp285-fall22.ml

Recall where we ended last lecture...

Sorting

When comparing different sorting algorithms, these are some of the properties we care about:

- **Best-case/worst-case/average-case time complexity**
- **In-place: can we use only O(1) additional space?**
- **Adaptive: does it run faster if the array is partially sorted?**
- **Stable: will elements of the same value stay ordered relative to each other?**

Selection Sort!!

Selection Sort Pseudocode

```
algorithm selectionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 0, 1, 2, ..., N-2
    min_index = i
    for j = i+1, i+2, ..., N-1
      if vec[j] < vec[min_index]
        min_index = j
    temp = vec[i]
    vec[i] = vec[min_index]
    vec[min_index] = temp
```

What's the tight upper-bound on the:

- Best-case runtime? $O(n^2)$
- Worst-case runtime? $O(n^2)$
- Average-case runtime? $O(n^2)$
- Worst-case space complexity? $O(1)$

Is this adaptive? No

https://visualgo.net/en/sorting

Insertion Sort!!

Insertion Sort In-Place Pseudocode

```
algorithm insertionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 1, 2, ..., N-1
    next = vec[i]
    j = i-1
    while j >= 0 and vec[j] > next
      vec[j+1] = vec[j]
      j = j-1
    vec[j+1] = next
```

What's the tight upper-bound on the:

- Best-case runtime? O(n)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? $O(n^2)$
- Worst-case space complexity? O(1

Is this adaptive? Yes

https://visualgo.net/en/sorting

QuickSort!!

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

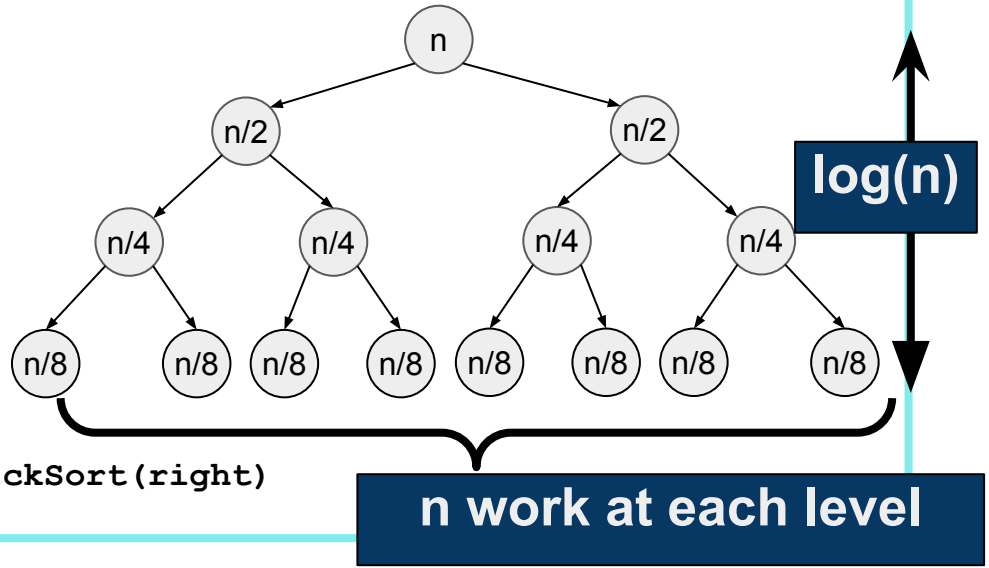
- Pick the last element in the list. (Pivot)
- Put the rest of the elements into two partitions (lists)
  - "all elements <= pivot"
  - "all elements > pivot"
- Then do the same steps on the two partitions until the lists are small enough to not need sorting. (recursion!)
- Once we've sorted the smaller lists, glue them back together at each level, along with the pivot.

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)

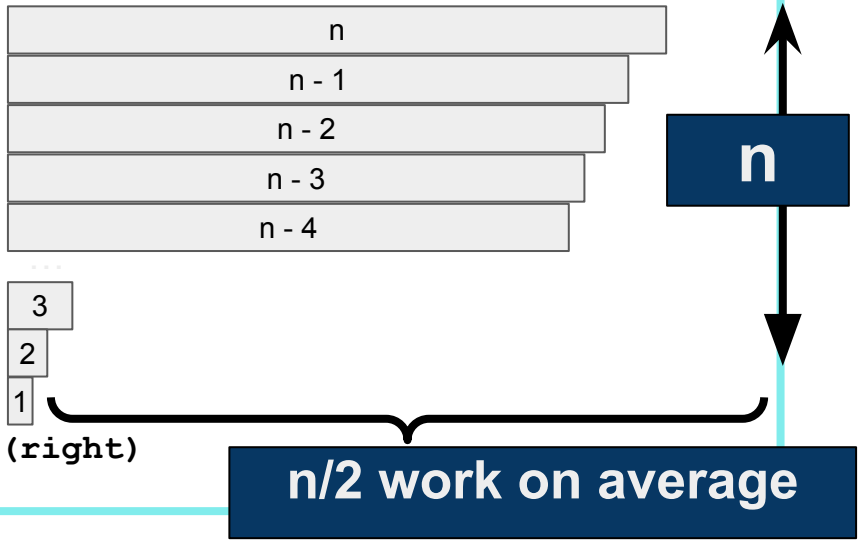


QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? $O(n^2)$



QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? O(nlogn)
- Worst-case space complexity?$O(n^2)$

Is it adaptive? NO

Big Questions!

- What is divide and conquer (technically)?
- What about MergeSort?
- How to pass behavioral interviews?

Big Questions!

- What is divide and conquer (technically)?
- What about MergeSort?
- How to pass behavioral interviews?

Divide & Conquer

Divide & Conquer is a pattern in recursive algorithms that has two defining characteristics:

1. At each step, there are 2 or more recursive calls
2. The problem is being reduced by some multiplicative factor at each call

If there is only 1 recursive call, then it's called **decrease and conquer**.

What is this function?

```
algorithm doSomePrinting
  input: positive integer n
  output: prints n, floor(n/2), floor(n/4), floor(n/8), ..., 1

  print n
  if n > 0
    doSomePrinting(floor(n/2))
```

Poll Options

1. **Decrease & Conquer**
2. **Divide & Conquer**
3. **Neither**

Divide & Conquer is a pattern in recursive algorithms that has two defining characteristics:

1. At each step, there are 2 or more recursive calls
2. The problem is being reduced by some multiplicative factor at each call

If there is only 1 recursive call, then it's called decrease and conquer.

What is this function?

```
algorithm createTwoHalves
  input: positive integer n
  output: ??
  if n == 1
     return 1
  return createTwoHalves(n/2)
         + createTwoHalves(n/2)
```

Poll Options

1. **Decrease & Conquer**
2. **Divide & Conquer**
3. **Neither**

Divide & Conquer is a pattern in recursive algorithms that has two defining characteristics:

1. At each step, there are 2 or more recursive calls
2. The problem is being reduced by some multiplicative factor at each call

If there is only 1 recursive call, then it's called decrease and conquer.

Kahoot!

www.kahoot.it, Code: 107 406

Enter your @aggies.ncat email

What is this function?

```
algorithm fib
  input: integer n >= 0
  output: nth fibonacci number
  if n = 0 || n == 1:
    return n
  else
    return fib(n-1) + fib(n-2)
```

Poll Options

1. **Decrease & Conquer**
2. **Divide & Conquer**
3. **Neither**

Divide & Conquer is a pattern in recursive algorithms that has two defining characteristics:

1. At each step, there are 2 or more recursive calls
2. The problem is being reduced by some multiplicative factor at each call

If there is only 1 recursive call, then it's called decrease and conquer.

