Divide and Conquer Algorithms

CSE 101: Design and Analysis of Algorithms

Lecture 14

CSE 101: Design and analysis of algorithms

- Divide and conquer algorithms
  - Reading: Sections 2.3 and 2.4
- Homework 6 will be assigned today
  - Due Nov 20, 11:59 PM

Warning

- If you have an algorithm that has a recursion such as

$$T(n) = aT(n-b) + O\left(n^d\right)$$

- This is sometimes referred to as a reduce and conquer. Be careful of this type of recursion because if a is greater than 1, then the algorithm may take exponential time.

Search

- Given a sorted list of integers and a target integer, determine the index of the target if it is in the list
- What is the fastest runtime that we could hope for?

Search

- Given a sorted list of integers and a target integer, determine the index of the target if it is in the list
- What is the fastest runtime that we could hope for?
- We have two options for each number we compare to the target (binary tree)

Search

- Sorted list of n integers:

  4, 29, 31, 52, 79, 82, 83, 99, 100, 111, 142, 153, 160, 169, 181

Search

- Sorted list of n integers:

  4, 29, 31, 52, 79, 82, 83, 99, 100, 111, 142, 153, 160, 169, 181

- Binary tree of n nodes

Search

- Sorted list of n integers:

  4, 29, 31, 52, 79, 82, 83, 99, 100, 111, 142, 153, 160, 169, 181

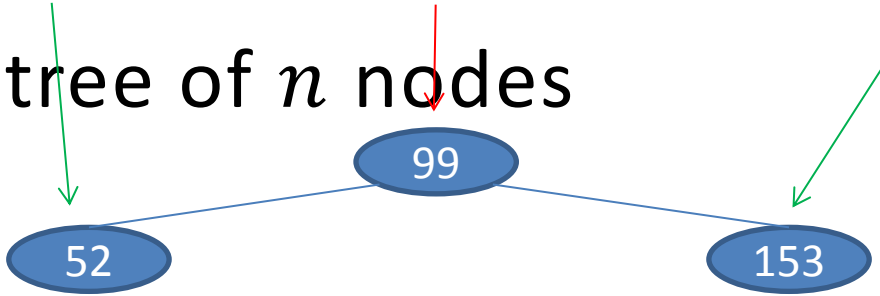
- Binary tree of n nodes

Search

- Sorted list of n integers:

  4, 29, 31, 52, 79, 82, 83, 99, 100, 111, 142, 153, 160, 169, 181

- Binary tree of n nodes

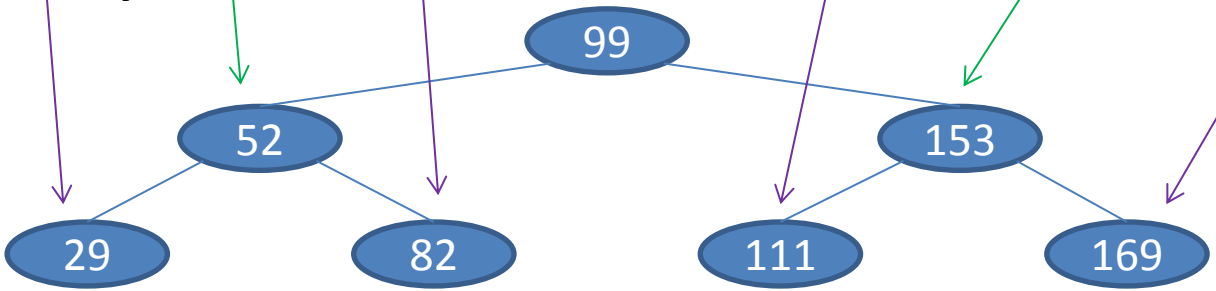


Search

- Sorted list of n integers:

  4, 29, 31, 52, 79, 82, 83, 99, 100, 111, 142, 153, 160, 169, 181

- Binary tree of n nodes

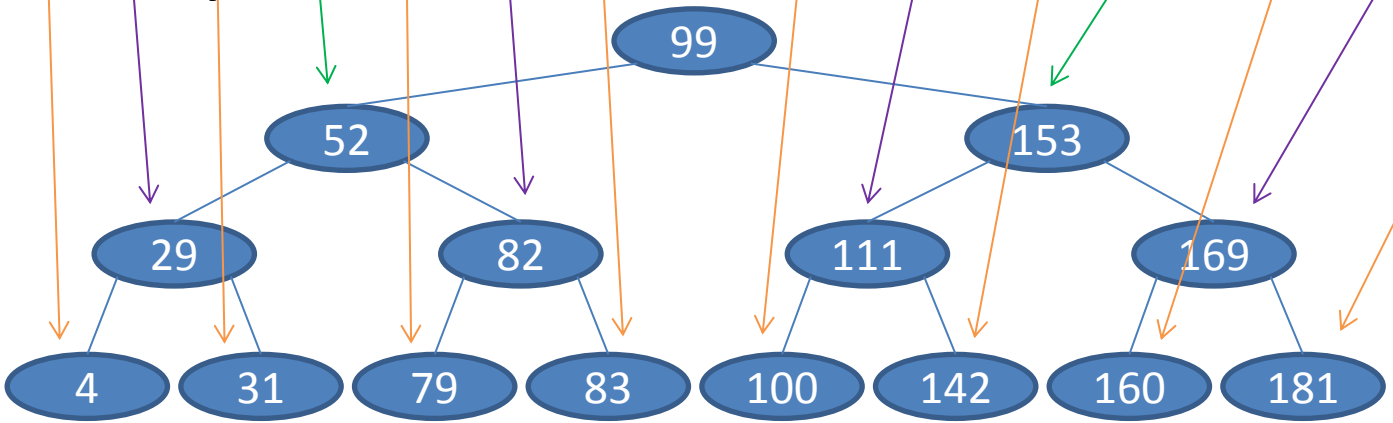


Search

- Sorted list of n integers:

4, 29, 31, 52, 79, 82, 83, 99, 100, 111, 142, 153, 160, 169, 181

- Binary tree of n nodes

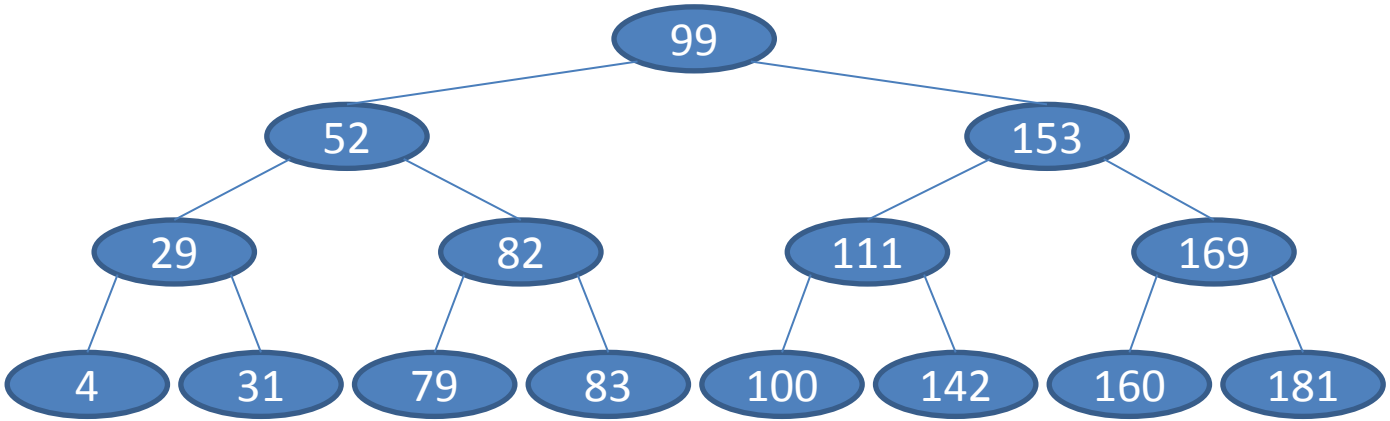


Search

- Sorted list of n integers:

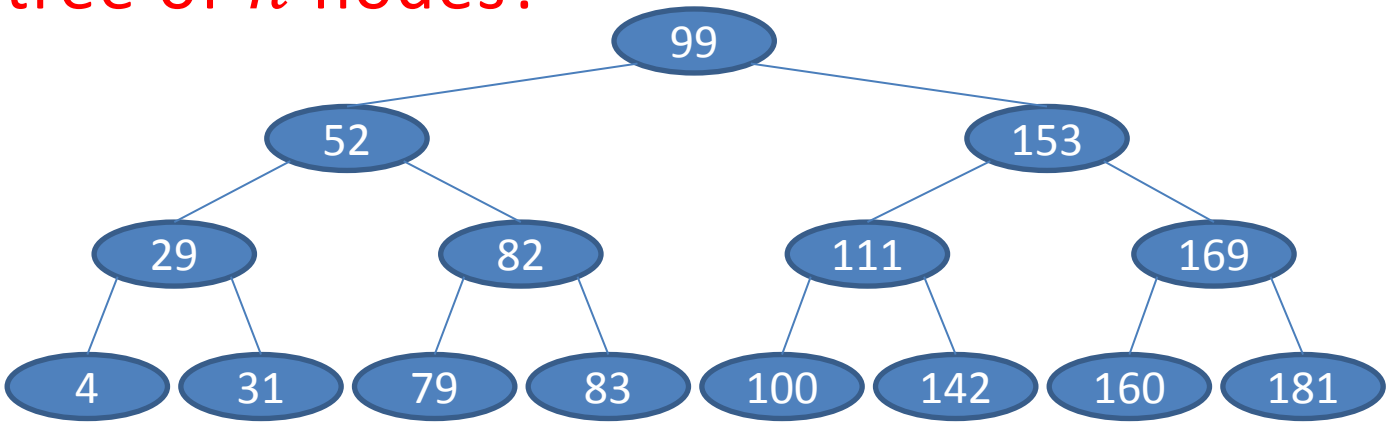
  4, 29, 31, 52, 79, 82, 83, 99, 100, 111, 142, 153, 160, 169, 181

