COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 8: Master Theorem, O(n) Sorting

Chris Lucas (cflucas@ncat.edu)

HW2 is due!

Tonight @ 11:59pm ET

HW2 is due!

Tonight @ 11:59pm ET

Video walkthroughs!

HW2 is due!

09/20 @ 11:59pm ET

Video walkthroughs!

HW1 grades!

By either tonight or tomorrow!

HW3 released by EoD!

Due 9/27 @ 11:59pm ET

See Tolu’s Email!

Feedback is a Gift EC Opportunity!

Tech. Mock Interviews!

10/10-10/13 EC Opportunity!

Big Questions!

- MergeSort Generalized and Recurrence Relations!
- What is the Master Theorem?
- Is O(n) Sorting possible?

Big Questions!

- MergeSort Generalized and Recurrence Relations!
- What is the Master Theorem?
- Is O(n) Sorting possible?

Recall where we ended last lecture…

Divide & Conquer

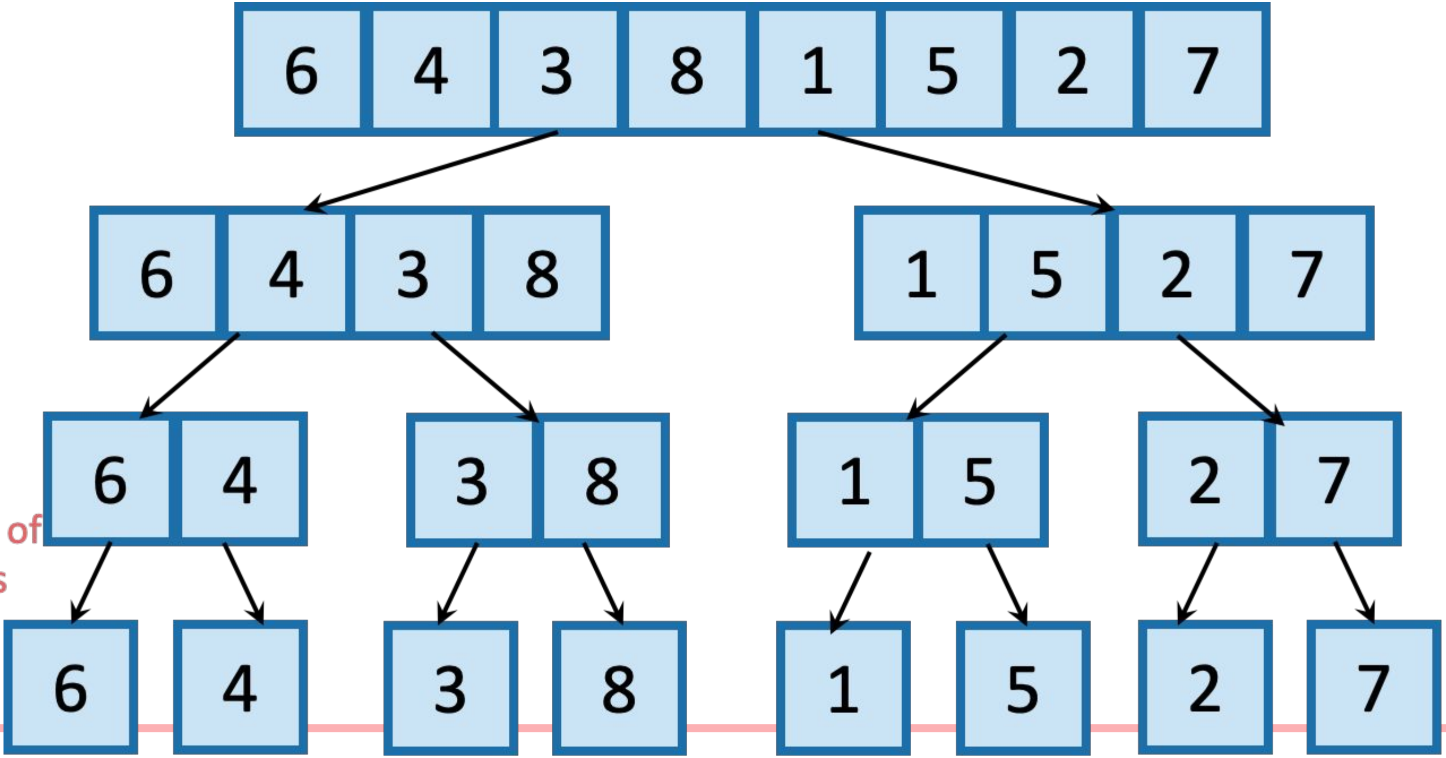
Divide & Conquer is a pattern in recursive algorithms that has two defining characteristics:

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 (even if reduced by some constant amount instead of a multiplicative factor), then it's called **decrease and conquer**.

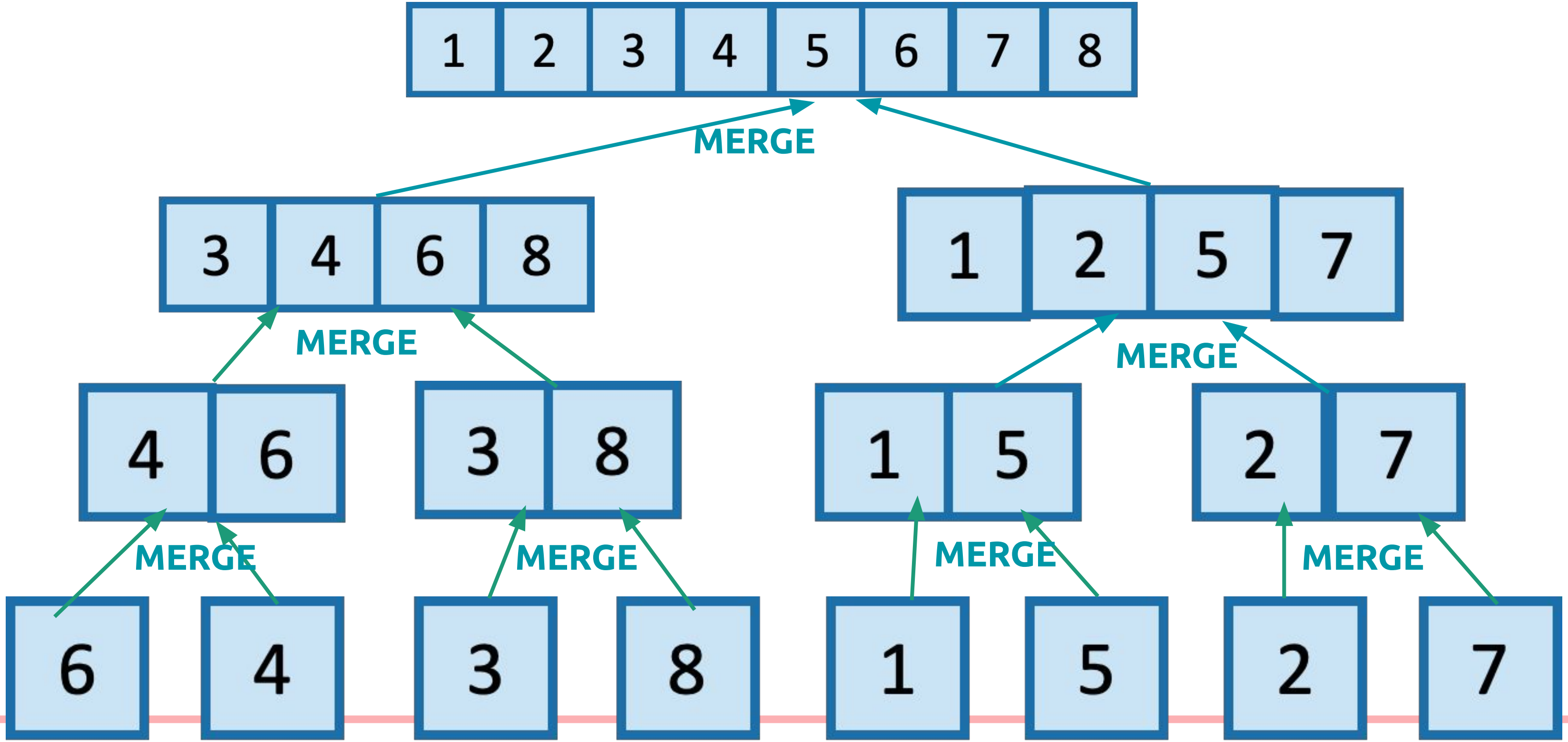
What actually happens?