What is this function?

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

Divide & Conquer is a pattern in recursive algorithms that has two defining characteristics:

1. At each step, there are 2 or more recursive calls
2. The problem is being reduced by some multiplicative factor at each call

If there is only 1 recursive call, then it's called decrease and conquer.

Poll Options

1. **Decrease & Conquer**
2. **Divide & Conquer**
3. **Neither**

What is this function?

```
algorithm binarySearchHelper
  Input: sorted vector<int> vec, integer target x, left index a, and right
index b
  Output: index of x in vec if it exists, -1 otherwise

  if a > b
    return -1
  midpoint = floor((b + a)/ 2)
  if vec[midpoint] == x
    return midpoint
  else if vec[midpoint] < x
    return binarySearchHelper(vec, x, midpoint+1, b)
  else
    return binarySearchHelper(vec, x, a, midpoint-1)
```

Divide & Conquer is a pattern in recursive algorithms that has two defining characteristics:

1. At each step, there are 2 or more recursive calls
2. The problem is being reduced by some multiplicative factor at each call

If there is only 1 recursive call, then it's called decrease and conquer.

Poll Options

1. **Decrease & Conquer**
2. **Divide & Conquer**
3. **Neither**

Big Questions!

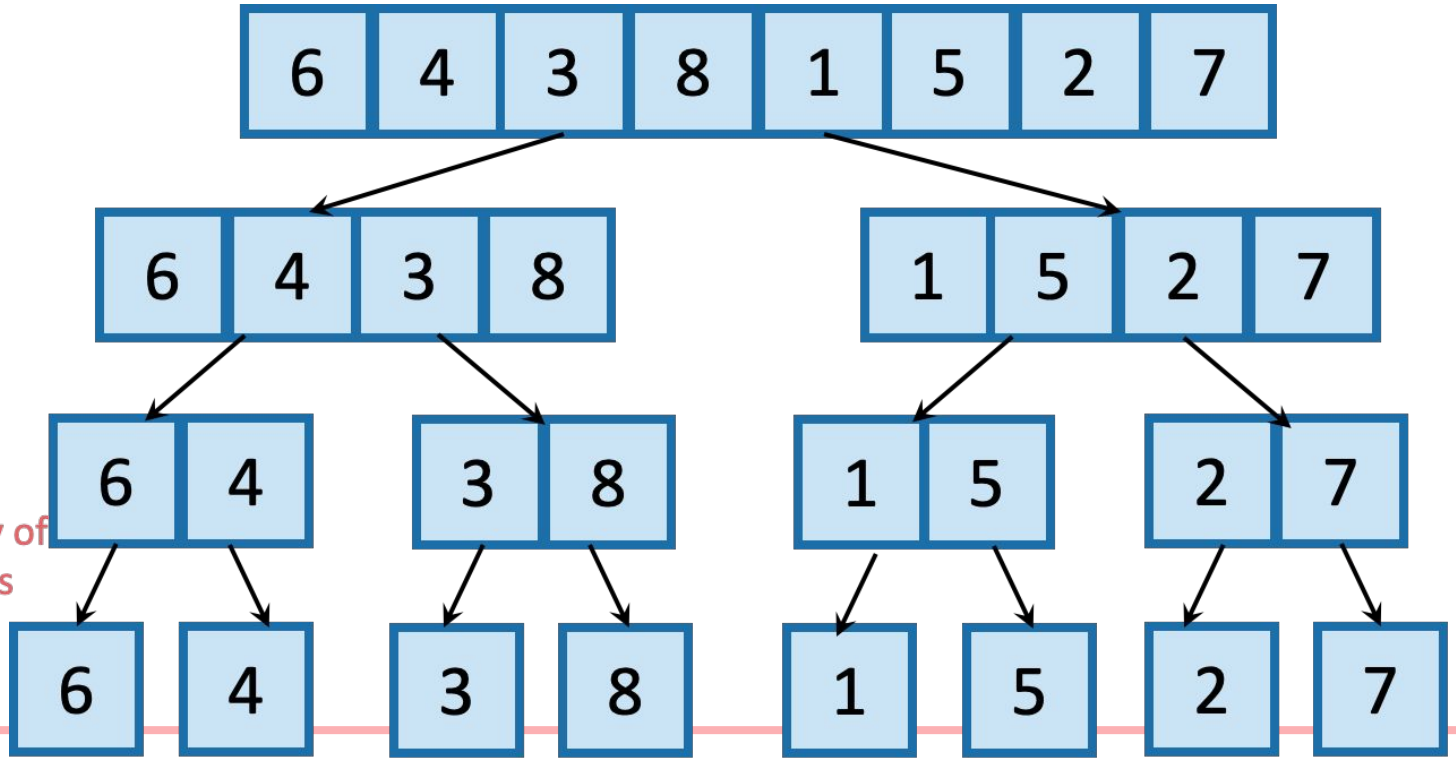
- What is divide and conquer (technically)?
- What about MergeSort?
- How to pass behavioral interviews?

MergeSort!!

MergeSort Intuition

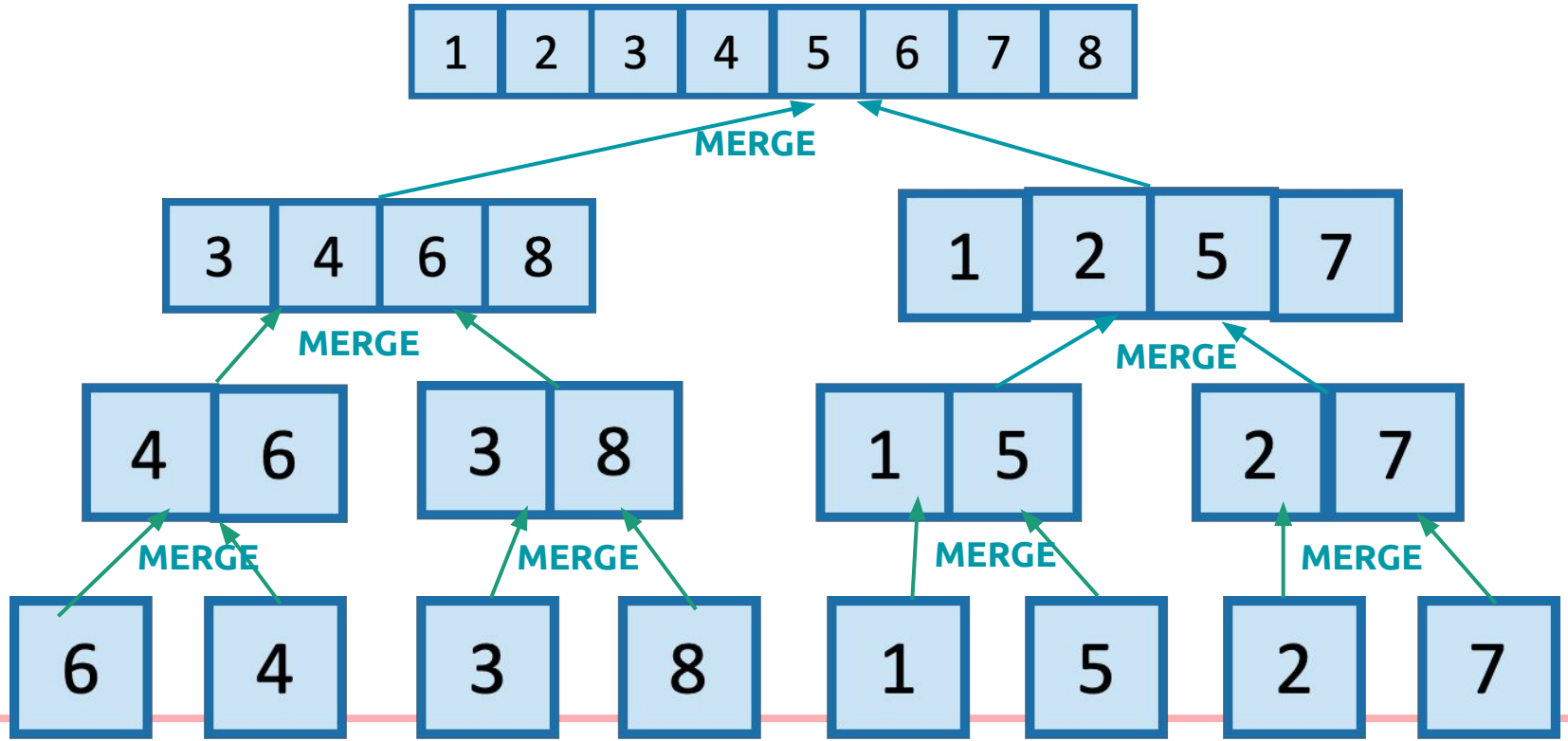
- **Base case:** If the length of the vec is size 1 or smaller, then the vec is already sorted.
- **Recursive calls:** Divide the vec into left and right halves and recursively sort the left and right half.
- **Solution building:** Once you have the sorted left and sorted right, merge them together into one big sorted vec.

