# CSC 225

Algorithms and Data Structures I
Fall 2014
Rich Little

# Algorithm Design Technique
# Divide and Conquer: Quicksort

- Mergesort divides the input set according to the position of the elements (i.e., first and second part of sequence)
- Quicksort divides the input set according to the value of the elements

**http://en.wikipedia.org/wiki/Quicksort**

# Quicksort

- The input data are stored in an array *A*[*L*..*R*] where *L* and *R* are the leftmost and rightmost indices of the data in this array
- Approach: Partition the input set into two (ideally) equal-sized subsets $S_1$ and $S_2$ using a *pivot* (i.e., typically a value from the input data)
- Apply the algorithm recursively for the two subsets $S_1$ and $S_2$ until size 1 is reached

# Algorithm QuickSort($A[L..R]$)

```
if A.length > 1 then
  p ← pickPivot(A[L..R])
  M ← partition(A[L..R], A[p])
  QuickSort(A[L..M])
  QuickSort(A[M+1..R])
end
```

# Pivot Computation

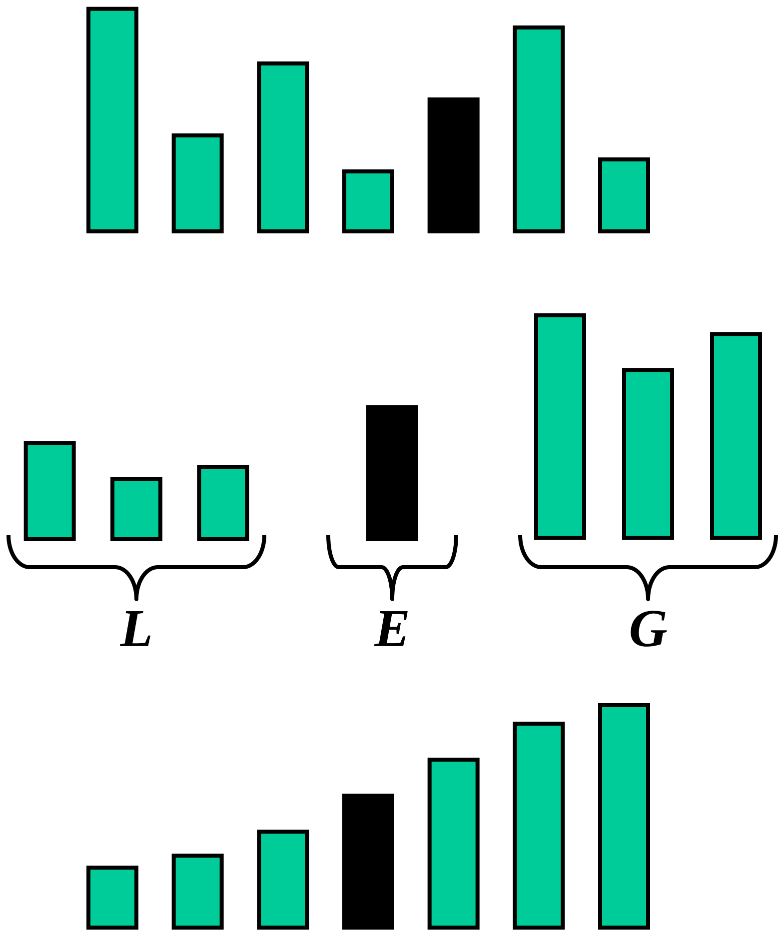
- Picking a pivot should be a $O(1)$ operation
- The median is the perfect pivot; computing the median takes $O(n)$ time
- Any value close to the median is still a good pivot
- The largest or smallest value would be a bad pivot, because it would split the array into subarrays of size 1 and n-1
- Constant time approaches for picking a pivot $p$
  - ➢ First element in subarray $A[L]$
  - ➢ Last element in subarray $A[L]$
  - ➢ Middle element of subarray $A[(L+R)/2]$
  - ➢ Average of three elements $(A[L] + A[R] + [(L+R)/2])/3$
  - ➢ Compute the average of 5 or 7 elements
  - ➢ Randomized selection of pivot—randomly select index in range L..R