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)
```

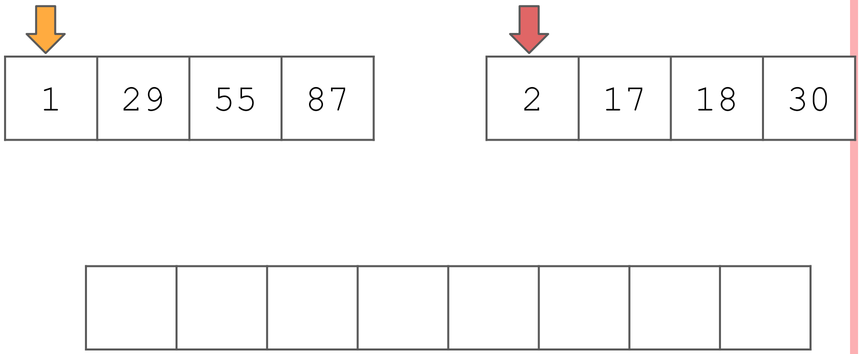
Base case: If the length of the vec is size 1 or smaller, then the vec is already sorted.

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.

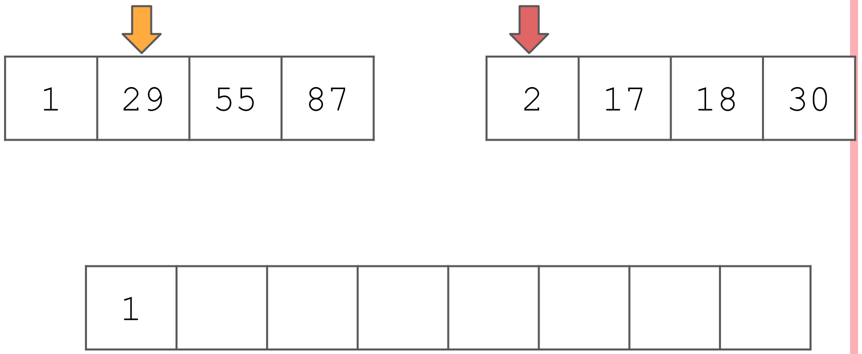
Merge Sort Pseudocode

```
algorithm 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
```



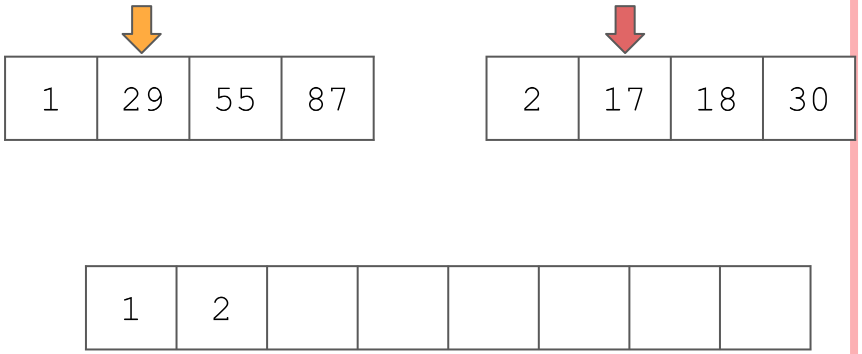
Merge Sort Pseudocode

```
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
```



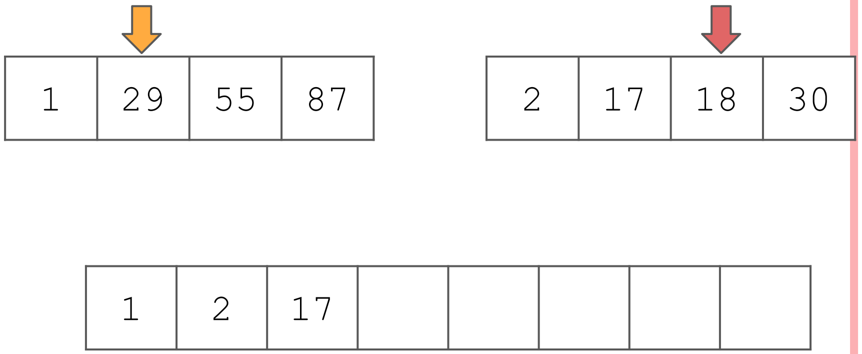
Merge Sort Pseudocode

```
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
```



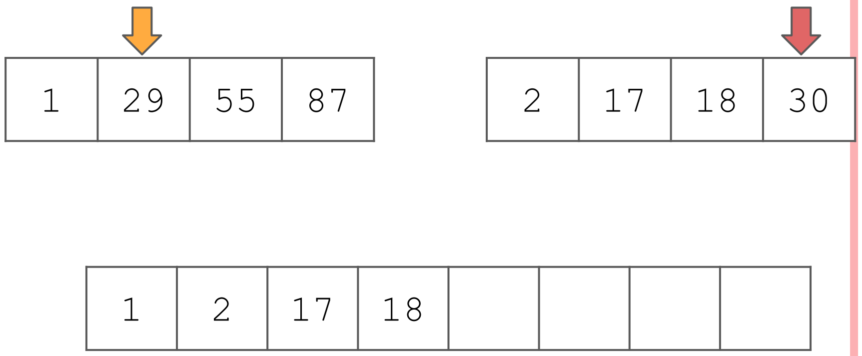
Merge Sort Pseudocode

```
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
```



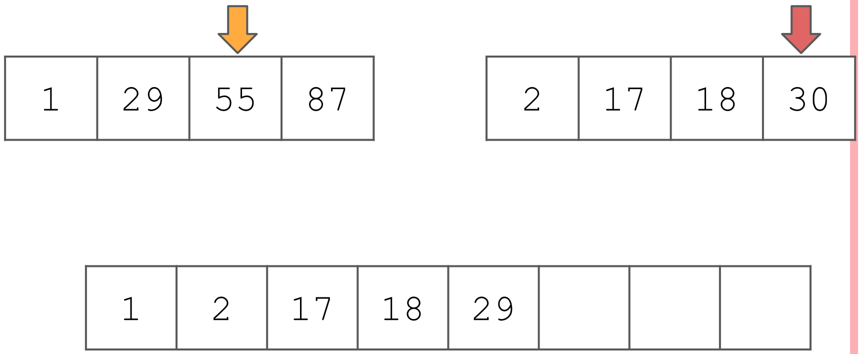
Merge Sort Pseudocode

```
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
```



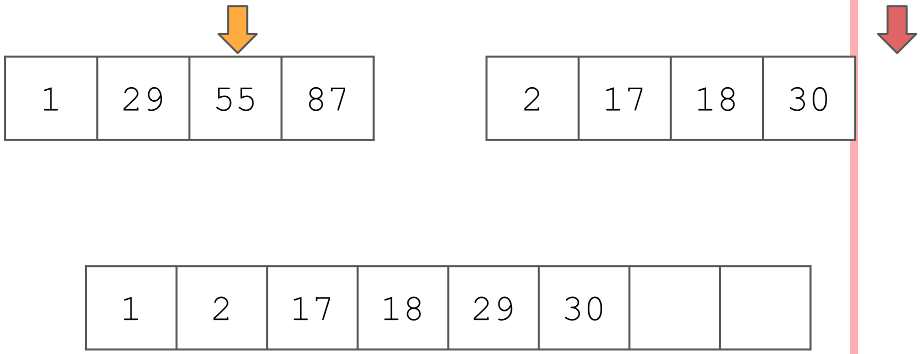
Merge Sort Pseudocode