What actually happens?
First, we recurse all the way down to base cases.



This array of length 1 is sorted!

Then we merge on our way back up!



A bunch of sorted lists of length 1 (in the order of the original sequence).

Merge Sort Pseudocode

```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec with its elements in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```

Base case: If the length of the vec is size 1 or smaller, then the vec is already sorted.

Recursive calls: Divide the vec into left and right halves and recursively sort the left and right half.

Solution building: Once you have the sorted left and sorted right, merge them together into one big sorted vec.

Merge Sort Pseudocode

```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec with its elements in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```

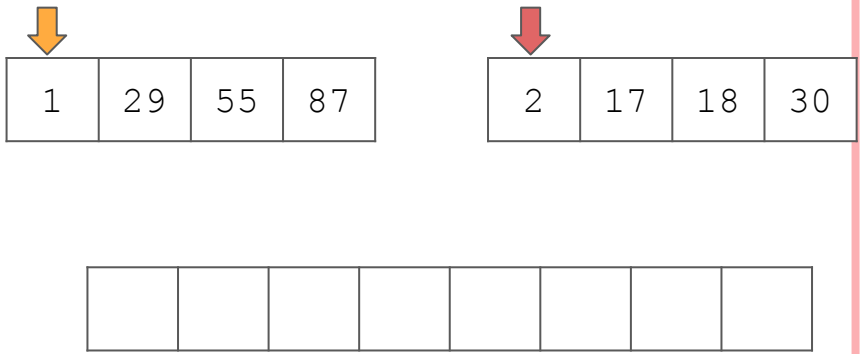
Base case: If the length of the vec is size 1 or smaller, then the vec is already sorted.

Recursive calls: Divide the vec into left and right halves and recursively sort the left and right half.

Solution building: Once you have the sorted left and sorted right, merge them together into one big sorted vec.

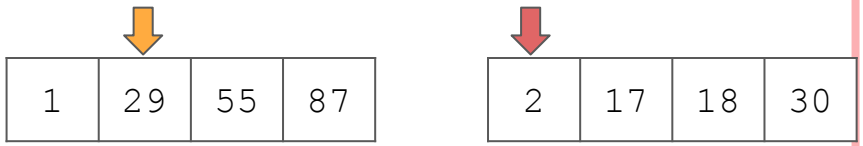
Merge Sort Pseudocode

```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```



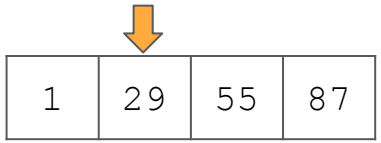
Merge Sort Pseudocode

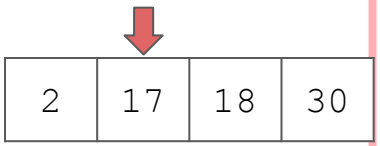
```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```



Merge Sort Pseudocode

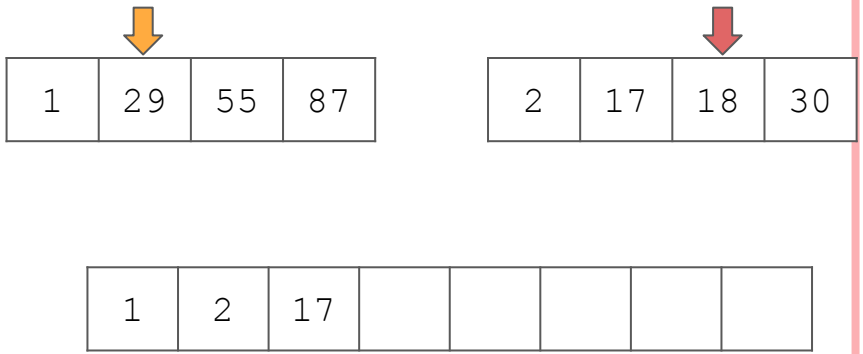
```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```





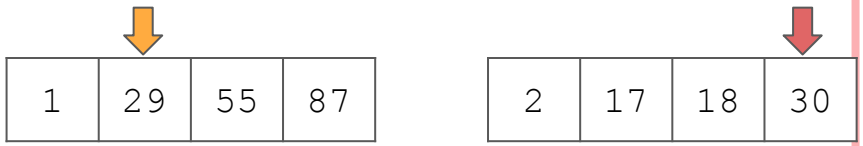
Merge Sort Pseudocode

```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```



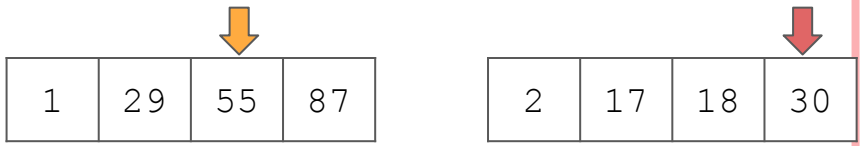
Merge Sort Pseudocode

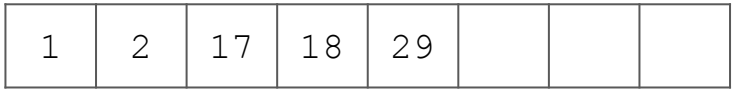
```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```



Merge Sort Pseudocode

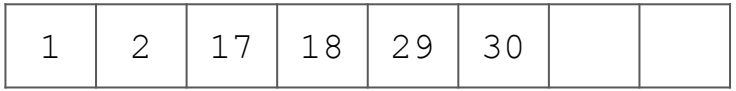
```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```