- Binary tree of n nodes



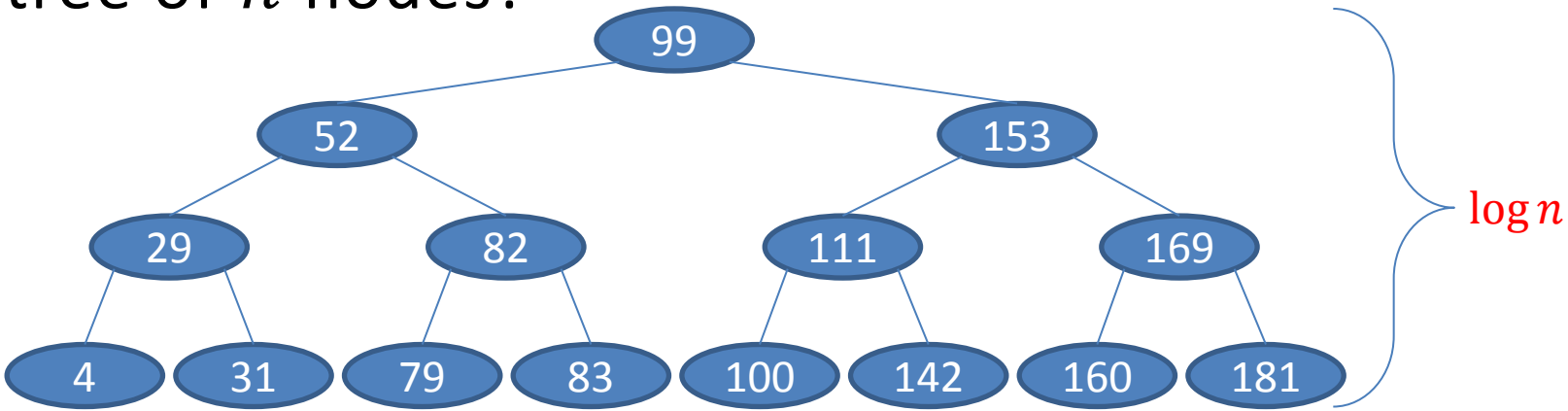
Search

- What is the shortest height possible for a binary tree of n nodes?



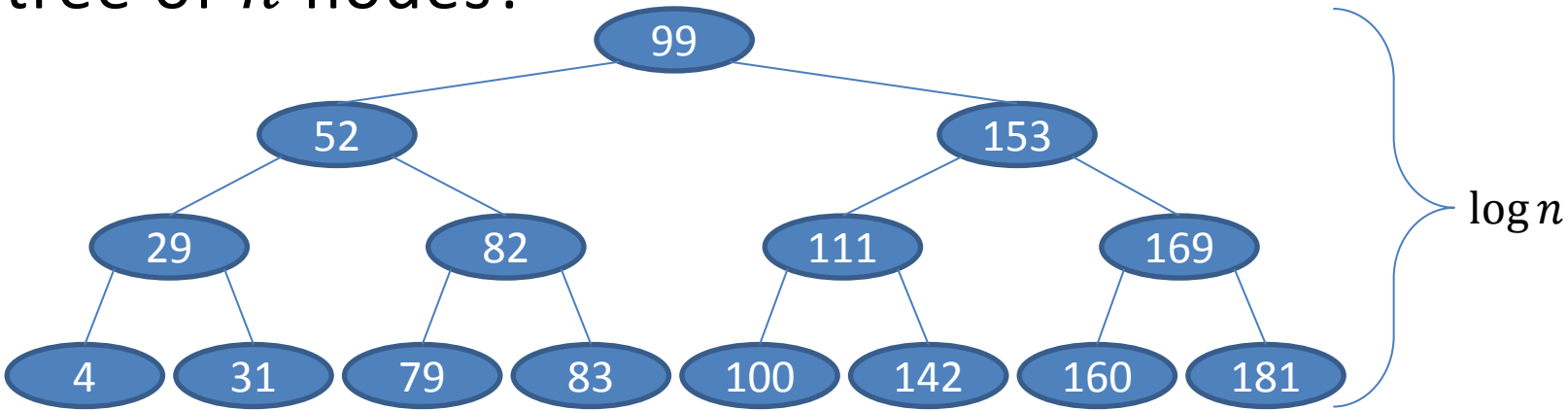
Search

- What is the shortest height possible for a binary tree of n nodes?



Search

- What is the shortest height possible for a binary tree of n nodes?



- Any search algorithm takes $\Omega(\log n)$

Divide and conquer search

- Starting with a sorted list of n integers and a target, the goal is to output the index of the target if it is in the list
- Break a problem into similar subproblems
- Solve each subproblem recursively
- Combine (non-recursive part)

Divide and conquer search

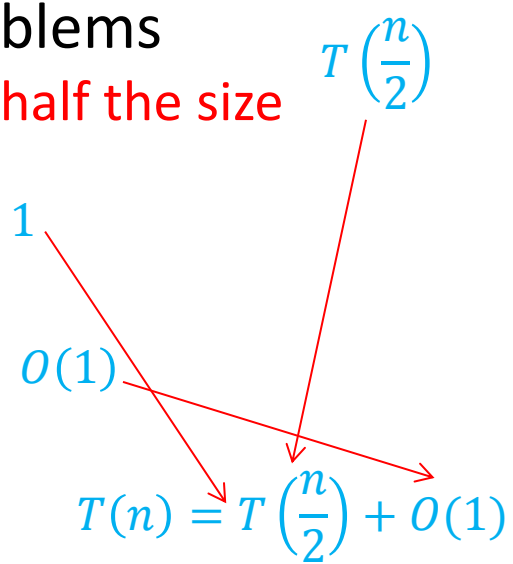
- Starting with a sorted list of n integers and a target, the goal is to output the index of the target if it is in the list
- Break a problem into similar subproblems
  - Split the list into two sublists each of half the size
- Solve each subproblem recursively
  - Recursively search one of the lists
- Combine (non-recursive part)
  - Compare and add index, if necessary

Divide and conquer search

- Starting with a sorted list of n integers and a target, the goal is to output the index of the target if it is in the list
- Break a problem into similar subproblems
  - Split the list into two sublists each of half the size
- Solve each subproblem recursively
  - Recursively search one of the lists
- Combine (non-recursive part)
  - Compare and add index, if necessary
- Time complexity?

Divide and conquer search

- Starting with a sorted list of n integers and a target, the goal is to output the index of the target if it is in the list
- Break a problem into similar subproblems
  - Split the list into two sublists each of half the size
- Solve each subproblem recursively
  - Recursively search one of the lists
- Combine (non-recursive part)
  - Compare and add index, if necessary



“Degenerate” divide and conquer

- The important aspect of divide and conquer is that the input size decreases in the recursive call, hopefully by a constant factor, not just one or two smaller
- Sometimes this can even happen when there is only one recursive call
  - Let’s call this the degenerate case, because we are not really “dividing” the input into parts

Master theorem applied to binary search

$$T(n) = aT(n/b) + O(n^d)$$

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

- The recursion for the runtime is

$$T(n) = T(n/2) + O(1)$$

- So, we have that $a = 1$, $b = 2$, and $d = 0$. In this case, $a = b^d$ so

$$T(n) \in O(n^0 \log n) = O(\log n)$$

Sorting

- Average time complexity?
  - Bubble sort
  - Insert sort
  - Selection sort
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort
  - Insert sort
  - Selection sort
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort
  - Selection sort
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort
  - Selection sort
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort $O(n^2)$
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort $O(n^2)$
  - Quicksort
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort $O(n^2)$
  - Quicksort $O(n \log n)$
  - Merge sort

Sorting

- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort $O(n^2)$
  - Quicksort $O(n \log n)$
  - Merge sort

Sorting

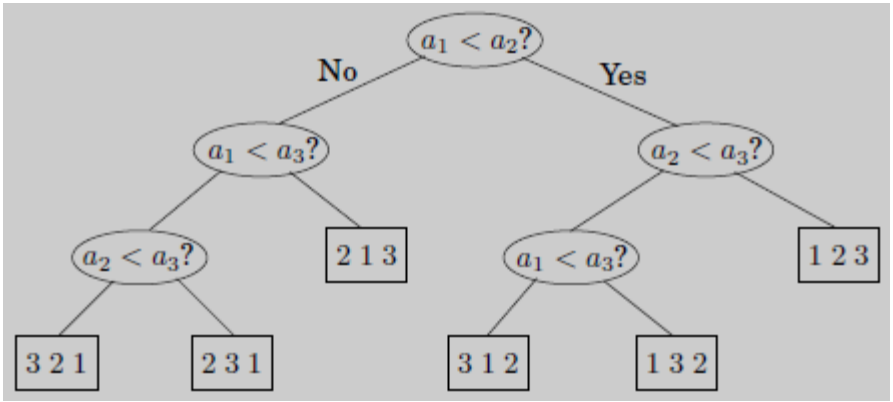
- Average time complexity?
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort $O(n^2)$
  - Quicksort $O(n \log n)$
  - Merge sort $O(n \log n)$

Sorting

- Average time complexity
  - Bubble sort $O(n^2)$
  - Insert sort $O(n^2)$
  - Selection sort $O(n^2)$
  - Quicksort $O(n \log n)$
  - Merge sort $O(n \log n)$

Sorting

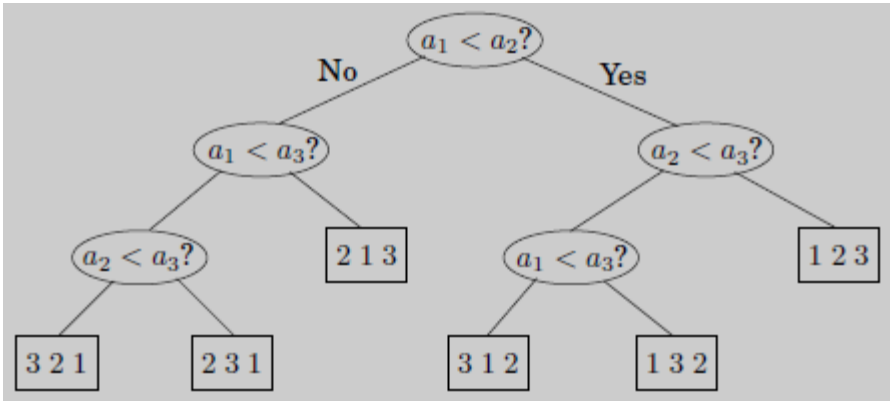
- How long should it take to sort things?
- If we must sort things based on comparisons, then we must travel down a path in this tree



$n = 3$

Sorting

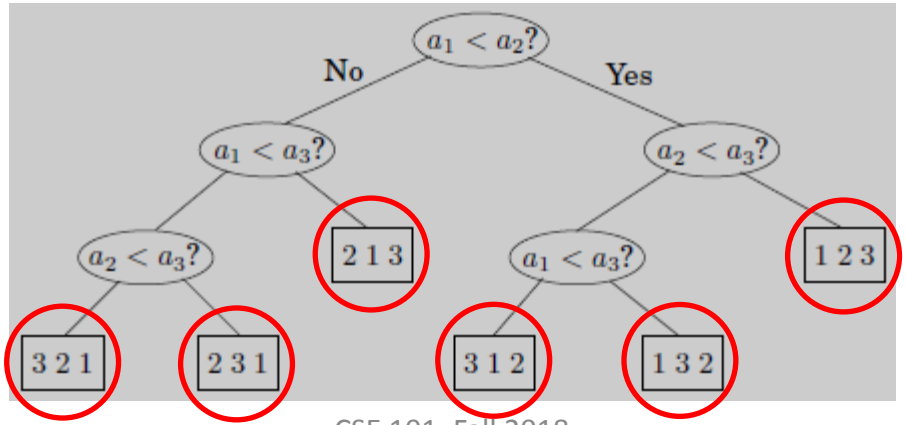
- How many leaves does the tree have?
- What is the height?