```
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
```



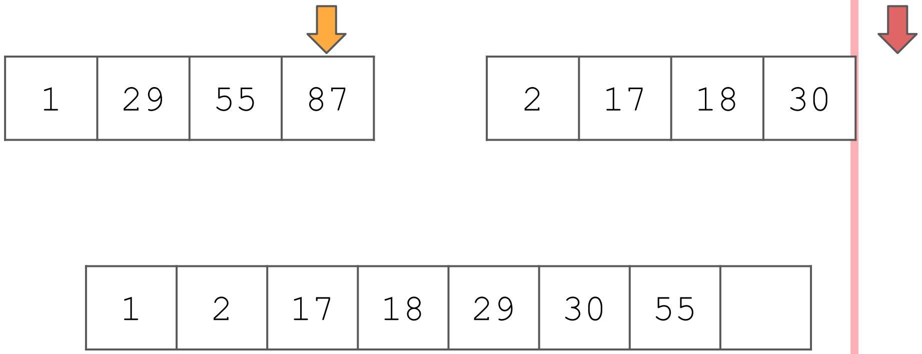
Merge Sort Pseudocode

```
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
```

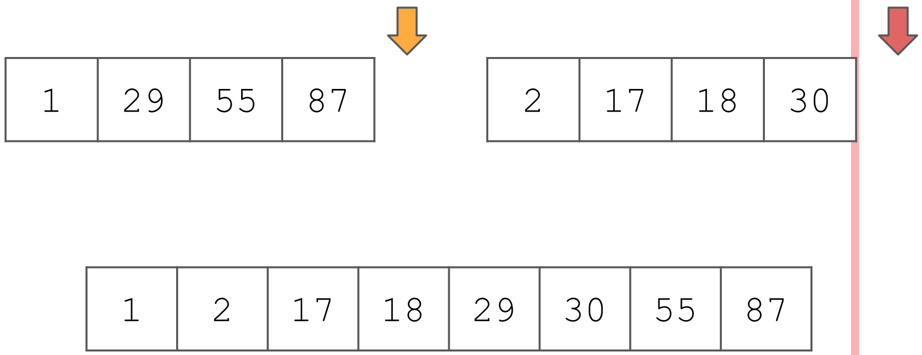
1	29	55	87

2	17	18	30

1	2	17	18	29	30	55	87

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:

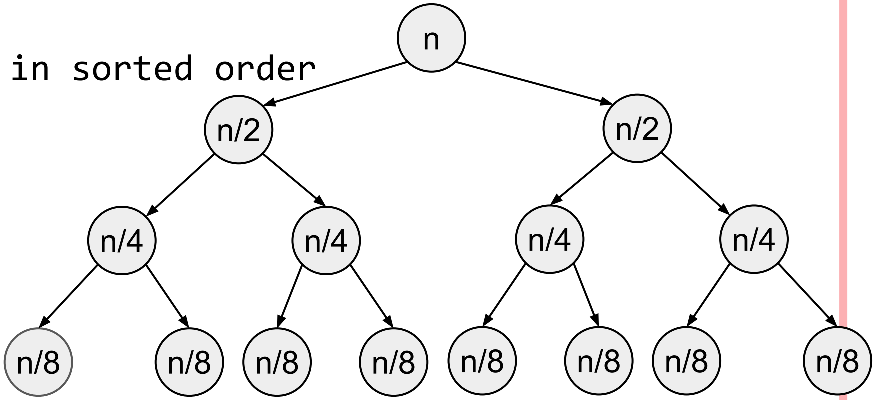
Merge Sort Pseudocode

```
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)

Time and Space Complexity of Merge Sort

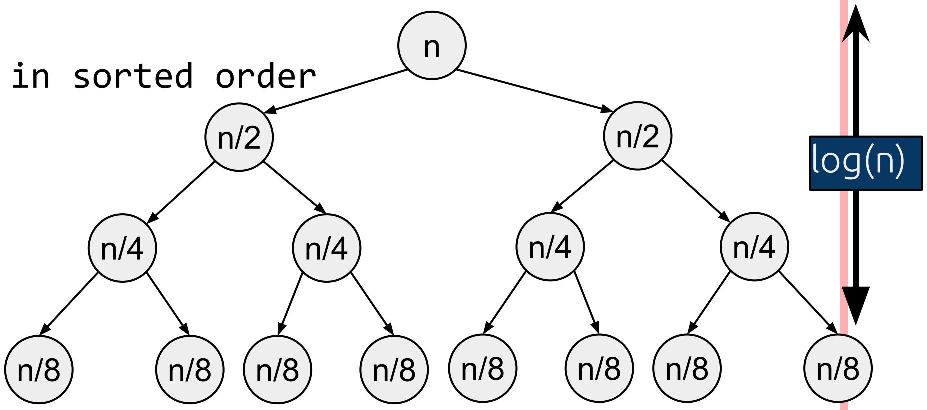
```
algorithm mergeSort
  Input: vector of ints vec of size N
  Output: vec such that its elements are in sorted order

  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?

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)
```

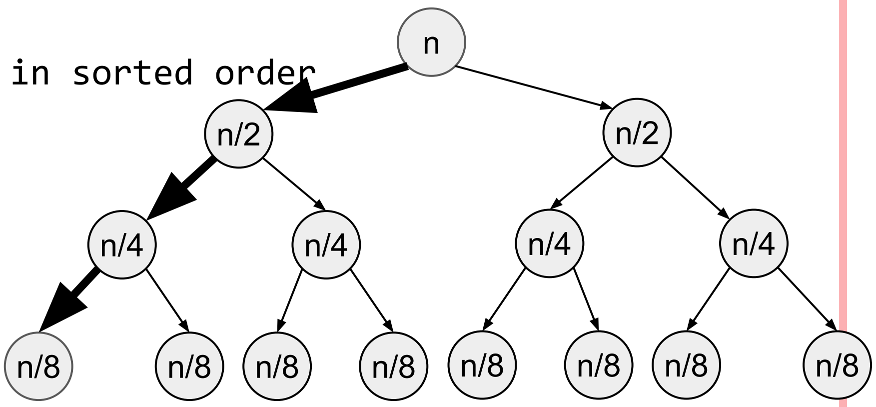


n total work happening at each level, with log(n) levels.

- 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?

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

MergeSort - Recurrence Relation!

- Let $T(n)$ be the running time of MergeSort on a length n array
- We know that $T(n) = O(n \log (n))$
- We also know that $T(n)$ satisfies the following
-

```
MERGESORT(A):
    n = length(A)
    if n ≤ 1:
        return A
    L = MERGESORT(A[:n/2])
    R = MERGESORT(A[n/2:])
    return MERGE(L,R)
```

MergeSort - Recurrence Relation!