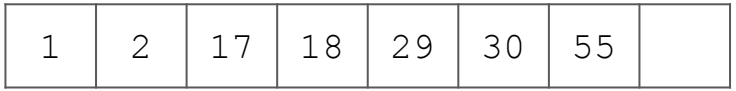
Merge Sort Pseudocode

```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```



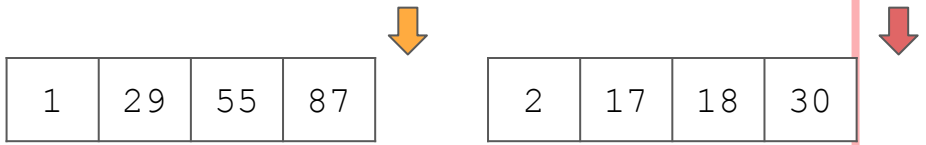
Merge Sort Pseudocode

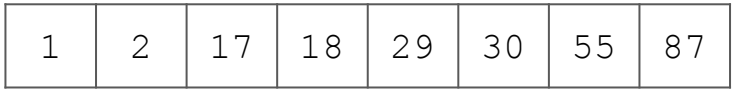
```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```



Merge Sort Pseudocode

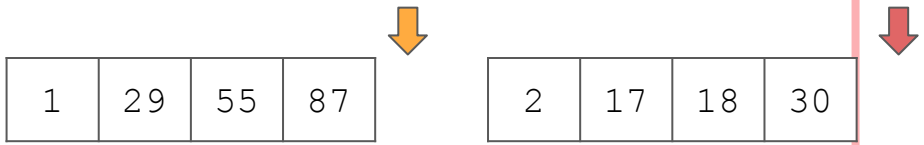
```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```

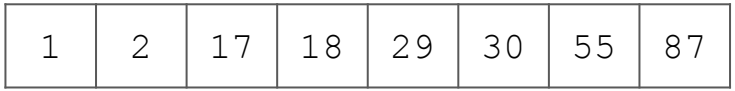




Merge Sort Pseudocode

```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```





Merge Sort Pseudocode

```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```

- Runtime:
- Space complexity:

Merge Sort Pseudocode

```
algorithm merge
  Input: two sorted vecs vec1 and vec2
  Output: vec3 that contains the elements of vec1 and vec2 in sorted order

  i = 0; j  = 0; vec3 = empty vec
  while i < vec1.size() and j < vec2.size()
    if vec1[i] <= vec2[j]
      vec3.push_back(vec1[i])
      i++
    else
      vec3.push_back(vec2[j])
      j++

   // Add remainder of other vector
   while i < vec1.size()
     vec3.push_back(vec1[i])
     i++
   while j < vec2.size()
     vec3.push_back(vec2[j])
     j++

  return vec3
```

- Runtime: O(n)
- Space complexity: O(n)

Merge Sort Pseudocode

```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec with its elements in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```

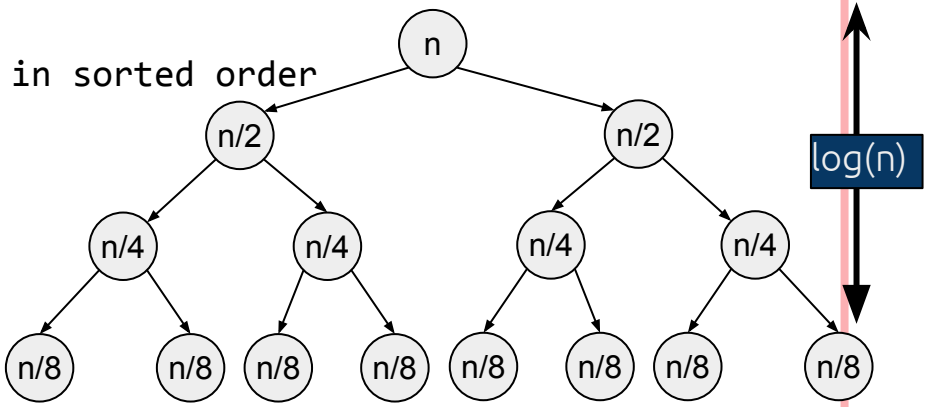
Base case: If the length of the vec is size 1 or smaller, then the vec is already sorted.

Recursive calls: Divide the vec into left and right halves and recursively sort the left and right half.

Solution building: Once you have the sorted left and sorted right, merge them together into one big sorted vec.

Time and Space Complexity of Merge Sort

```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec such that its elements are in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```

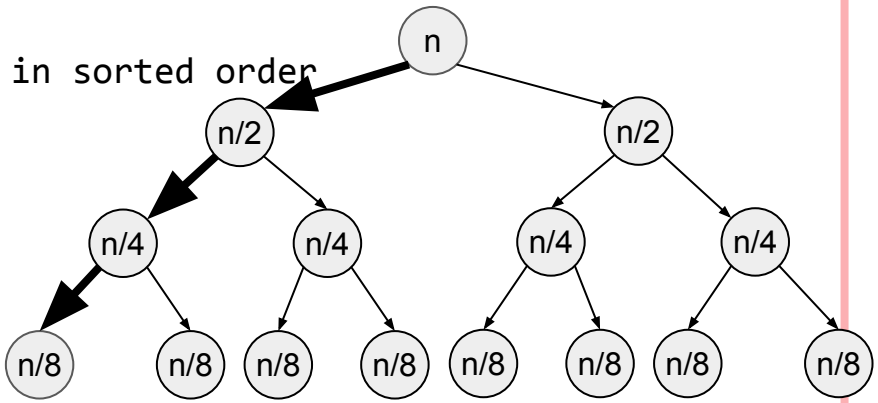


n total work happening at each level, with log(n) levels.

- Best-Case Runtime? O(n log(n))
- Average-Case Runtime? O(n log(n))
- Worst-Case Runtime? O(n log(n))
- Worst-Case Space complexity?

Time and Space Complexity of Merge Sort

```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec such that its elements are in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```



n total work happening at each level, with log(n) levels.

- Best-Case Runtime? O(n log(n))
- Average-Case Runtime? O(n log(n))
- Worst-Case Runtime? O(n log(n))
- Worst-Case Space complexity? O(n)

Properties of Merge Sort

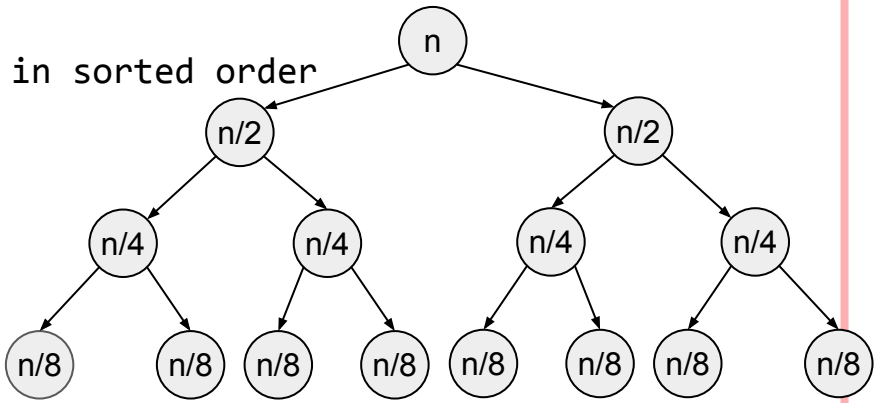
```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec such that its elements are in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```

Although we're talking about divide & conquer...is this sorting implementation:

- Stable?
- In-Place?
- Adaptable?

Time and Space Complexity of Merge Sort

```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec such that its elements are in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```



- Best-Case Runtime?
- Average-Case Runtime?
- Worst-Case Runtime?
- Worst-Case Space complexity?