$n = 3$

Sorting

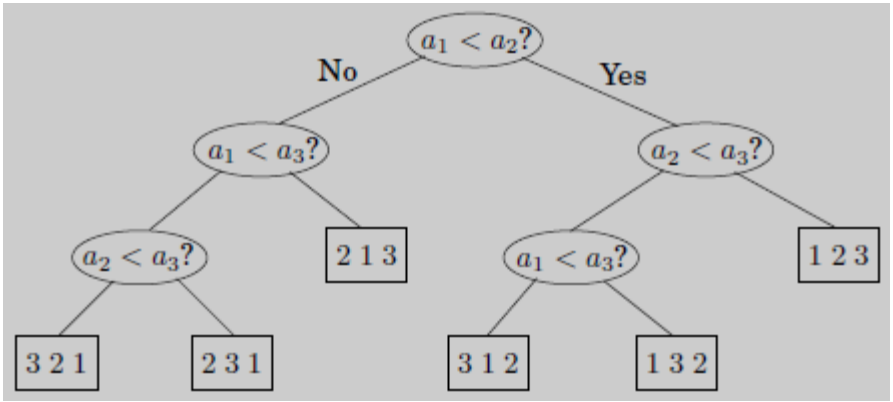
- How many leaves does the tree have? $n! = 6$
- What is the height?



$n = 3$

Sorting

- How many leaves does the tree have? $n! = 6$
- What is the height?

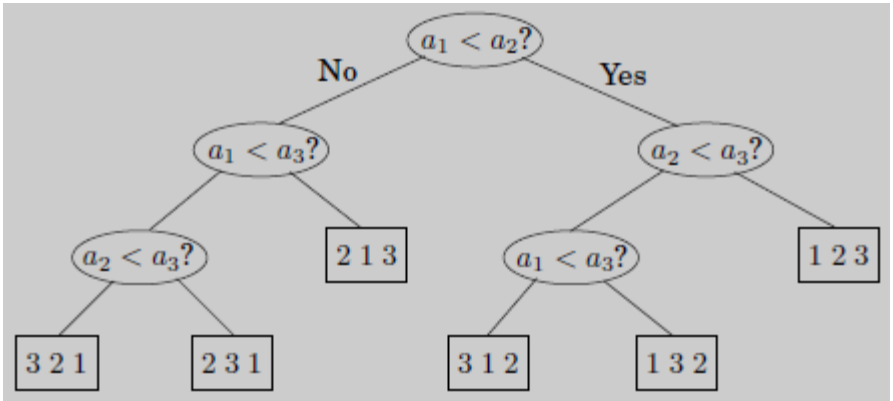


$n = 3$

Sorting

- How many leaves does the tree have? $n! = 6$
- What is the height? $\log(n!) = 2.59$

$\log(n!)$



$n = 3$

Sorting

- How many leaves does the tree have? $n!$
- What is the height? $\log(n!)$

Sorting

- How many leaves does the tree have? $n!$
- What is the height? $\log(n!)$

- Any sorting algorithm that relies on comparisons between elements runs in $\Omega(\log(n!))$ time

Sorting lower bound runtime

- Any sorting algorithm that relies on comparisons between elements runs in $\Omega(\log(n!))$ time

Sorting lower bound runtime

- Any sorting algorithm that relies on comparisons between elements runs in $\Omega(\log(n!))$ time
- Is there a simpler expression than $\log(n!)$?

Sorting lower bound runtime

- Any sorting algorithm that relies on comparisons between elements runs in $\Omega(\log(n!))$ time
- Is there a simpler expression than $\log(n!)$?

$$\log(n!) = \log(n) + \log(n-1) + \cdots + \log(2) + \log(1)$$

$$\log(n!) = \sum_{k=1}^{n} \log k$$

Sorting lower bound runtime

- Any sorting algorithm that relies on comparisons between elements runs in $\Omega(\log(n!))$ time
- Is there a simpler expression than $\log(n!)$?

$$\log(n!) = \log(n) + \log(n-1) + \cdots + \log(2) + \log(1) < n\log(n)$$

$$\log(n!) = \sum_{k=1}^{n} \log k = \Theta(n\log(n))$$

Sorting lower bound runtime

- What is the *fastest possible worst case* for any sorting algorithm?
  $\log(n!)$

Sorting lower bound runtime

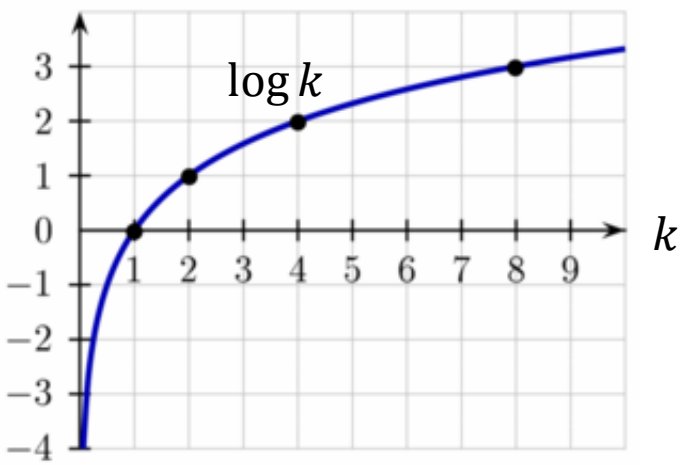
- What is the *fastest possible worst case* for any sorting algorithm? $\log(n!)$
- How big is that?

$$\log(n!) = \sum_{k=1}^{n} \log k$$

Sorting lower bound runtime

- What is the *fastest possible worst case* for any sorting algorithm? $\log(n!)$
- How big is that?

$$\log(n!) = \sum_{k=1}^{n} \log k$$



Sorting lower bound runtime

- What is the *fastest possible worst case* for any sorting algorithm? $\log(n!)$
- How big is that?

$$\log(n!) = \sum_{k=1}^{n} \log k$$

$$\int_1^n \log x \, dx = x \log x - x \Big|_1^n = (n \log n - n) - (0 - 1) = n \log n - n + 1$$

Sorting lower bound runtime

- What is the *fastest possible worst case* for any sorting algorithm? $\log(n!)$
- How big is that?

$$\log(n!) = \sum_{k=1}^{n} \log k$$

$$\int_1^n \log x \, dx = x \log x - x \Big|_1^n = (n \log n - n) - (0 - 1) = n \log n - n + 1$$

$$\log(n!) = \Omega(n \log n)$$

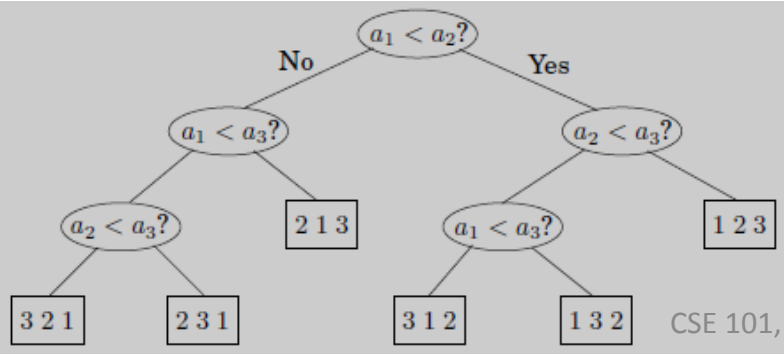
Sorting lower bound runtime

- Any sorting algorithm that relies on comparisons between elements runs in $\Omega(\log(n!))$ time
- There are sorting algorithms that run faster, but they rely on some prior knowledge about the elements, e.g., if you know the values are only allowed to come from a small range of values

Sorting 1, 2, 3 elements

- Sorting 1 element takes $\lceil \log(1!) \rceil = 0$ comparisons
- Sorting 2 elements takes $\lceil \log(2!) \rceil = 1$ comparison
- Sorting 3 elements should take $\lceil \log(3!) \rceil = 3$ comparisons
  - You could compare every pair of elements

    $\binom{3}{2} = 3$ comparisons
  - Or by the tree



Sorting 4 elements

- Sorting 4 elements should take $\lceil \log(4!) \rceil = 5$ comparisons
  - You could compare every pair of elements

    $\binom{4}{2} = 6$ comparisons
  - Exercise: sort 4 elements only using 5 comparisons

Divide and conquer sort

- Starting with a list of integers, the goal is to output the list in sorted order
- Break a problem into similar subproblems
- Solve each subproblem recursively
- Combine

Divide and conquer sort

- Starting with a list of integers, the goal is to output the list in sorted order
- Break a problem into similar subproblems
  - Split the list into two sublists each of half the size
- Solve each subproblem recursively
  - Recursively sort the two sublists
- Combine
  - Put the two sorted sublists together to create a sorted list of all the elements

mergesort algorithm

function mergesort($a[1 \ldots n]$)

if $n > 1$:

ML = mergesort$\left(a\left[1 \ldots \left\lfloor \frac{n}{2} \right\rfloor\right]\right)$

MR = mergesort $\left(a\left[\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots n\right]\right)$

return merge(ML,MR)

else:

return a

mergesort algorithm, runtime

function mergesort($a[1 \ldots n]$)

if $n > 1$:

ML = mergesort$\left(a\left[1 \ldots \left\lfloor \frac{n}{2} \right\rfloor\right]\right)$

MR = mergesort $\left(a\left[\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots n\right]\right)$

return merge(ML,MR)

else:

return a

Suppose mergesort runs in $T(n)$ time for inputs of length n. Then, each recursive call runs in $T\left(\frac{n}{2}\right)$ time and merge runs in $O(k + \ell)$ time where $k, \ell = n/2$, so merge runs in $O(n)$ time.

mergesort algorithm, runtime

function mergesort($a[1 \dots n]$) $T(n)$

if $n > 1$:

ML = mergesort$\left(a\left[1 \dots \left\lfloor\frac{n}{2}\right\rfloor\right]\right)$ $T\left(\frac{n}{2}\right)$

MR = mergesort $\left(a\left[\left\lfloor\frac{n}{2}\right\rfloor + 1, \dots n\right]\right)$ $T\left(\frac{n}{2}\right)$

return merge(ML,MR) $O(n)$

else:

return a

Suppose mergesort runs in $T(n)$ time for inputs of length n. Then, each recursive call runs in $T\left(\frac{n}{2}\right)$ time and merge runs in $O(k + \ell)$ time where $k, \ell = n/2$, so merge runs in $O(n)$ time.

mergesort algorithm, runtime

function mergesort($a[1 \dots n]$) $T(n)$

if $n > 1$:

ML = mergesort$\left(a\left[1 \dots \left\lfloor\frac{n}{2}\right\rfloor\right]\right)$ $T\left(\frac{n}{2}\right)$

MR = mergesort $\left(a\left[\left\lfloor\frac{n}{2}\right\rfloor + 1, \dots n\right]\right)$ $T\left(\frac{n}{2}\right)$

return merge(ML,MR) $O(n)$

else:

return a

Suppose mergesort runs in $T(n)$ time for inputs of length n. Then, each recursive call runs in $T\left(\frac{n}{2}\right)$ time and merge runs in $O(k + \ell)$ time where $k, \ell = n/2$, so merge runs in $O(n)$ time.

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

mergesort algorithm, runtime

function mergesort($a[1 \dots n]$) $T(n)$

if $n > 1$:

ML = mergesort$\left(a\left[1 \dots \left\lfloor\frac{n}{2}\right\rfloor\right]\right)$ $T\left(\frac{n}{2}\right)$

MR = mergesort $\left(a\left[\left\lfloor\frac{n}{2}\right\rfloor + 1, \dots n\right]\right)$ $T\left(\frac{n}{2}\right)$

return merge(ML,MR) $O(n)$

else:

return a

Suppose mergesort runs in $T(n)$ time for inputs of length n. Then, each recursive call runs in $T\left(\frac{n}{2}\right)$ time and merge runs in $O(k + \ell)$ time where $k, \ell = n/2$, so merge runs in $O(n)$ time.

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

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

by the master theorem

mergesort algorithm, correctness

- Correctness of divide and conquer algorithms is generally very easy. We use strong induction.
- Base case:

mergesort algorithm, correctness

- Correctness of divide and conquer algorithms is generally very easy. We use strong induction.
- Base case: $n = 1$, mergesort returns the original array a which is trivially sorted
- Inductive hypothesis:

mergesort algorithm, correctness

- Correctness of divide and conquer algorithms is generally very easy. We use strong induction.
- Base case: $n = 1$, mergesort returns the original array a which is trivially sorted
- Inductive hypothesis: suppose that for some $n \geq 1$, mergesort($a[1 \ldots k]$) outputs the elements of a in sorted order on all inputs of size k where $1 \leq k \leq n$. We want to show that it works for inputs of size $n + 1$.

mergesort algorithm, correctness

- Inductive hypothesis: suppose that for some $n \geq 1$, mergesort($a[1 \ldots k]$) outputs the elements of a in sorted order on all inputs of size k where $1 \leq k \leq n$. We want to show that it works for inputs of size $n + 1$.
  - Inductive step:

mergesort algorithm, correctness

- Inductive hypothesis: suppose that for some $n \geq 1$, mergesort($a[1 \ldots k]$) outputs the elements of a in sorted order on all inputs of size k where $1 \leq k \leq n$. We want to show that it works for inputs of size $n + 1$.
  - Inductive step: since $n \geq 1$ mergesort($a[1, \ldots, n + 1]$) returns merge(ML,MR) where ML = mergesort$\left(a\left[1 \ldots \left\lfloor \frac{n+1}{2} \right\rfloor\right]\right)$ and MR = mergesort $\left(a\left[\left\lfloor \frac{n+1}{2} \right\rfloor + 1, \ldots n + 1\right]\right)$

mergesort algorithm, correctness

- Inductive hypothesis: suppose that for some $n \geq 1$, mergesort($a[1 \ldots k]$) outputs the elements of a in sorted order on all inputs of size k where $1 \leq k \leq n$. We want to show that it works for inputs of size $n + 1$.
  - Inductive step: since $n \geq 1$ mergesort($a[1, \ldots, n+1]$) returns merge(ML,MR) where ML = mergesort$\left(a\left[1 \ldots \left\lfloor \frac{n+1}{2} \right\rfloor\right]\right)$ and MR = mergesort $\left(a\left[\left\lfloor \frac{n+1}{2} \right\rfloor + 1, \ldots n+1\right]\right)$
  - Since $\left\lfloor \frac{n+1}{2} \right\rfloor \leq n$, the inductive hypothesis ensures that ML and MR are sorted. And, merge combines two sorted lists so the algorithm returns the elements in sorted order.

Median

- The median of a list of numbers is the middle number in the sorted list.
- If the list has n values and n is odd, then the middle element is clear. It is the $\lceil n/2 \rceil$th smallest element.
- Example

$$med(8,2,9,11,4) = 8$$

because $n = 5$ and 8 is the $3rd = \lceil 5/2 \rceil$th smallest element of the list

Median

- The median of a list of numbers is the middle number in the list.
- If the list has n values and n is even, then there are two middle elements. Let's say that the median is the $n/2$th smallest element. Then, the median is also the $\lceil n/2 \rceil$th smallest element.
- Example

$$med(10,23,7,26,17,3) = 10$$

because $n = 6$ and 10 is the $3rd = \lceil 6/2 \rceil$th smallest element of the list

Median

- The purpose of the median is to summarize a set of numbers. The average is also a commonly used value. The median is more typical of the data.
- For example, suppose in a company with 20 employees, the CEO makes $1 million and all the other workers each make $50,000
- Then the average is $97,500 and the median is $50,000, which is much closer to the typical worker's salary

Median, algorithm

- Can you think of an efficient way to find the median?
  - How long would it take?
  - Is there a lower bound on the runtime of all median selection algorithms?

Median, algorithm

- Can you think of an efficient way to find the median?
  - How long would it take?
  - Is there a lower bound on the runtime of all median selection algorithms?
- Sort the list then find the $\lceil n/2 \rceil$th element $O(n \log n)$
- You can never have a faster runtime than $O(n)$ because you at least have to look at every element

Median, algorithm

- Can you think of an efficient way to find the median?
  - How long would it take?
  - Is there a lower bound on the runtime of all median selection algorithms?
- Sort the list then find the $\lceil n/2 \rceil$th element $O(n \log n)$
- You can never have a faster runtime than $O(n)$ because you at least have to look at every element
- All selection algorithms are $\Omega(n)$

Selection

- What if we designed an algorithm that takes as input a list of numbers of length n and an integer $1 \leq k \leq n$ and outputs the kth smallest integer in the list
- Then we could just plug in $\lceil n/2 \rceil$ for k and we could find the median!

Divide and conquer selection

- Let's think about selection in a divide and conquer type of way
- Break a problem into similar subproblems
- Solve each subproblem recursively
- Combine

Divide and conquer selection

- Let's think about selection in a divide and conquer type of way
- Break a problem into similar subproblems
  - Split the list into two sublists
- Solve each subproblem recursively
  - Recursively select from one of the sublists
- Combine
  - Determine how to split the list again

Divide and conquer selection

- How would you split the list?
  - Just splitting the list down the middle does not help so much
  - What we will do is pick a random "pivot" and split the list into all integers greater than the pivot and all that are less than the pivot
  - Then, we can determine which list to look in to find the kth smallest element
    - Note the value of k may change depending on which list we are looking in

Divide and conquer selection

- Example
  - Selection([40,31,6,51,76,58,97,37,86,31,19,30,68],7)
- Pick a random pivot, say 31. Then, divide the list into three groups SL, Sv, SR such that SL contains all elements smaller than 31, Sv is all elements equal to 31, and SR is all elements greater than 31.

Divide and conquer selection

- Example
  - Selection([40,31,6,51,76,58,97,37,86,31,19,30,68],7)
- Pick a random pivot, say 31. Then, divide the list into three groups SL, Sv, SR such that SL contains all elements smaller than 31, Sv is all elements equal to 31, and SR is all elements greater than 31.

- SL=[6,19,30], size = 3
- Sv=[31,31], size = 2
- SR=[40,51,76,58,97,37,86,68], size = 8

Divide and conquer selection

- Example
  - Selection([40,31,6,51,76,58,97,37,86,31,19,30,68],7)

- SL=[6,19,30], size = 3
- Sv=[31,31], size = 2
- SR=[40,51,76,58,97,37,86,68], size = 8

- Since k=7 is bigger than the size of SL, we know the kth biggest element cannot be in SL. Since k=7 it is bigger than size of SL plus size of Sv, it cannot be in Sv either. Therefore it must be in SR.

Divide and conquer selection

- Example
  - Selection([40,31,6,51,76,58,97,37,86,31,19,30,68],7)

- SL=[6,19,30], size = 3
- Sv=[31,31], size = 2
- SR=[40,51,76,58,97,37,86,68], size = 8

- Since k=7 is bigger than the size of SL, we know the kth biggest element cannot be in SL. Since k=7 it is bigger than size of SL plus size of Sv, it cannot be in Sv either. Therefore it must be in SR.
- So the 7^{th} biggest element in the original list is what number in SR?