- Let T(n) be the running time of MergeSort on a length n array
- We know that T(n) = O(n log (n))
- We also know that T(n) satisfies the following
-

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Running time for problem of size n

Two smaller problems

The cost to Merge!

```
MERGESORT(A):
    n = length(A)
    if n ≤ 1:
        return A
    L = MERGESORT(A[:n/2])
    R = MERGESORT(A[n/2:])
    return MERGE(L,R)
```

What's a "Recurrence Relation"

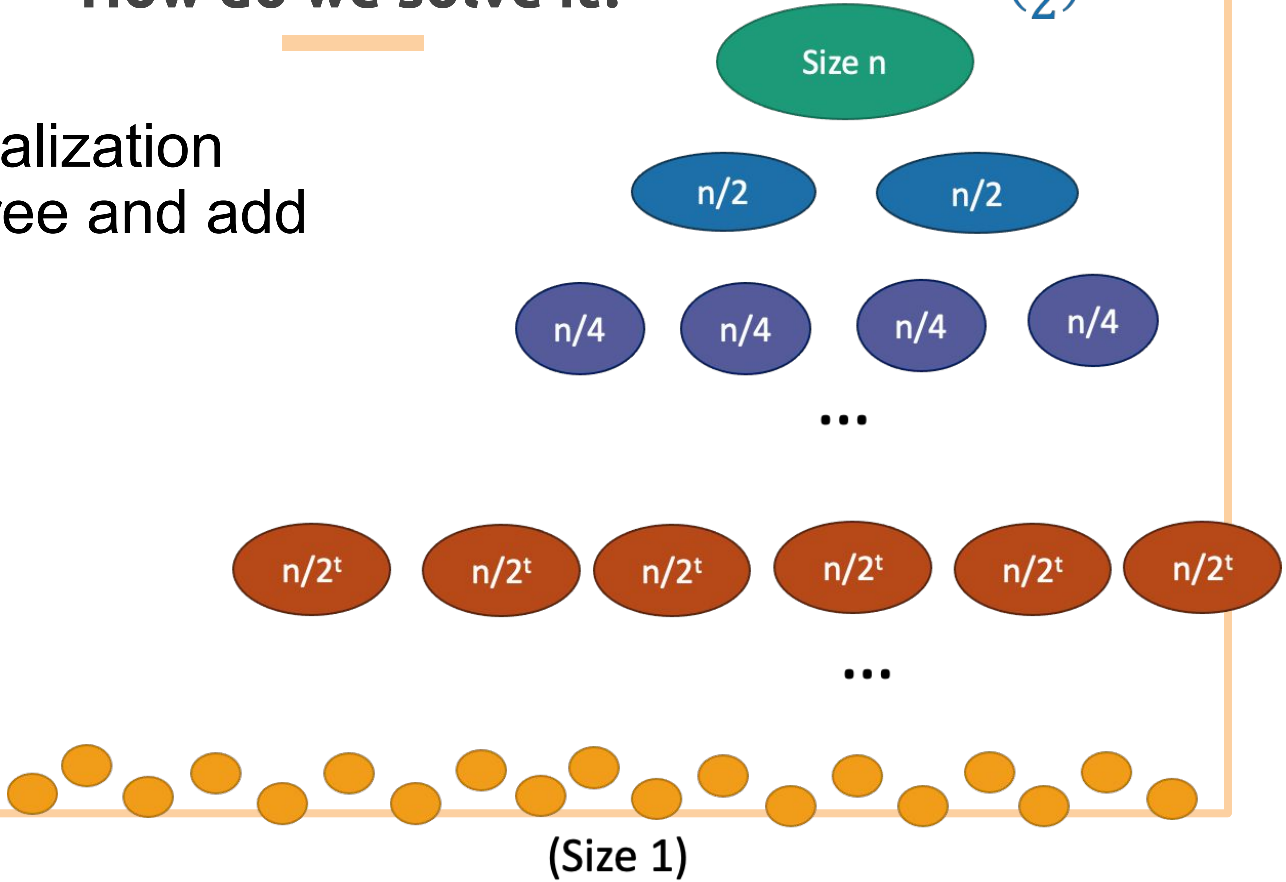
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

- The above is called a **recurrence relation**
- Because it gives a formula for **T(n)** in terms of **T(less than n)**
- Not that useful - normally, we want a **closed form expression**
- For example, **T(n) = O(n log (n))** because then we can plug-in numbers directly!

How do we solve it?

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

- The "tree" visualization
- We draw the tree and add up all the work



Let's try it!

- $T_1(n) = T_1\left(\frac{n}{2}\right) + n, \quad T_1(1) = 1.$

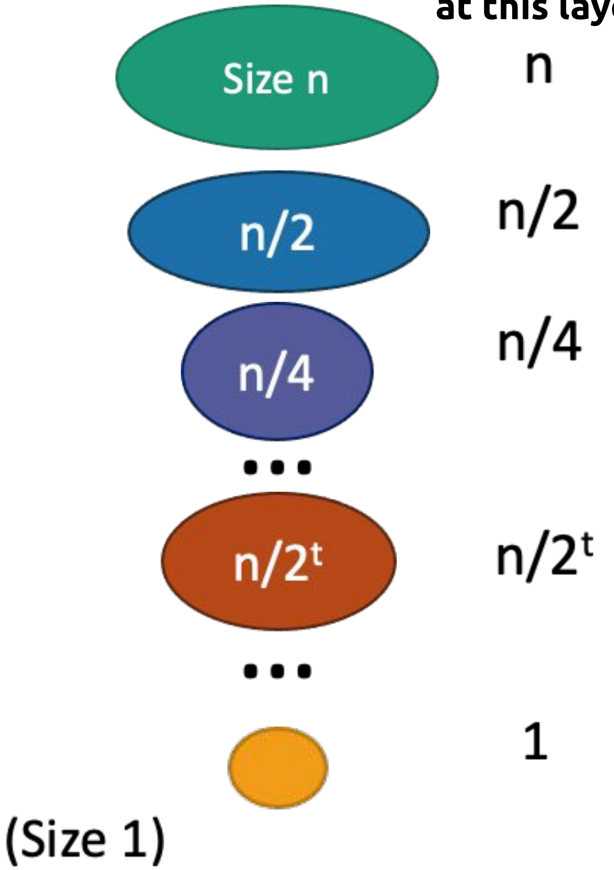
Let's try it!

- $T_1(n) = T_1\left(\frac{n}{2}\right) + n, \quad T_1(1) = 1.$

$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} = 2n - 1$$

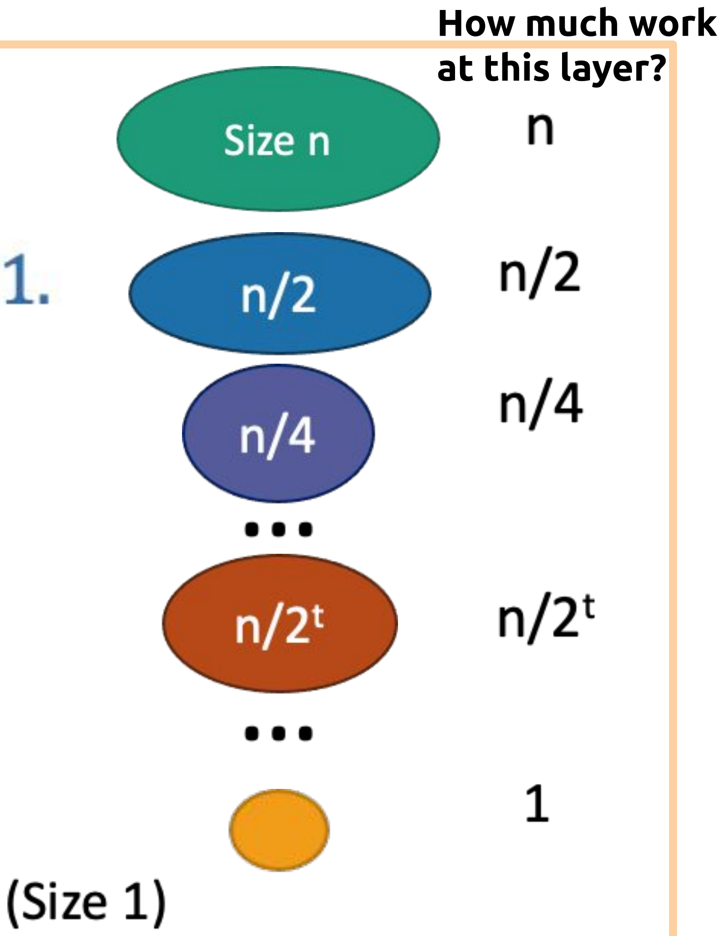
$T_1(n) = O(n).$

How much work at this layer?