Properties of Merge Sort

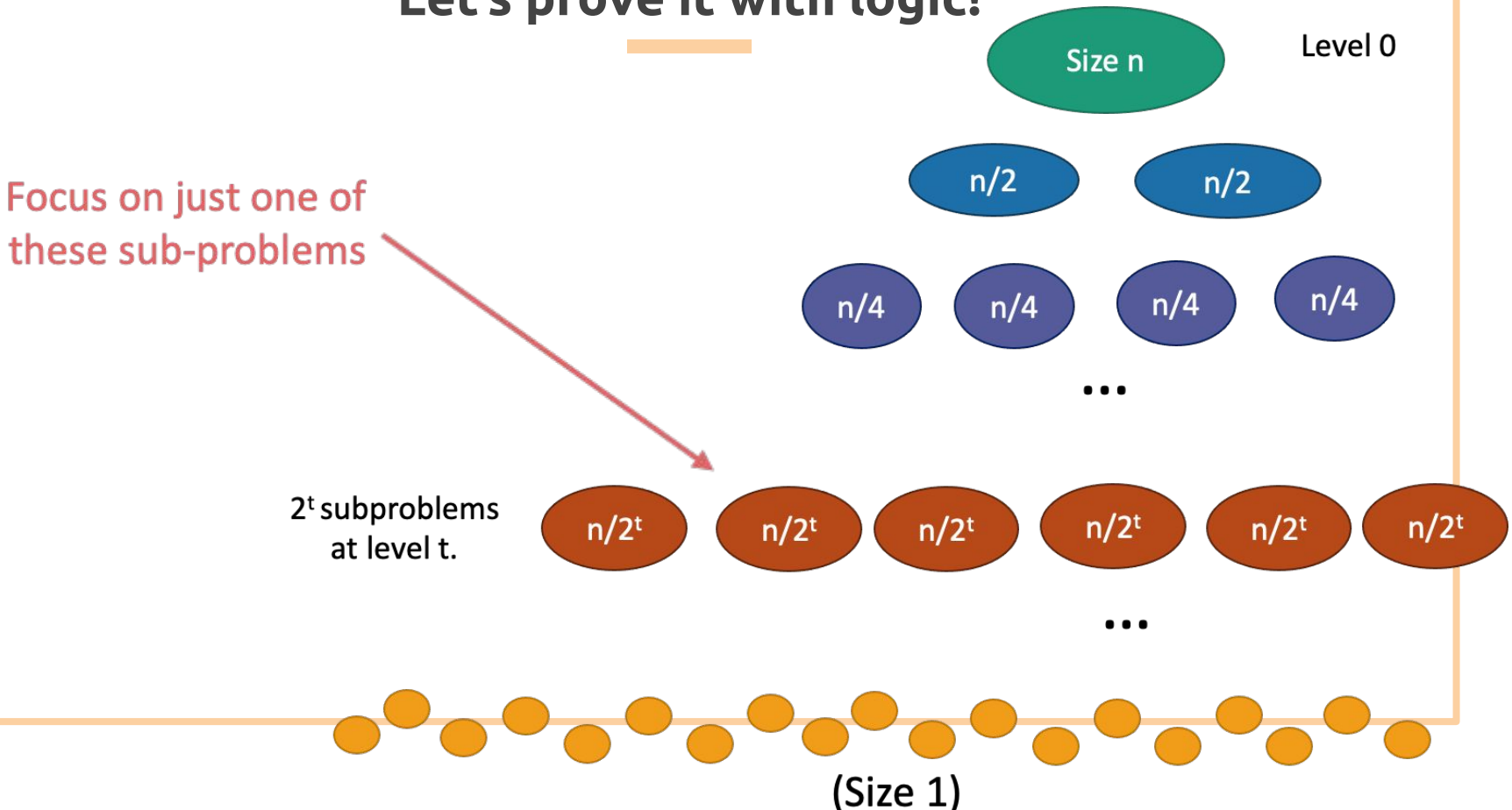
```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec such that its elements are in sorted order

  if N <= 1
    return vec
  midpoint = floor(N/2)
  left = mergeSort(vec[0 to midpoint])
  right = mergeSort(vec[midpoint to N])
  return merge(left, right)
```

Although we're talking about divide & conquer...is this sorting implementation:

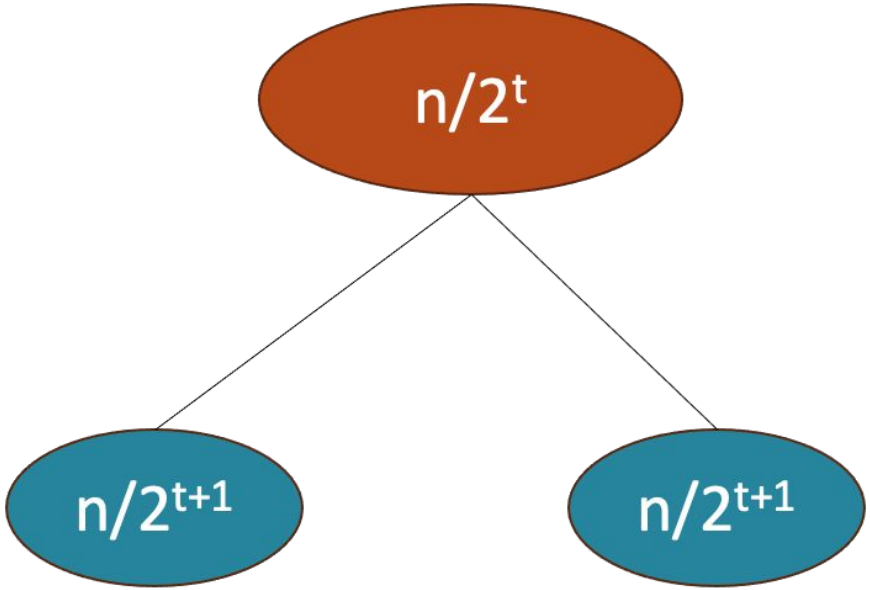
- Stable? Yes
- In-Place? No
- Adaptable? No

Let’s code it!!!

Let's prove it with logic!



How much work in this (arbitrary) sub-problem?



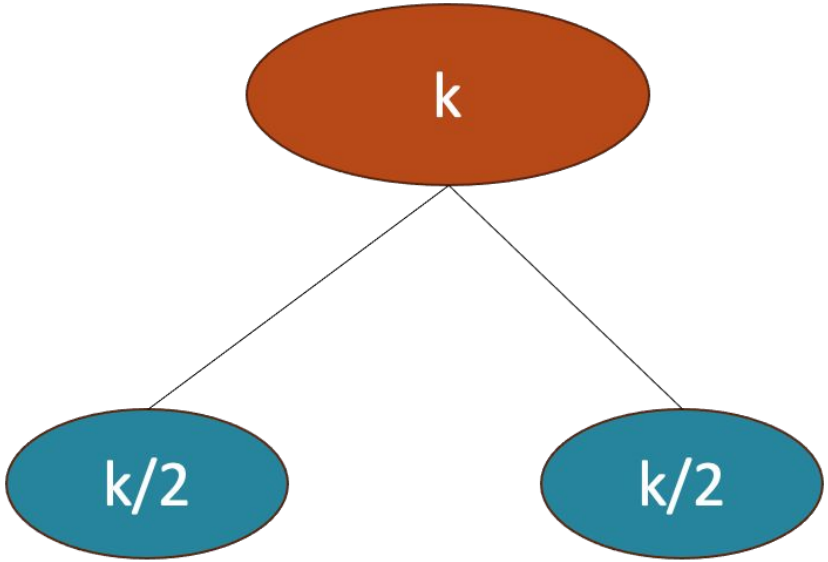
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this (arbitrary) sub-problem?