Divide and conquer selection

- Example
  - Selection([40,31,6,51,76,58,97,37,86,31,19,30,68],7)

- SL=[6,19,30], size = 3
- Sv=[31,31], size = 2
- SR=[40,51,76,58,97,37,86,68], size = 8

- Since k=7 is bigger than the size of SL, we know the kth biggest element cannot be in SL. Since k=7 it is bigger than size of SL plus size of Sv, it cannot be in Sv either. Therefore it must be in SR.
- So the 7^{th} biggest element in the original list is what number in SR? 7-3-2 = 2

Divide and conquer selection

- Example
  - Selection([40,31,6,51,76,58,97,37,86,31,19,30,68],7)

- SL=[6,19,30], size = 3
- Sv=[31,31], size = 2
- SR=[40,51,76,58,97,37,86,68], size = 8

So, the 7th biggest element in the original list is the 2nd biggest in SR?

- Selection([40,31,6,51,76,58,97,37,86,31,19,30,68],7) = Selection ([40,51,76,58,97,37,86,68],2) = 40

Selection algorithm

- Input: list of n integers and integer k, where $1 \le k \le n$
- Output: the kth biggest number in the set of integers

```
function Selection(a[1…n],k)
if n==1:
        return a[1]
pick a random integer in the list v
Split the list into sets SL, Sv, SR
if k≤|SL|:
        return Selection(SL,k)
else if k≤|SL|+|Sv|:
        return v
else:
        return Selection(SR, k-|SL|-|Sv|)
```

Selection algorithm, runtime

- Input: list of n integers and integer k, where $1 \le k \le n$
- Output: the kth biggest number in the set of integers

function Selection(a[1…n],k) $T(n)$
if n==1:
    return a[1] $O(1)$
pick a random integer in the list v $O(1)$
Split the list into sets SL, Sv, SR $O(n)$
if k≤|SL|:
    return Selection(SL,k) $T(|SL|)$
else if k≤|SL|+|Sv|: $O(1)$
    return v
else:
    return Selection(SR, k-|SL|-|Sv|) $T(|SR|)$

Selection algorithm, runtime

- Input: list of n integers and integer k, where 1 ≤ k ≤ n
- Output: the kth biggest number in the set of integers

function Selection(a[1…n],k) $T(n)$
if n==1:
    return a[1] $O(1)$
pick a random integer in the list v $O(1)$
Split the list into sets SL, Sv, SR $O(n)$
if k≤|SL|:
    return Selection(SL,k) $T(|SL|)$
else if k≤|SL|+|Sv|: $O(1)$
    return v
else:
    return Selection(SR, k-|SL|-|Sv|) $T(|SR|)$

The runtime is dependent on |SL| and |SR|

Selection algorithm, runtime

- The runtime is dependent on |SL| and |SR|
- If we were so lucky as to choose v to be close to the median every time, then $|SL| \approx |SR| \approx n/2$. And so, no matter which set we recurse on,

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

Selection algorithm, runtime

- The runtime is dependent on |SL| and |SR|
- If we were so lucky as to choose v to be close to the median every time, then $|SL| \approx |SR| \approx n/2$. And so, no matter which set we recurse on,

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

$$T(n) = O(n)$$ by the master theorem

Selection algorithm, runtime

- The runtime is dependent on |SL| and |SR|
- Conversely, if we were so unlucky as to choose v to be the maximum (or minimum) then |SL| (or |SR|) = n-1 and

$$T(n) = T(n-1) + O(n)$$

Selection algorithm, runtime

- The runtime is dependent on |SL| and |SR|
- Conversely, if we were so unlucky as to choose v to be the maximum (or minimum) then |SL| (or |SR|) = n-1 and

$$T(n) = T(n-1) + O(n)$$
$$T(n) = O(n^2)$$

- Which is worse than sorting then finding

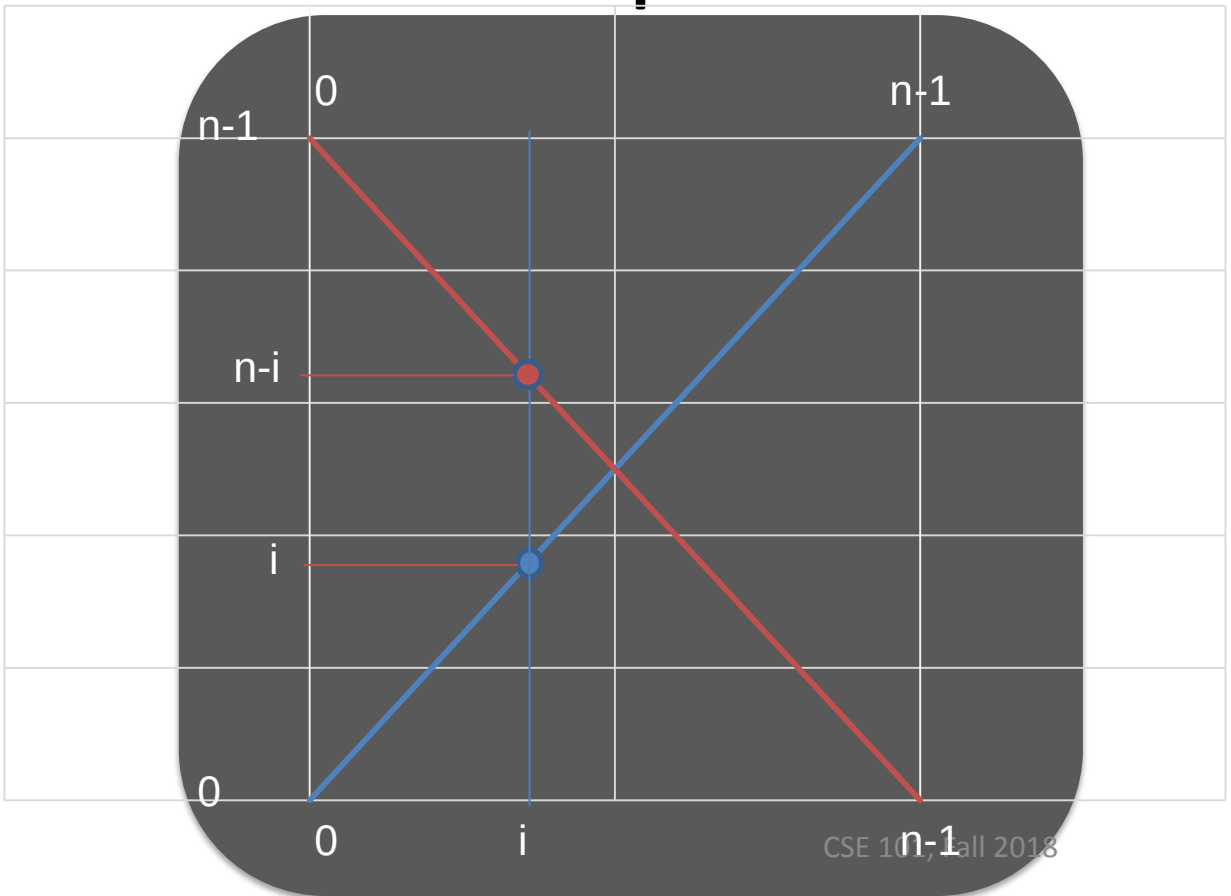
Selection algorithm, runtime

- The runtime is dependent on |SL| and |SR|
- Conversely, if we were so unlucky as to choose v to be the maximum (or minimum) then |SL| (or |SR|) = n-1 and

$$T(n) = T(n-1) + O(n)$$
$$T(n) = O(n^2)$$

- Which is worse than sorting then finding
- Although there is a chance of having a high runtime, is it still worth it?

Expected runtime

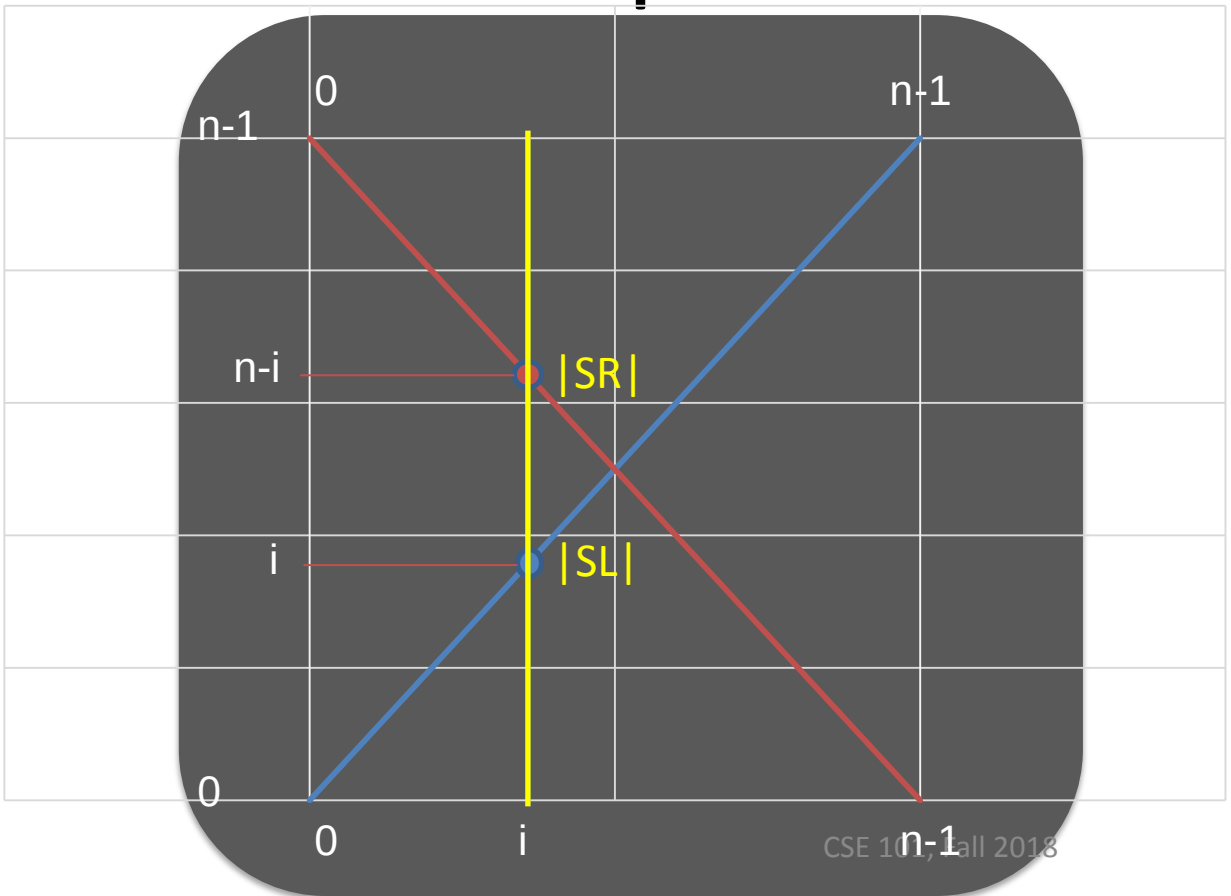


If you randomly select the ith element, then your list will be split into a list of length i and a list of length n-i