Let's try another!

- $T_2(n) = 4T_2\left(\frac{n}{2}\right) + n, \quad T_2(1) = 1.$



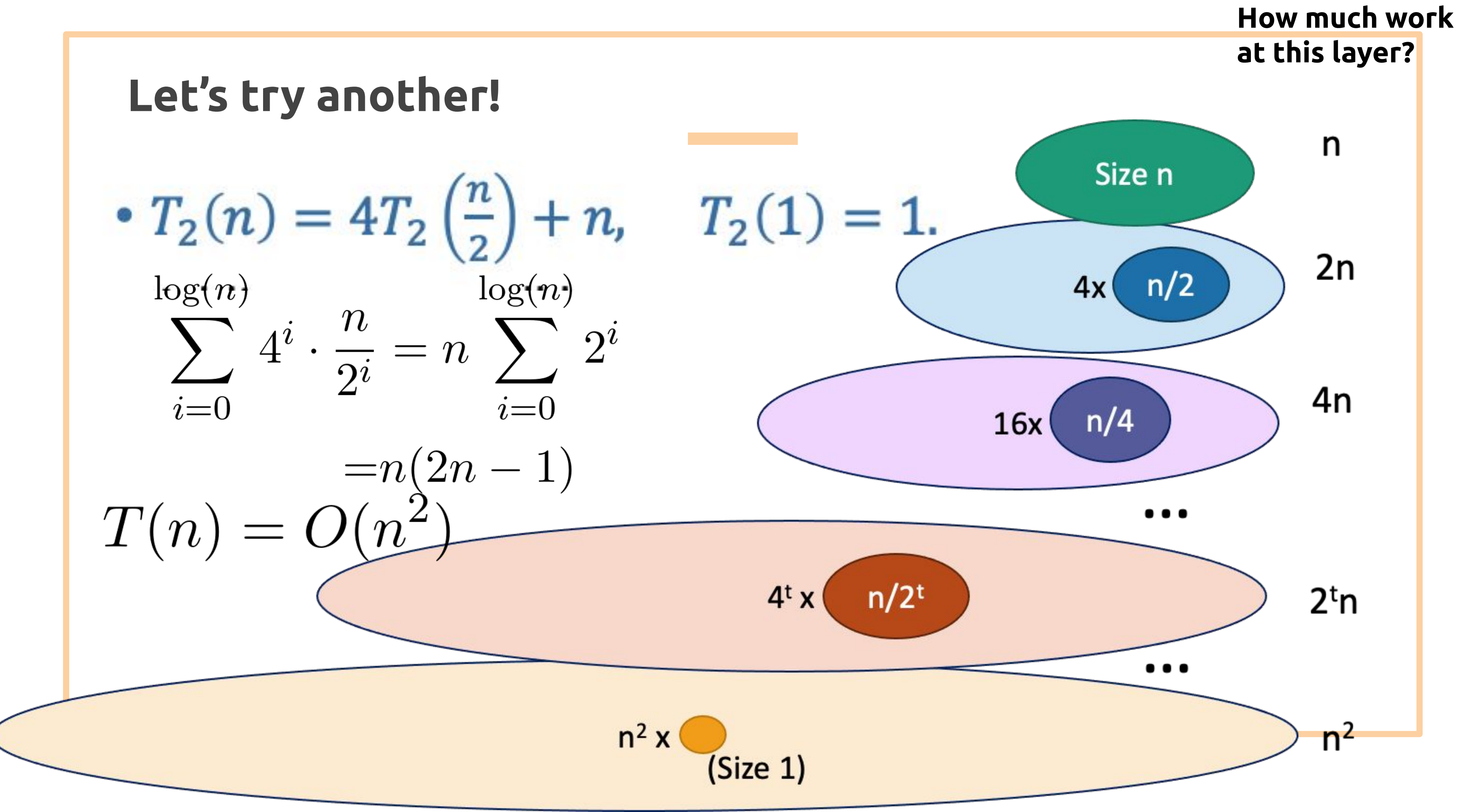
Let’s try another!

- $T_2(n) = 4T_2\left(\frac{n}{2}\right) + n, \qquad T_2(1) = 1.$

$$\sum_{i=0}^{\log(n)} 4^i \cdot \frac{n}{2^i} = n \sum_{i=0}^{\log(n)} 2^i$$

$$= n(2n - 1)$$

$$T(n) = O(n^2)$$



More Examples

T(n) = time to solve a problem of size n.

$$T(n) = 4T(n/2) + O(n)$$

$$T(n) = O(n^2)$$

Similar to our recursive multiplication.

$$T(n) = 3T(n/2) + O(n)$$

$$T(n) = O(n^{\log_2(3)}) \approx n^{1.6}$$

Karatsuba integer multiplication

$$T(n) = 2T(n/2) + O(n)$$

$$T(n) = O(n\log(n))$$

Merge sort

What’s the pattern?

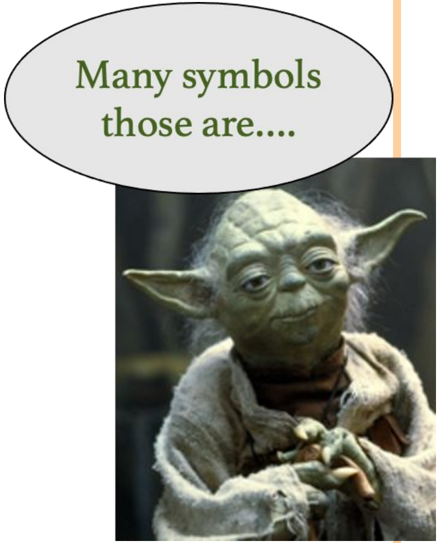
Big Questions!

- MergeSort Generalized and Recurrence Relations!
- What is the Master Theorem?
- Is O(n) Sorting possible?

The Master Theorem

- Suppose that $a \geq 1, b > 1$, and d are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$



The Master Theorem

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

work needed to combine the solutions

number of subproblems

Factor by which input size shrinks

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Back to our examples

- Needlessly recursive integer mult.
  - T(n) = 4 T(n/2) + O(n)
  - T(n) = O(n^2)

a = 4
b = 2
d = 1

$a > b^d$ ✓

- Karatsuba integer multiplication
  - T(n) = 3 T(n/2) + O(n)
  - T(n) = O($n^{\log\_2(3)} \approx n^{1.6}$)

a = 3
b = 2
d = 1

$a > b^d$ ✓

- MergeSort
  - T(n) = 2T(n/2) + O(n)
  - T(n) = O(nlog(n))

a = 2
b = 2
d = 1

$a = b^d$ ✓

- That other one
  - T(n) = T(n/2) + O(n)
  - T(n) = O(n)

a = 1
b = 2
d = 1

$a < b^d$ ✓

What’s the recurrence relation of Binary Search?

```
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 = (a + b) / 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)
```

Kahoot!

www.kahoot.it, Code: 747 7748

Enter your @aggies.ncat email

What’s the runtime of Binary Search?

```
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 = (a + b) / 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)
```

Kahoot!