Let $k=n/2^t$...

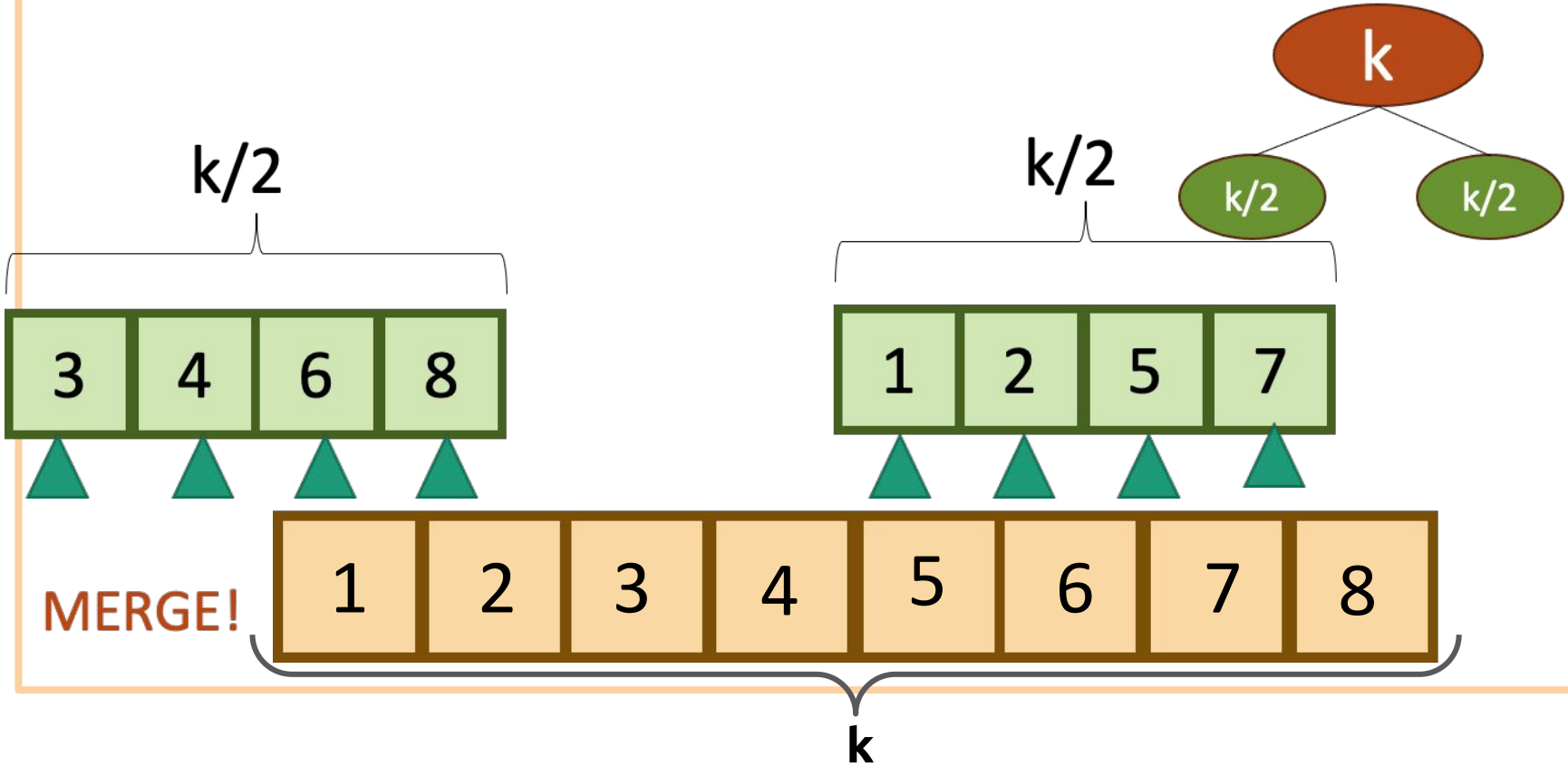


Time spent MERGE-ing the two subproblems

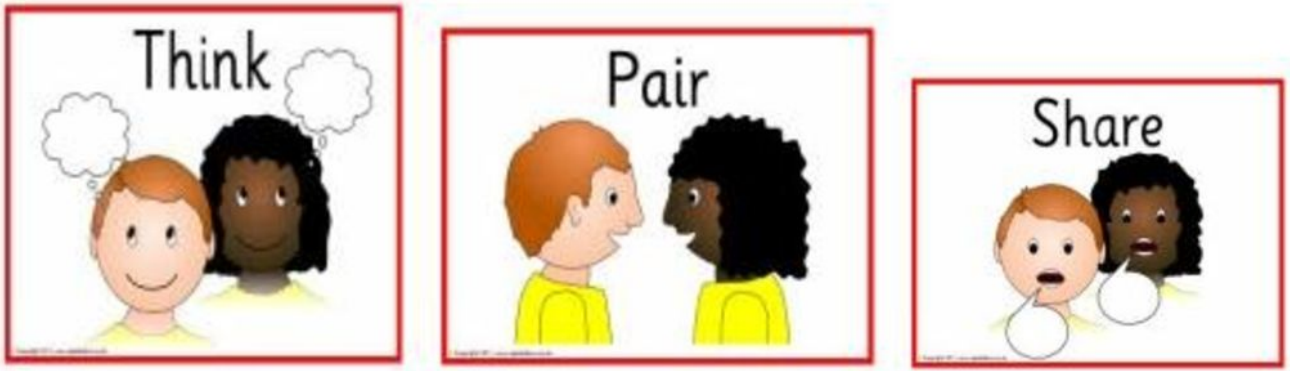
+

Time spent within the two sub-problems

So the question is - how long does it take to MERGE?

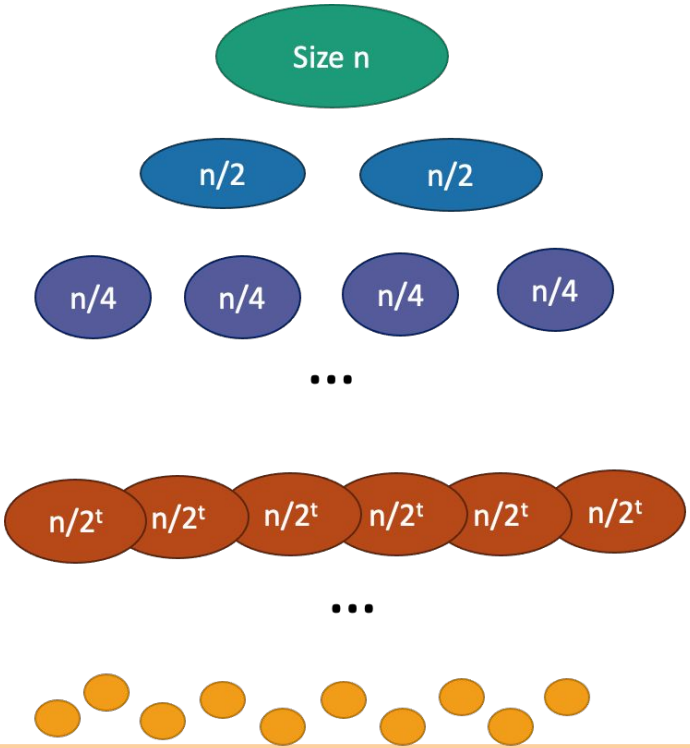


So the question is - how long does it take to MERGE?