So when we recurse on the smaller list, it will take time proportional to

$$\max(i, n-i)$$

Expected runtime

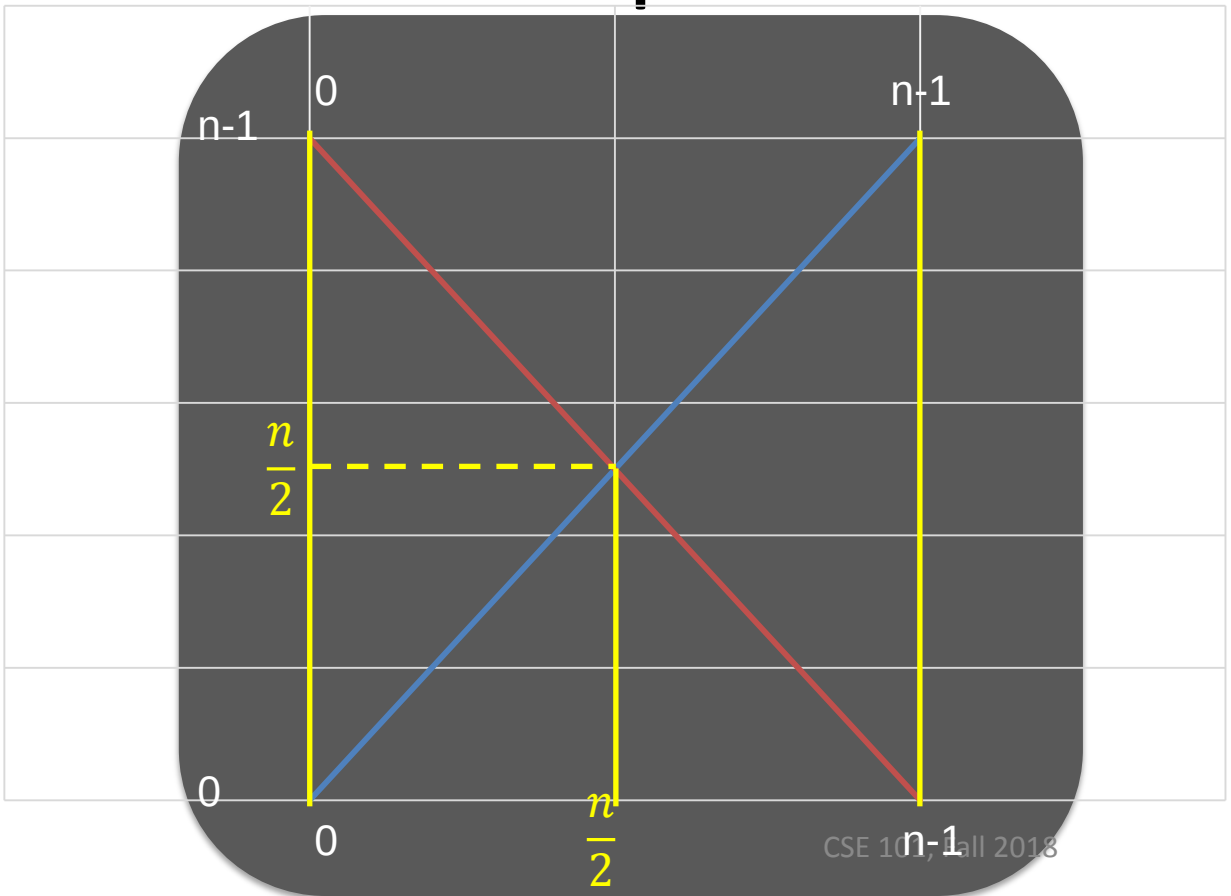


If you randomly select the ith element, then your list will be split into a list of length i and a list of length n-i

So when we recurse on the smaller list, it will take time proportional to

$$\max(i, n-i)$$

Expected runtime



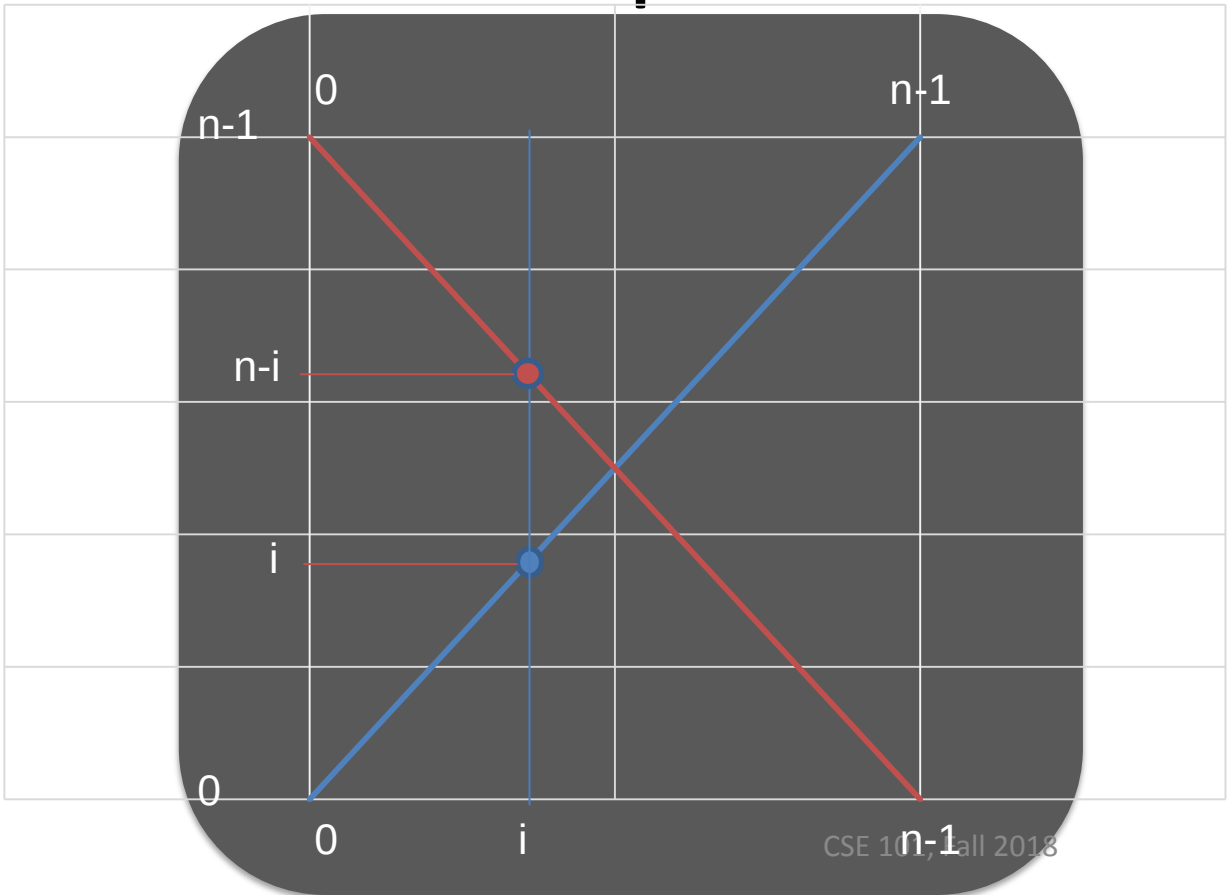
Clearly, the split with the smallest maximum size is when

i=n/2

and worst case is

i=n or i=1

Expected runtime

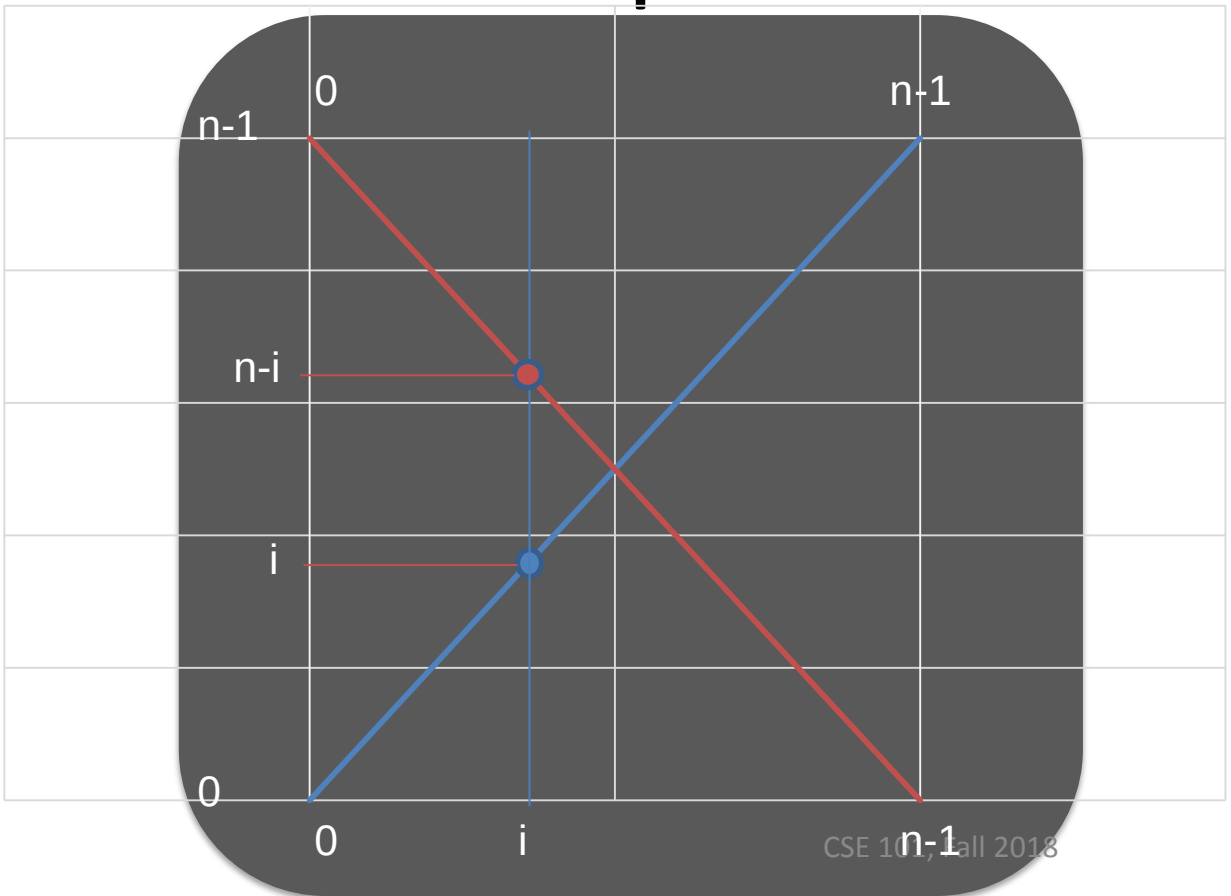


What is the expected runtime?