www.kahoot.it, Code: 747 7748

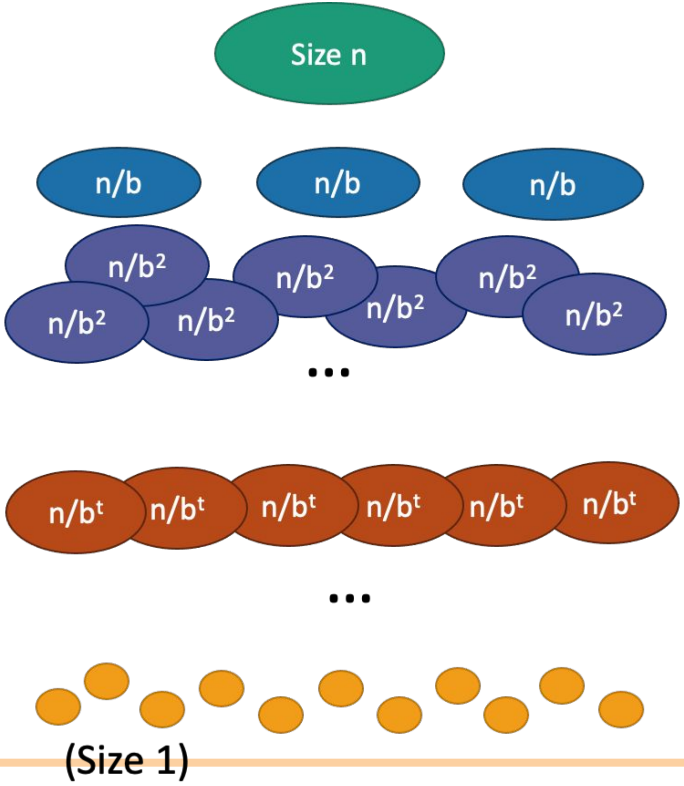
Enter your @aggies.ncat email

How do we proof this?

- We'll do the same thing we did for MergeSort, but using variables!

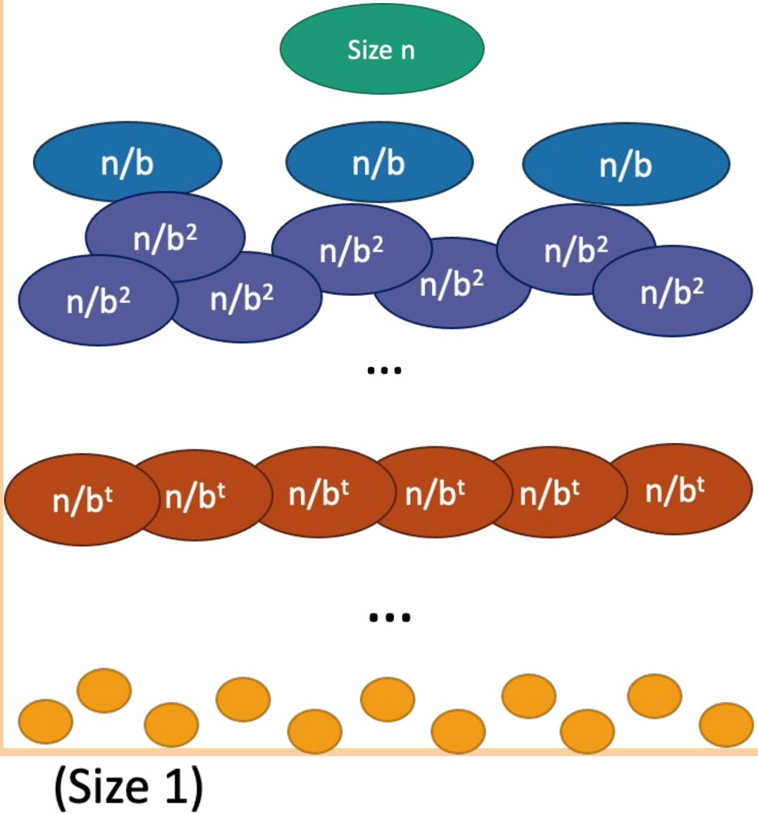
$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$

Our Recursion Tree



$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$

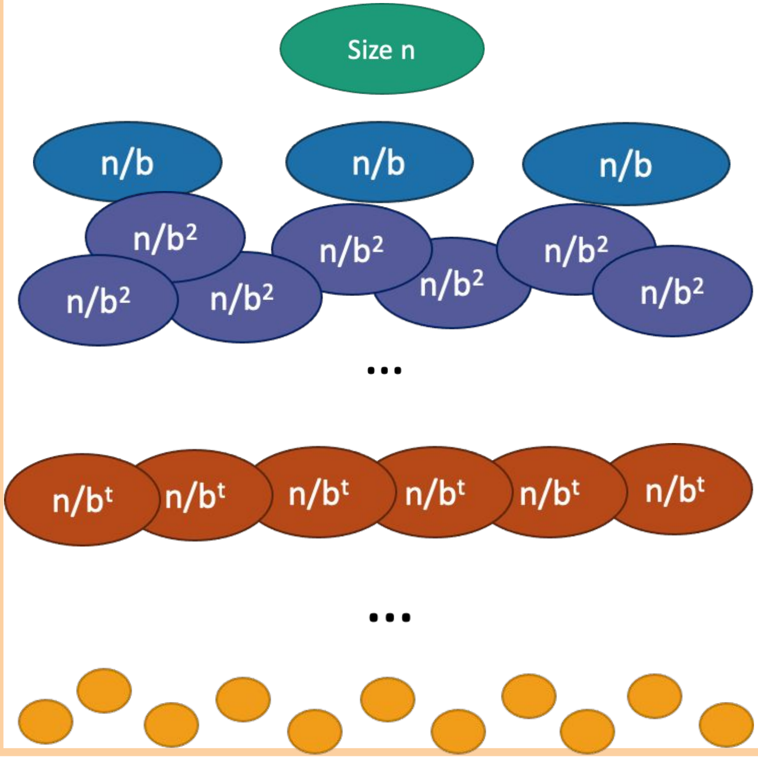
Our Recursion Tree



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	
1	a	n/b	
2	a^2	n/b^2	
…			
t	a^t	n/b^t	
…			
$\log_b(n)$	$a^{\log_b(n)}$	1	

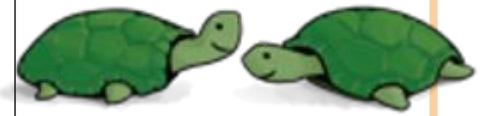
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$

Our Recursion Tree



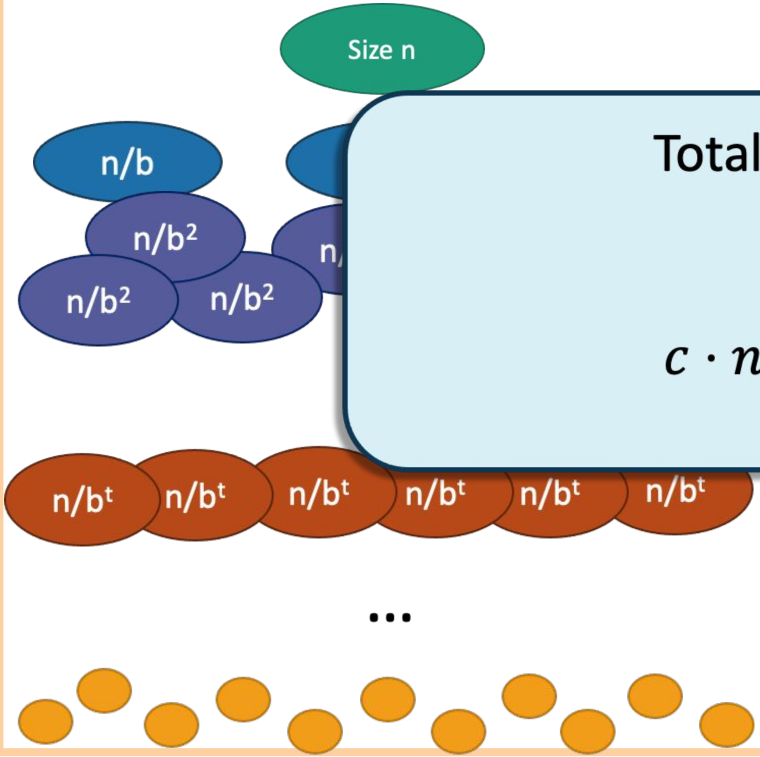
(Size 1)