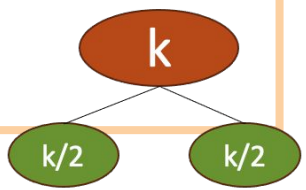


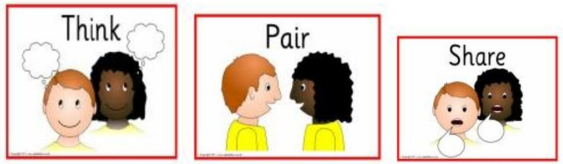
Answer: It takes time O(k), since we just walk across the list once.

Our Recursion Tree

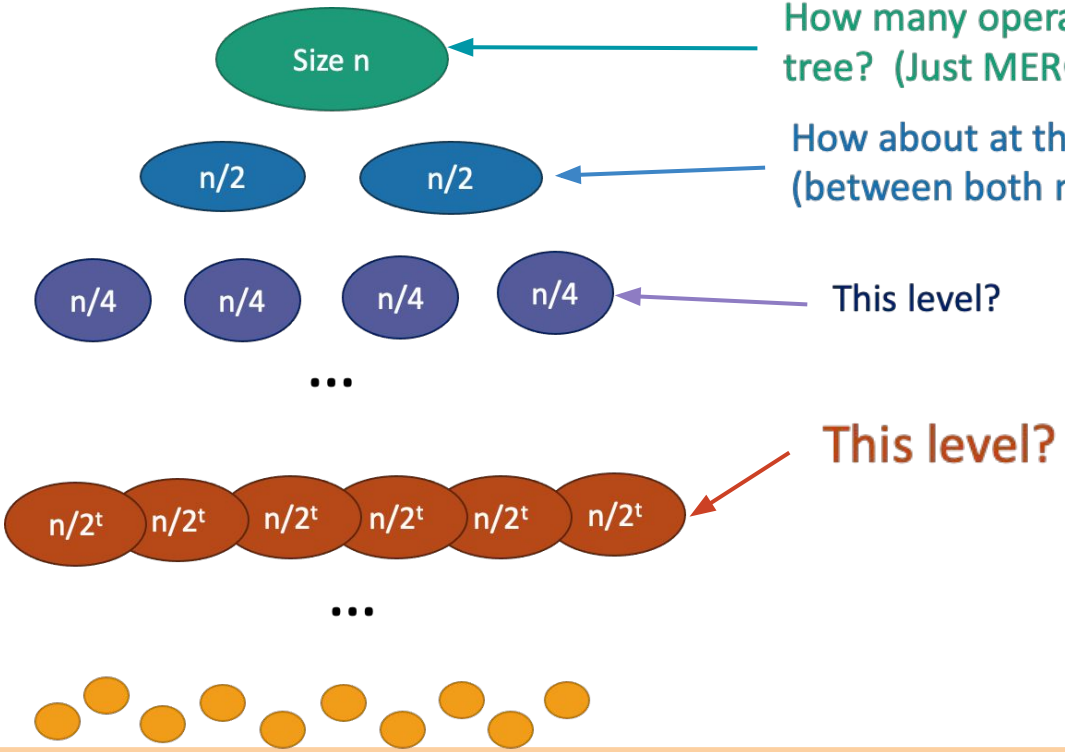


There are O(k) operations done at this node.





Our Recursion Tree



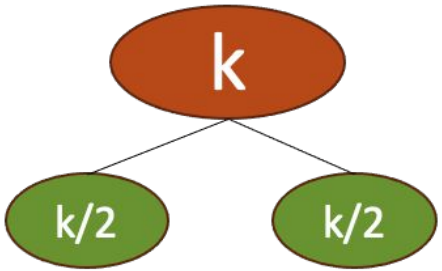
How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

How about at this level of the tree? (between both n/2-sized problems)

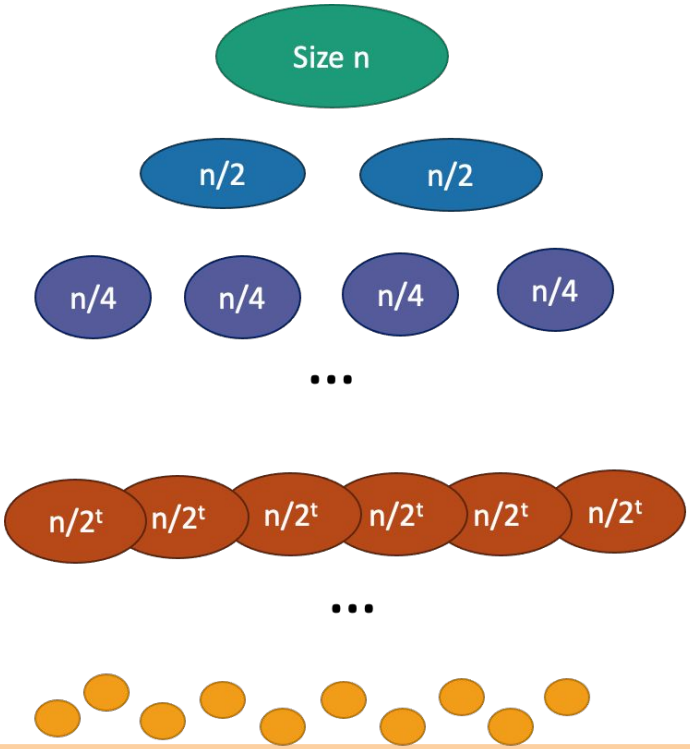
This level?

This level?

There are O(k) operations done at this node.



Our Recursion Tree



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	O(n)
1	2	n/2	O(n)
2	4	n/4	O(n)
...			
t	2^t	$n/2^t$	O(n)
...			
log(n)	n	1	O(n)

Total Runtime

- O(n) steps per level, at every level
- log(n) + 1 levels
- O(n log(n)) total!

That was the claim!

So far?!

- Sorting: InsertionSort & MergeSort
- What does it mean to work and be fast?
  - Worst-Case Analysis
  - Big-Oh Notation
- Analyzing correctness of iterative + recursive algos
  - Induction!!!
- Analyzing running time of recursive algorithms
  - By drawing out the tree and adding it all up!

Big Questions!

- What is divide and conquer (technically)?
- What about MergeSort?
- How to pass behavioral interviews?

Behavioral Mock Interviews
Interviewer Training

Denzel Palm
University Recruiter

Meta

Let’s Quickly Check In:

Please scan the QR Code & fill out the survey to come

Hello 😊

I’m Denzel Palm and I am a University recruiter here at Meta on our Early Talent Diversity (University) Recruitment Team— here are a few things about me so we can get acquainted:

- Based in Detroit, MI🏙️
- Been @ Meta for <1 year 🖥️
- Western Michigan Alum
- Comms Mkt Major🏛️

What are Behavioral Mock Interviews?