Well, what is our random variable?

For each input and sequence of random choices of pivots, the random variable is the runtime of that particular outcome

Expected runtime

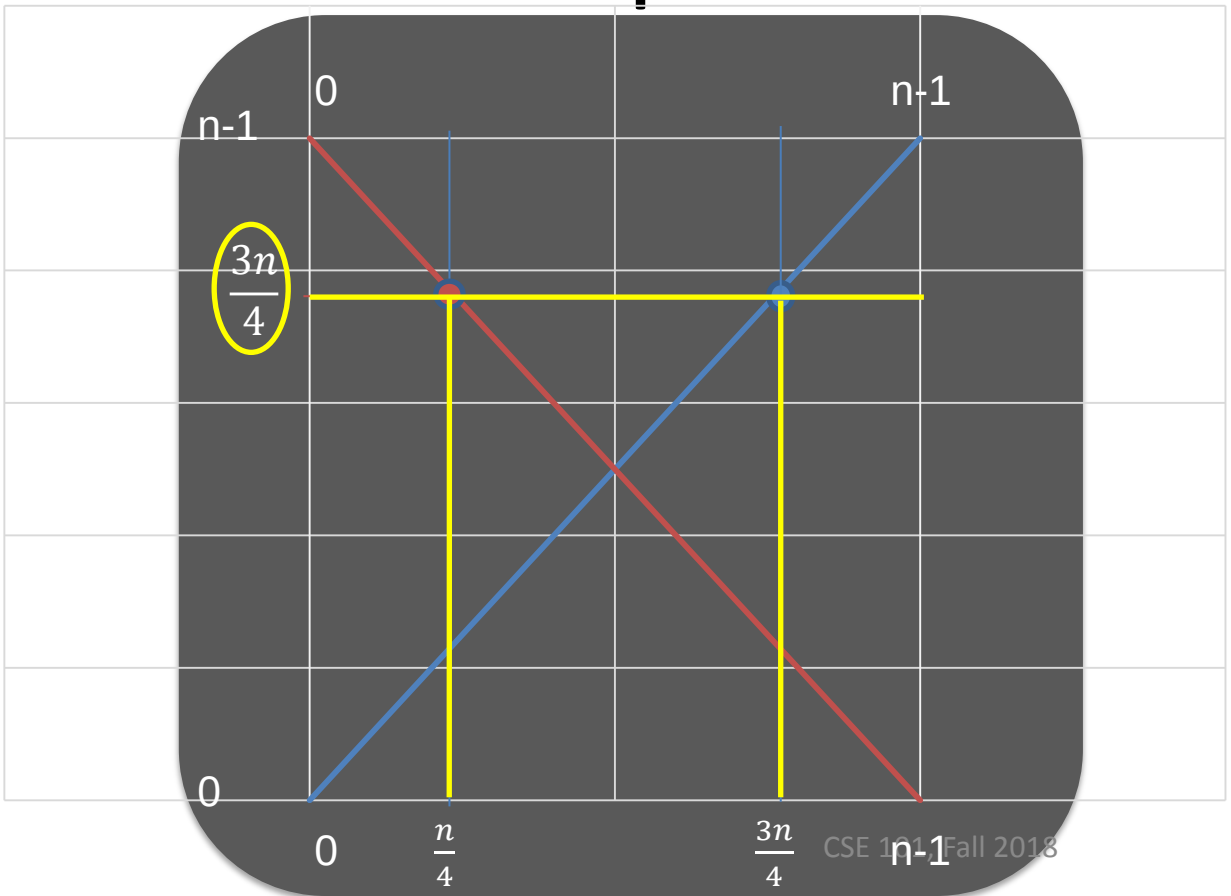


So, if we want to find the expected runtime, then we must sum over all possibilities of choices

Let $ET(n)$ be the expected runtime. Then

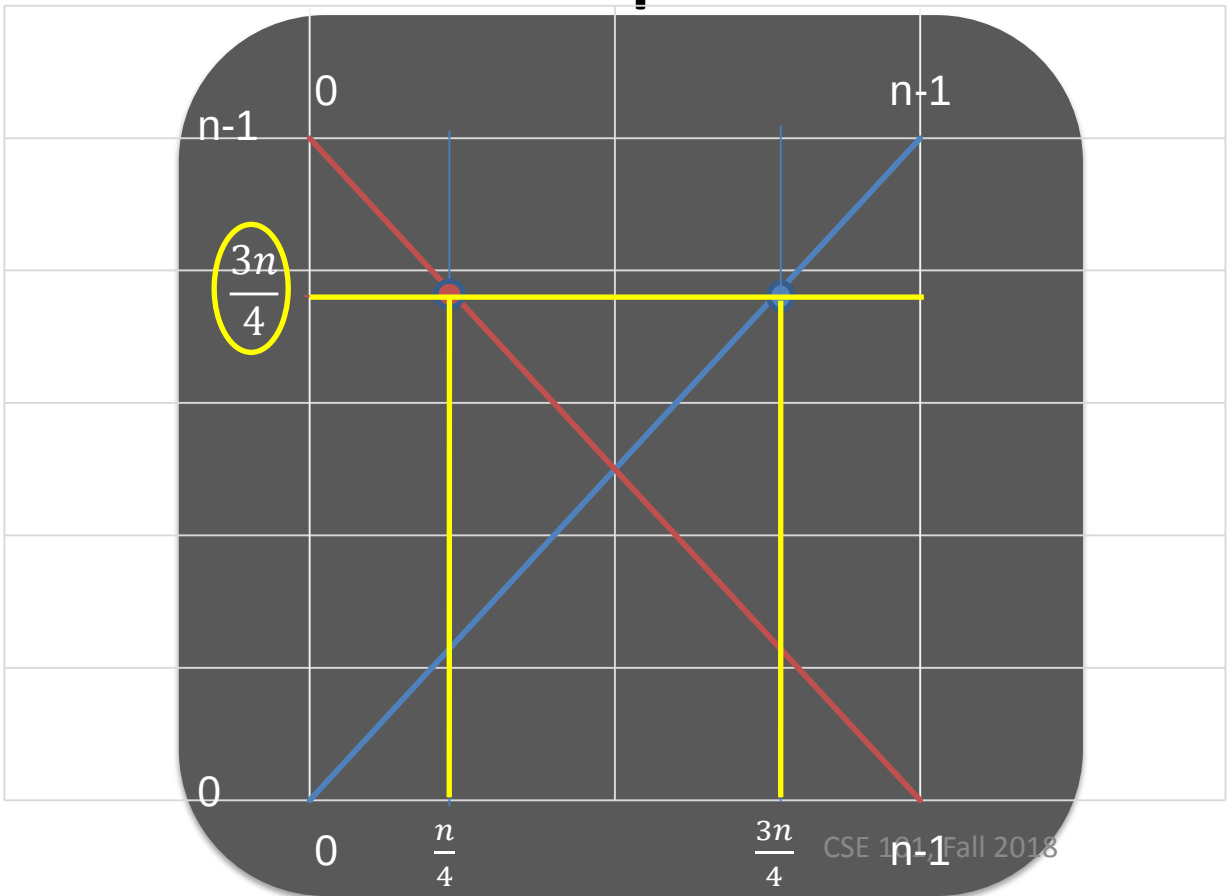
$$ET(n) = \frac{1}{n}\sum_{i=1}^{n} ET(\max(i, n-i)) + O(n)$$

Expected runtime



What is the probability of choosing a value from 1 to n in the interval $\left[\frac{n}{4}, \frac{3n}{4}\right]$ if all values are equally likely?

Expected runtime



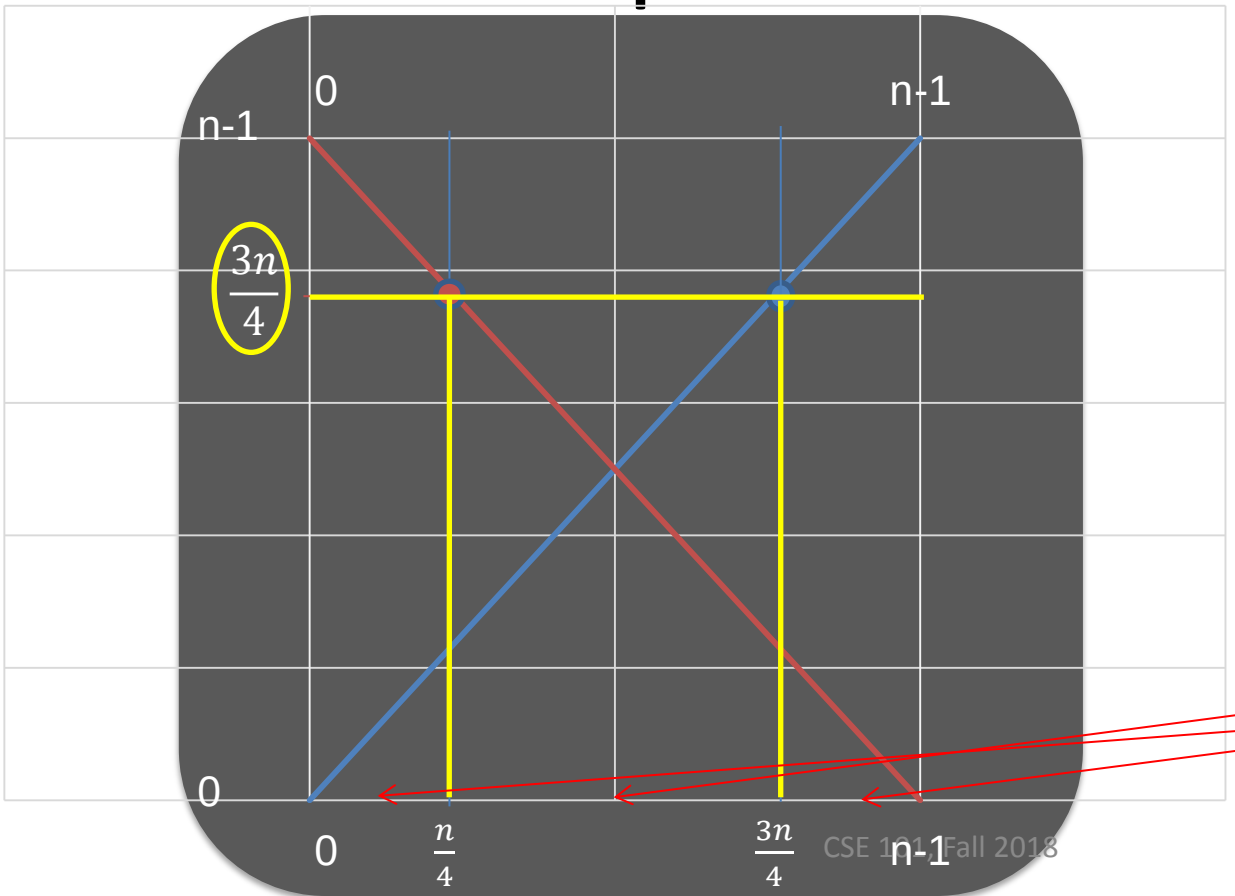
If you did choose a value between n/4 and 3n/4 then the sizes of the subproblems would both be $\leq \frac{3n}{4}$