Level	# problems	Size of each problem	Amount of work at this level
0	1	n	
1	a	n/b	
2	a^2	n/b^2	
...			
t	a^t	n/b^t	
...			
$\log_b(n)$	$a^{\log_b(n)}$	1	



$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$

Our Recursion Tree



(Size 1)

Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$c \cdot n^d$
			$c\left(\frac{n}{b}\right)^d$
			$c\left(\frac{n}{b^2}\right)^d$
t	a^t	n/b^t	$a^t c\left(\frac{n}{b^t}\right)^d$
...			
$\log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b n} c$

Total work is at most:

$$c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

We can check each case and see the theorem holds!

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

But we won't in lecture. Available in slides!

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Case 1: a = b^d

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Equal to 1!

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Case 1: a = b^d

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Equal to 1!

$$= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} 1$$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Case 1: a = b^d

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Equal to 1!

$$= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} 1$$

$$= c \cdot n^d \cdot (\log_b(n) + 1)$$

Case 1: a = b^d

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Equal to 1!

$$= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} 1$$

$$= c \cdot n^d \cdot (\log_b(n) + 1)$$

$$= c \cdot n^d \cdot \left(\frac{\log(n)}{\log(b)} + 1\right)$$

Case 1: a = b^d

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Equal to 1!

$$= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} 1$$

$$= c \cdot n^d \cdot (\log_b(n) + 1)$$

$$= c \cdot n^d \cdot \left(\frac{\log(n)}{\log(b)} + 1\right)$$

$$= \Theta(n^d \log(n))$$

Case 2: a < b^d

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Less than 1!

Aside: Geometric Sums

$$\sum_{t=0}^{N} x^t$$

If 0 < x < 1

$$x^0 + x^1 + x^2 + \cdots + x^N$$

$$\Theta(1)$$

If x > 1

$$x^0 + x^1 + x^2 + \cdots + x^N$$

$$\Theta(x^N)$$

Case 2: a < b^d

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Less than 1!

Case 2: a < b^d

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

$$= c \cdot n^d \cdot [\text{some constant}]$$

Less than 1!

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Case 2: a < b^d

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Less than 1!

$$= c \cdot n^d \cdot [\text{some constant}]$$

$$= \Theta(n^d)$$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Case 3: a > b^d

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Larger than 1!

Case 3: a > b^d

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

$$= \Theta\left(n^d \left(\frac{a}{b^d}\right)^{\log_b(n)}\right)$$

Larger than 1!

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Case 3: a > b^d

$$T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Larger than 1!

$$= \Theta\left(n^d \left(\frac{a}{b^d}\right)^{\log_b(n)}\right)$$

$$= \Theta\left(n^{\log_b(a)}\right)$$

We'll do it on the board!

Let's check each case!

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

But why? What are the three cases?

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

work needed to combine the solutions

number of subproblems

Factor by which input size shrinks

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

The eternal struggle

Branching causes the number of problems to explode!
The most work is at the bottom of the tree!

The problems lower in the tree are smaller!
The most work is at the top of the tree!

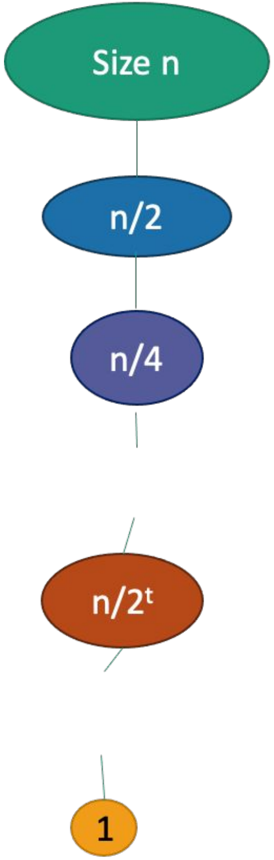
Tall and skinny tree

$$1.\ T(n) = T\left(\frac{n}{2}\right) + n, \qquad (a < b^d)$$

- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else.

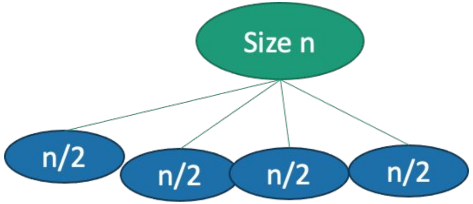
- T(n) = O(work at top) = O(n)

Most work at the top of the tree!



Needlessly recursive mult.: bushy tree

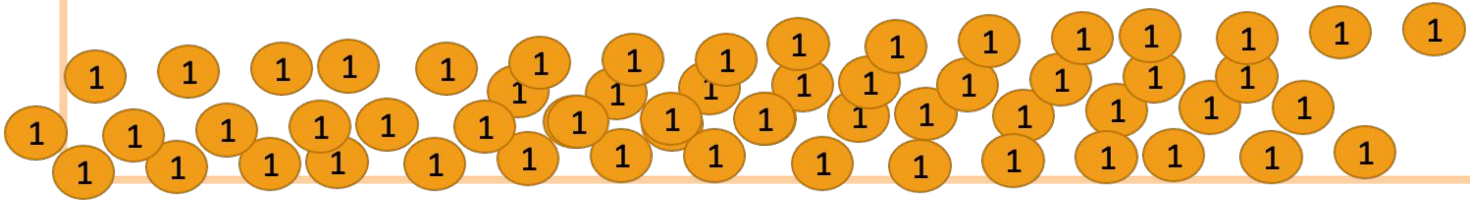
3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \qquad (a > b^d)$



- There are a HUGE number of leaves, and the total work is dominated by the time to do work at these leaves.

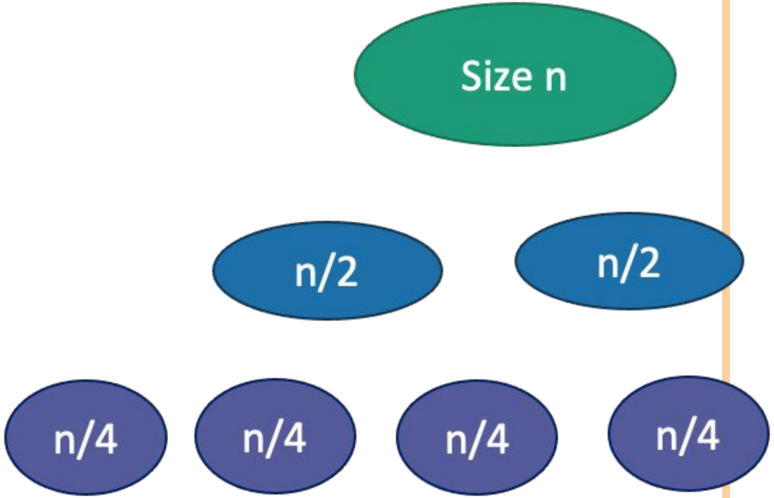
- T(n) = O(work at bottom) = O($4^{\text{depth of tree}}$) = O(n^2)

Most work at the bottom of the tree!



MergeSort: Just right

2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \qquad (a = b^d)$



- The branching **just** balances out the amount of work.
  - The same amount of work is done at every level.

- T(n) = (number of levels) * (work per level)
- = log(n) * O(n) = O(n log(n))

Big Questions!