- A behavioral mock interview is anytime somebody asks you to talk about yourself in order to learn whether you have the required soft skills for a role and will be a good fit for the company.
- **Soft Skills vs Hard Skills**: Hard skills refer to the job-related knowledge and abilities that employees need to perform their job duties effectively. Soft skills, on the other hand, are the personal qualities that help employees really thrive in the workplace.
- **Mock Interview do not count** towards or against a candidate's application process

What should a interviewee expect?

- Interviewee’s taking part in a Behavioral Mock Interview should expect the following:

- **Introduction:** The interviewer will introduce themselves and set expectations.
- **Questions:** Interviewer will ask a series of targeted questions geared towards Soft Skills.
- **Q&A:** Time will be allowed for Q&A. At this time ask any prepared questions you might have for the interviewer.

Behavioral Mock Interview Training

Interview Samples Questions

- **Time Management**
  - Tell me about the last time you handled a long-term project. How did you keep the project on track?
- **Overcoming Challenges**
  - Tell me about a time when you made a mistake. What did you do to correct it?
- **Communication**
  - Give me an example of a time when you persuaded someone. How'd you do it and why?
- **Teamwork**
  - Share an example of how you were able to motivate a coworker, your peers or your team.
- **Adaptability**
  - Tell me about a time when you had to be creative to solve a problem.

How To Best Answer Questions

- The **STAR** method is a structured manner of responding to a behavioral-based interview question by discussing the specific **situation, task, action, and result** of the situation you are describing.

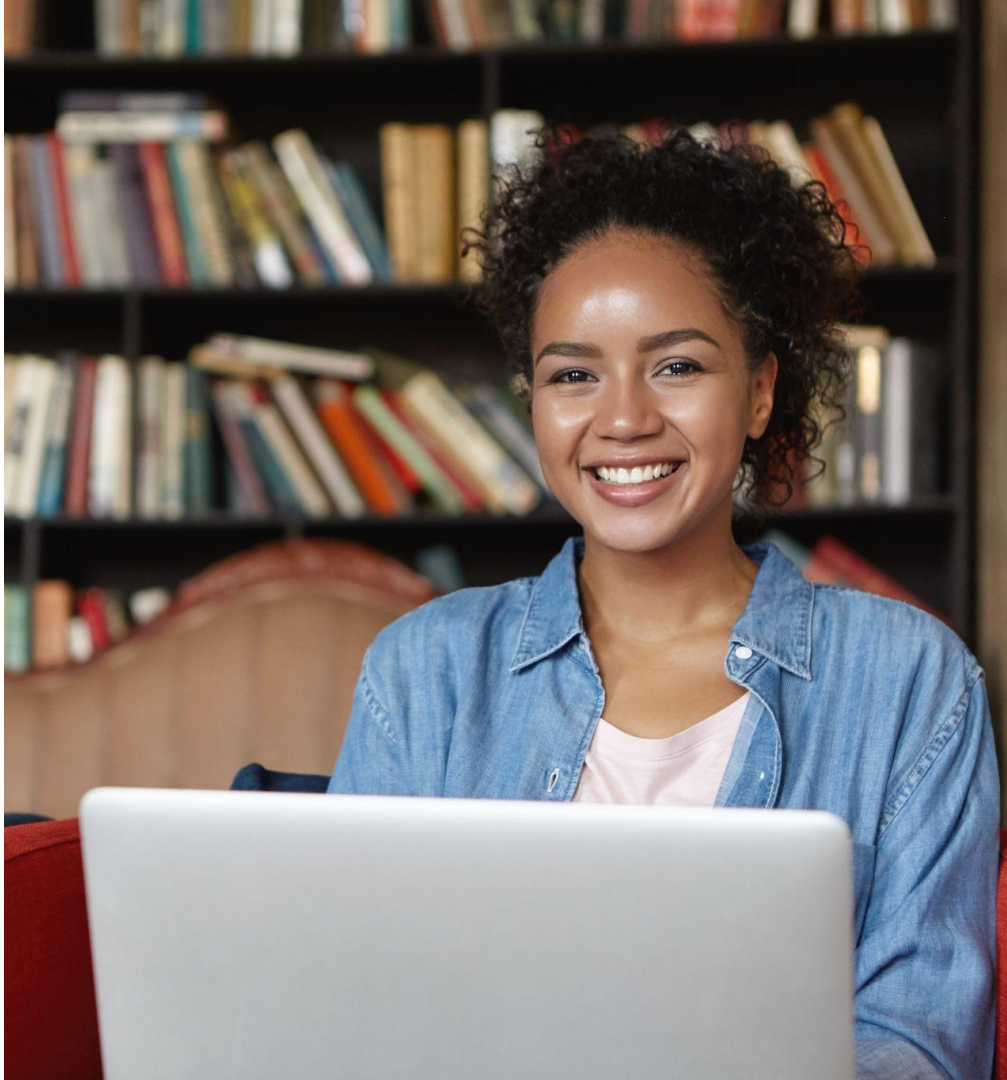
How to use the S-T-A-R Method

- **Situation**: Set the scene and give the necessary details for your example
- **Task**: Describe what your responsibility was in the situation.
- **Action**: Explain exactly what steps you took to address it
- **Result**: Share what outcomes your actions achieved and provide any data driven results.

Interview is Over, What Questions Should I Ask

- It’s a great best practice to come to your interview with prepared questions. Please remember to keep questions, professional and on topic.
- **Examples:**
- How would you describe your organization’s culture?
- What do you find most challenging about working for this organization?
- What have you enjoyed the most so far about working here?
- Can you tell me more about professional development opportunities your company offers?
- How do you evaluate success in this role?

QUESTIONS ?




Meta

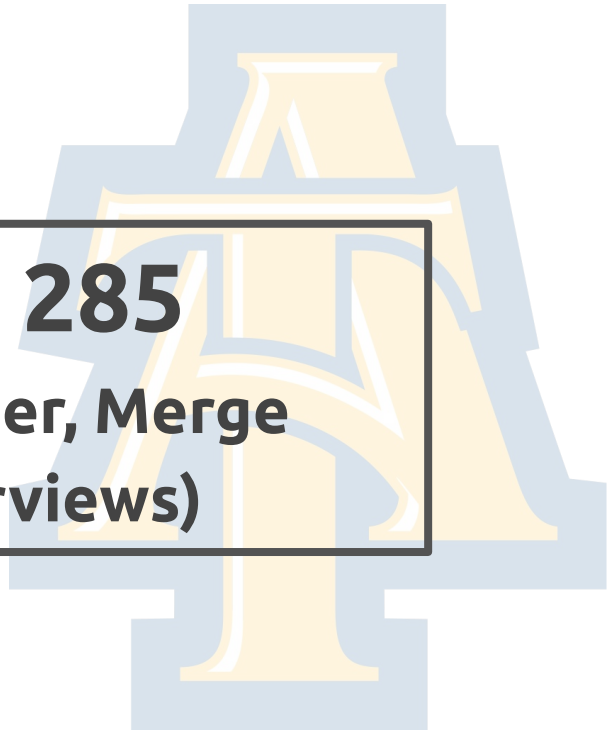
How was the pace today?

COMP - 285

Advanced Algorithms

Welcome to COMP 285

Lecture 7: Divide and Conquer, Merge Sort + CS Job Hunting (Interviews)



Lecturer: Chris Lucas (cflucas@ncat.edu)