Otherwise, the subproblems would be $\leq n$

So, we can compute an upper bound on the expected runtime.

$$ET(n) \leq \frac{1}{2}ET\left(\frac{3n}{4}\right) + \frac{1}{2}ET(n) + O(n)$$

Expected runtime



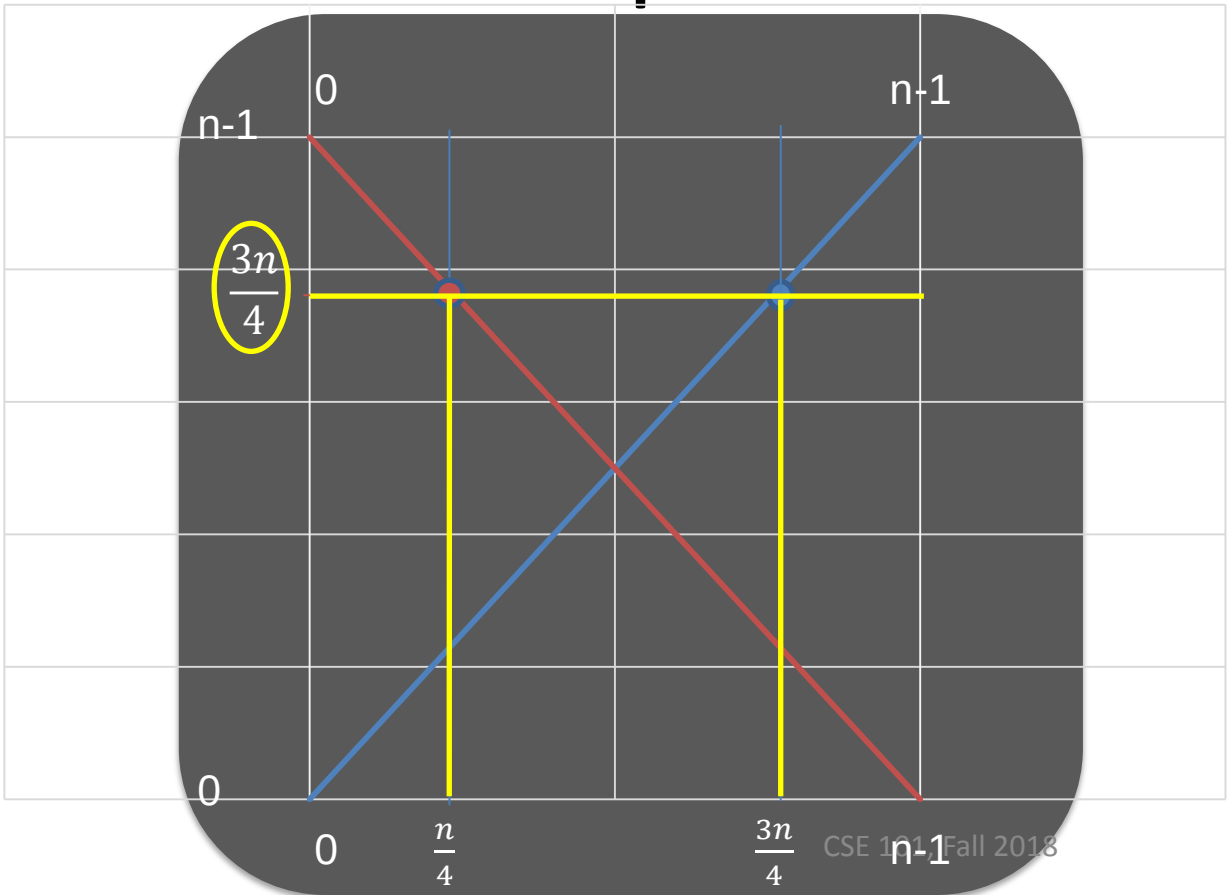
If you did choose a value between n/4 and 3n/4 then the sizes of the subproblems would both be $\leq \frac{3n}{4}$

Otherwise, the subproblems would be $\leq n$

So, we can compute an upper bound on the expected runtime.

$$ET(n) \leq \frac{1}{2}ET\left(\frac{3n}{4}\right) + \frac{1}{2}ET(n) + O(n)$$

Expected runtime



$$ET(n) \leq \frac{1}{2}ET\left(\frac{3n}{4}\right) + \frac{1}{2}ET(n) + O(n)$$

$$ET(n) \leq ET\left(\frac{3n}{4}\right) + O(n)$$

Plug into the master theorem with a=1, b=4/3, d=1

$a<b^d$ so

$$ET(n) \leq O(n)$$

Quicksort

- What have we noticed about the partitioning part of Selection?
- After partitioning, the “pivot” is in its correct position in sorted order
- Quicksort takes advantage of that

Divide and conquer quicksort

- Let's think about selection in a divide and conquer type of way
- Break a problem into similar subproblems
  - Split the list into two sublists by partitioning a pivot
- Solve each subproblem recursively
  - Recursively sort each sublist
- Combine
  - Concatenate the lists

Divide and conquer quicksort

procedure quicksort(a[1…n])

if $n \leq 1$:

    return a

set v to be a random element in a

partition a into SL,Sv,SR

return quicksort(SL)∘Sv∘ quicksort(SR)

Divide and conquer quicksort, runtime

procedure quicksort(a[1…n])

if n≤1:

return a

set v to be a random element in a

partition a into SL,Sv,SR

return quicksort(SL)∘Sv∘ quicksort(SR)

$T(n) = T(|SL|) + T(|SR|) + O(n)$

$O(n)$

$T(|SL|)$

$T(|SR|)$

Divide and conquer quicksort, runtime

$$ET(n) = \frac{1}{n}\left(\sum_{i=1}^{n} ET(n-i) + ET(i) + O(n)\right)$$

$|SR|$ $|SL|$

$$ET(n) = \frac{2}{n}\sum_{i=1}^{n} ET(i) + O(n)$$

$$ET(n) = O(n \log n)$$ Exercise

Selection (deterministic)

- Sometimes this algorithm we have described is called quick select because generally it is a very practical linear expected time algorithm. This algorithm is used in practice.
- For theoretic computer scientists, it is unsatisfactory to only have a randomized algorithm that could run in quadratic time
- Blum, Floyd, Pratt, Rivest, and Tarjan have developed a deterministic approach to finding the median (or any kth biggest element
  - They use a divide and conquer strategy to find a number close to the median and then use that to pivot the values

Selection (deterministic)

- The strategy is to split the list into sets of 5 and find the medians of all those sets. Then, find the median of the medians using a recursive call T(n/5).
- Then, partition the set just like in quickselect and recurse on SR or SL just like in quickselect

Median of medians

MofM(L,k)

If L has 10 or fewer elements:

    Sort(L) and return the kth element

Partition L into sublists S[i] of five elements each

For $i = 1, \ldots n/5$

    $m[i]$ =MofM(S[i],3)

M = MofM($[m[1], \ldots, m[n/5]]$,$n/10$)

Median of medians, runtime

MofM(L,k) $T(n)$

If L has 10 or fewer elements:

    Sort(L) and return the kth element

Partition L into sublists S[i] of five elements each

For $i = 1, \ldots n/5$ } $n/5$ iterations, so $O(n)$

    $m[i]$ =MofM(S[i],3) $O(1)$

M = MofM($[m[1], \ldots, m[n/5]]$,$n/10$) $T(n/5)$

Median of medians, runtime

MofM(L,k) $T(n)$

If L has 10 or fewer elements:

    Sort(L) and return the kth element

Partition L into sublists S[i] of five elements each

For $i = 1, \ldots n/5$

    $m[i]$ =MofM(S[i],3) $O(1)$

$n/5$ iterations, so $O(n)$

M = MofM($[m[1], \ldots, m[n/5]], n/10$) $T(n/5)$

Split the list into sets SL, Sv, SR

if k≤|SL|:

    return MofM(SL,k) $T(|SL|)$

else if k≤|SL|+|Sv|:

    return v

else:

    return MofM(SR, k-|SL|-|Sv|) $T(|SR|)$

$T\left(\frac{7n}{10}\right)$

$$|SL| \leq \frac{7n}{10}$$

$$|SR| \leq \frac{7n}{10}$$

Selection (deterministic)

- By construction, it can be shown that |SR|<7n/10 and |SL|<7n/10 and so no matter which set we recurse on, we have

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

$$T(n) \approx T\left(\frac{9n}{10}\right) + O(n)$$

$$T(n) \approx O(n)$$

- You cannot use the master theorem to solve this, but you can use induction to show that if $T(n) \leq cn$ for some c, then $T(n) \leq cn$
- And so we have a linear time selection algorithm!

Next lecture

- Divide and conquer algorithms
  - Reading: Chapter 2