- MergeSort Generalized and Recurrence Relations!
- What is the Master Theorem?
- Is O(n) Sorting possible?

General Intuition

When you know something extra about your problem, use it to make your algorithm more efficient. For example:

- **If you needed to find the maximum of a vector<int>, that would normally be O(n) to check every element in the vector. But if you knew it was sorted, it would be O(1) (take last element).**
- **If you needed to find whether or not an element was in a vector<int>, that would normally be O(n) to check every element. But if you knew it was sorted, it would be O(log(n)) (binary search).**
- **Let's see how this tip applies for sorting...**

Counting Sort Intuition

- If we constrain sorting to be slightly easier...
    - Input: vector<int> vec and an int k where every int in vec is between 0 and k (inclusive)
    - Output: vector<int> in sorted order
- How can we use this information to make the sorting problem easier?
- If we know k, then we could use a k+1 sized vector to store the counts of how often each element occurs. The element in index i in the counts corresponds to how often i occurs in vec.
- Counting Sort Demo: https://visualgo.net/en/sorting

Counting Sort Pseudocode

```
algorithm countingSort
  Input: vector<int> vec and an integer k where every int in vec
is between 0 and k
  Output: vector<int> in sorted order
// count each element using buckets
for (int i = 0; i < vec.size(); i++) {
    elem = input[i];
    buckets[elem] += 1;
}
for (int i = 1; i < k; i++) {
    count[i] += count[i-1];
}
for (int i = input.size()-1; i >= 0; i--) {
    elem = input[i];
    count[i] -= 1;
    output[count[i]] = elem;
}
```

Let’s look at a coding demo: https://replit.com/@samialsheikh/Lesson-9-On-sorts-SP22

Counting Sort Pseudocode

```
algorithm countingSort
  Input: vector<int> vec and an integer k where every int in vec
is between 0 and k
  Output: vector<int> in sorted order
// count each element using buckets
for (int i = 0; i < vec.size(); i++) {
    elem = input[i];
    buckets[elem] += 1;
}
for (int i = 1; i < k; i++) {
    count[i] += count[i-1];
}
for (int i = input.size()-1; i >= 0; i--) {
    elem = input[i];
    count[i] -= 1;
    output[count[i]] = elem;
}
```

n is vec.size() and k is value of k.

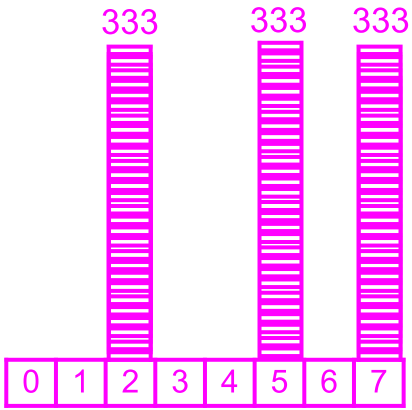
- Best-Case Runtime?
- Average-Case Runtime?
- Worst-Case Runtime?
- Worst-Case Space complexity?

input = {7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, … , 7, 7}
count = {0, 0, 0, 0, 0, 0, 0, 999}

Counting Sort Pseudocode

input = {7, 7, 7, 7, … , 2, 2, 2, 2, … 5, 5, 5, 5}
count = {0, 0, 333, 0, 0, 333, 0, 333}

```
algorithm countingSort
  Input: vector<int> vec and an integer k where every int in vec
is between 0 and k
  Output: vector<int> in sorted order
// count each element using buckets
for (int i = 0; i < vec.size(); i++) {
    elem = input[i];
    buckets[elem] += 1;
}
for (int i = 1; i < k; i++) {
    count[i] += count[i-1];
}
for (int i = input.size()-1; i >= 0; i--) {
    elem = input[i];
    count[i] -= 1;
    output[count[i]] = elem;
}
```



n is vec.size() and k is value of k.

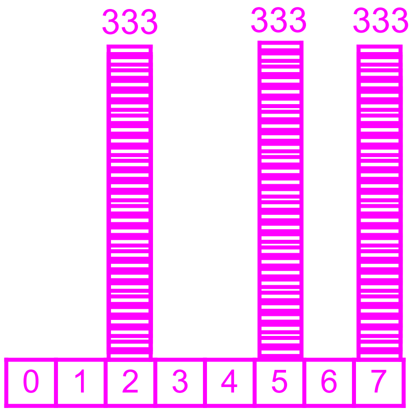
- Best-Case Runtime?
- Average-Case Runtime?
- Worst-Case Runtime?
- Worst-Case Space complexity?

input = {7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, … , 7, 7}
count = {0, 0, 0, 0, 0, 0, 0, 999}

Counting Sort Pseudocode

input = {7, 7, 7, 7, … , 2, 2, 2, 2, … 5, 5, 5, 5}
count = {0, 0, 333, 0, 0, 333, 0, 333}

```
algorithm countingSort
  Input: vector<int> vec and an integer k where every int in vec
is between 0 and k
  Output: vector<int> in sorted order
// count each element using buckets
for (int i = 0; i < vec.size(); i++) {
    elem = input[i];
    buckets[elem] += 1;
}
for (int i = 1; i < k; i++) {
    count[i] += count[i-1];
}
for (int i = input.size()-1; i >= 0; i--) {
    elem = input[i];
    count[i] -= 1;
    output[count[i]] = elem;
}
```



n is vec.size() and k is value of k.

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

Counting Sort Pseudocode

```
algorithm countingSort
  Input: vector<int> vec and an integer k where every int in vec
is between 0 and k
  Output: vector<int> in sorted order

  // count each element
  count = vector of k+1 zeros
  for element v in vec
    count[v] += 1

  // create output vector
  output = empty vector of ints
  for i in 0...k
    for j in 0...count[i]
      output.push_back(i)
  return output
```

What kind of input for Counting Sort would result in a lot of wasted space?

Counting Sort Pseudocode

```
algorithm countingSort
  Input: vector<int> vec and an integer k where every int in vec
is between 0 and k
  Output: vector<int> in sorted order

  // count each element
  count = vector of k+1 zeros
  for element v in vec
    count[v] += 1

  // create output vector
  output = empty vector of ints
  for i in 0...k
    for j in 0...count[i]
      output.push_back(i)
  return output
```

What kind of input for Counting Sort would result in a lot of wasted space?
Large k with a small vec (e.g. {9999999})

Radix Sort

- We can use our constrained sort here even further to address the limitations with Counting Sort.
- Numbers have digit places (ones digit, tens digit, hundreds digit, etc)
  - e.g. 14,820,129 has 8 digits
- Let’s see what happens if we can take advantage of this fact
- Radix Sort Demo: https://visualgo.net/en/sorting

Radix Sort Pseudocode

```
algorithm radixSort
  Input: vector<int> vec of size N
  Output: vec such that its elements are sorted

  d = the largest place value among all the numbers
  output = vec
  for i = 1, 2, ..., d
    output = use a stable sort on output keyed on digit i
  return output
```

Radix Sort Pseudocode

```
algorithm radixSort
  Input: vector<int> vec of size N
  Output: vec such that its elements are sorted

  d = the largest place value among all the numbers
  output = vec
  for i = 1, 2, ..., d
    output = use a stable sort on output keyed on digit i
  return output
```

We can use a stable counting sort keyed on the digit, which with k = 10 buckets is O(n + k) = O(n + 10) = O(n)

n is vec.size() and d is largest place value (pretty small!).
If using counting sort...

- Best-Case Runtime: O(nd)
- Average-Case Runtime: O(nd)
- Worst-Case Runtime: O(nd)
- Worst-Case Space complexity: O(n)

COMP - 285
Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 8: Master Theorem, O(n) Sorting

Chris Lucas (cflucas@ncat.edu)