# Why is Quicksort so fast?

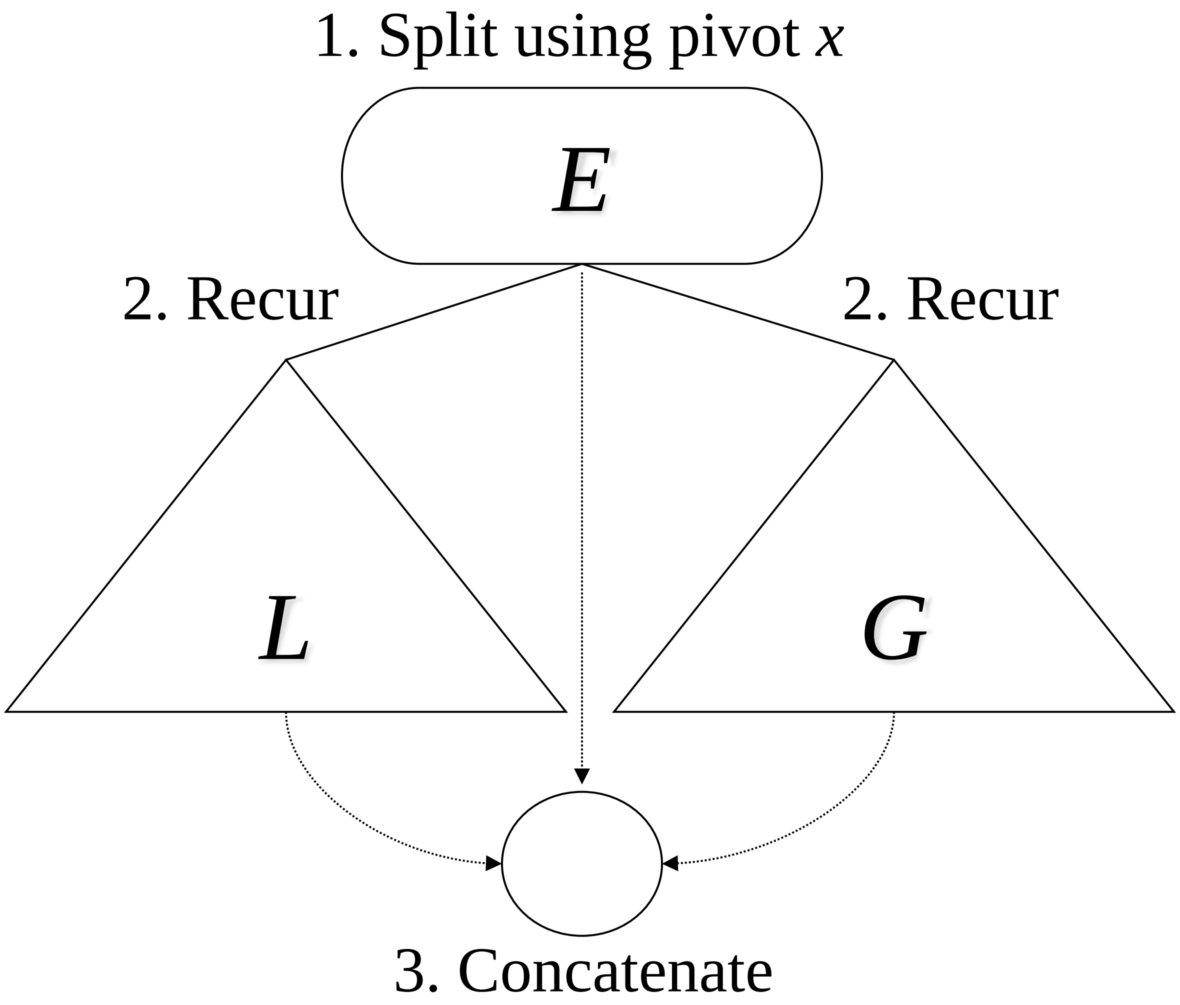
- In practice Quicksort runs in $O(n \log n)$ and almost never exhibits its worst-case behaviour of $O(n^2)$
- Moreover, Quicksort performs better than $O(n \log n)$ worst-case sorting algorithms
- The actual running time makes the difference
  - $T_{Quick}(n) = 1.18\ n \log n$
  - $T_{Heap}(n) = 2.22\ n \log n$
- Sorting out sorting
  - http://www.youtube.com/watch?v=SJwEwA5gOkM

# Quicksort as discussed in Textbook based on ADT Sequence

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element ***x*** (called pivot) and partition ***S*** into
    - ***L*** elements less than ***x***
    - ***E*** elements equal ***x***
    - ***G*** elements greater than ***x***
  - Recur: sort ***L*** and ***G***
  - Conquer: join ***L***, ***E*** and ***G***



# Quicksort Algorithm



# Algorithm split($L$, $E$, $G$, $S$, $x$)

- Let $L$, $E$, and $G$ be empty sequences.
- Insert in $L$ (and remove from $S$) all elements from $S$ that are less than $x$.
- Insert in $E$ (and remove from $S$) all elements from $S$ that are equal to $x$.
- Insert in $G$ (and remove from $S$) all elements from $S$ that are greater than $x$.
- $S$ is empty.

How fast can we implement algorithm split?

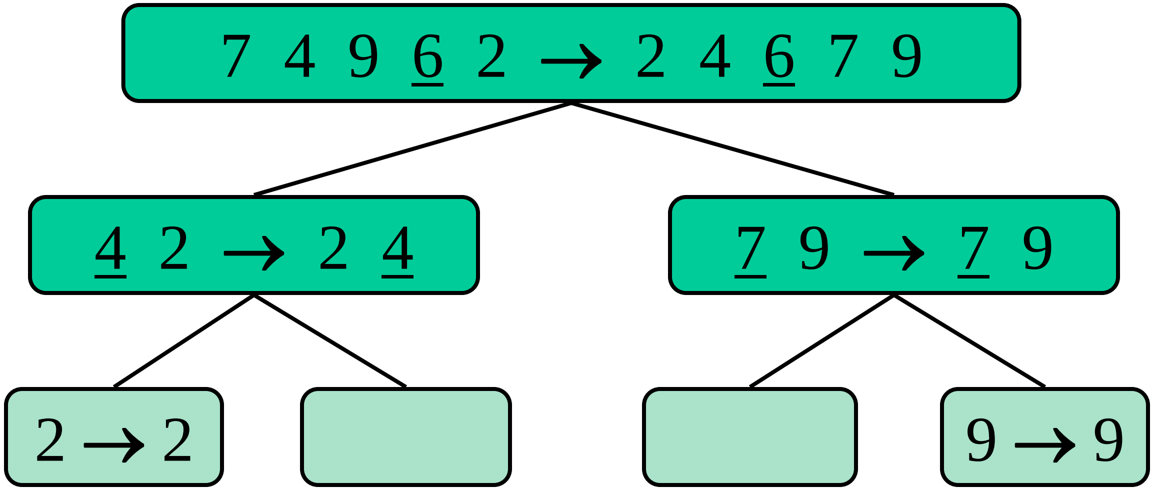
# Algorithm concatenate(*L*, *E*, *G*, *S*)

- Let *S* be an empty sequence.
- Put the elements back into *S* in order by first inserting the elements of *L*, then those of *E*, and finally those of *G*.

How fast can we implement concatenate?

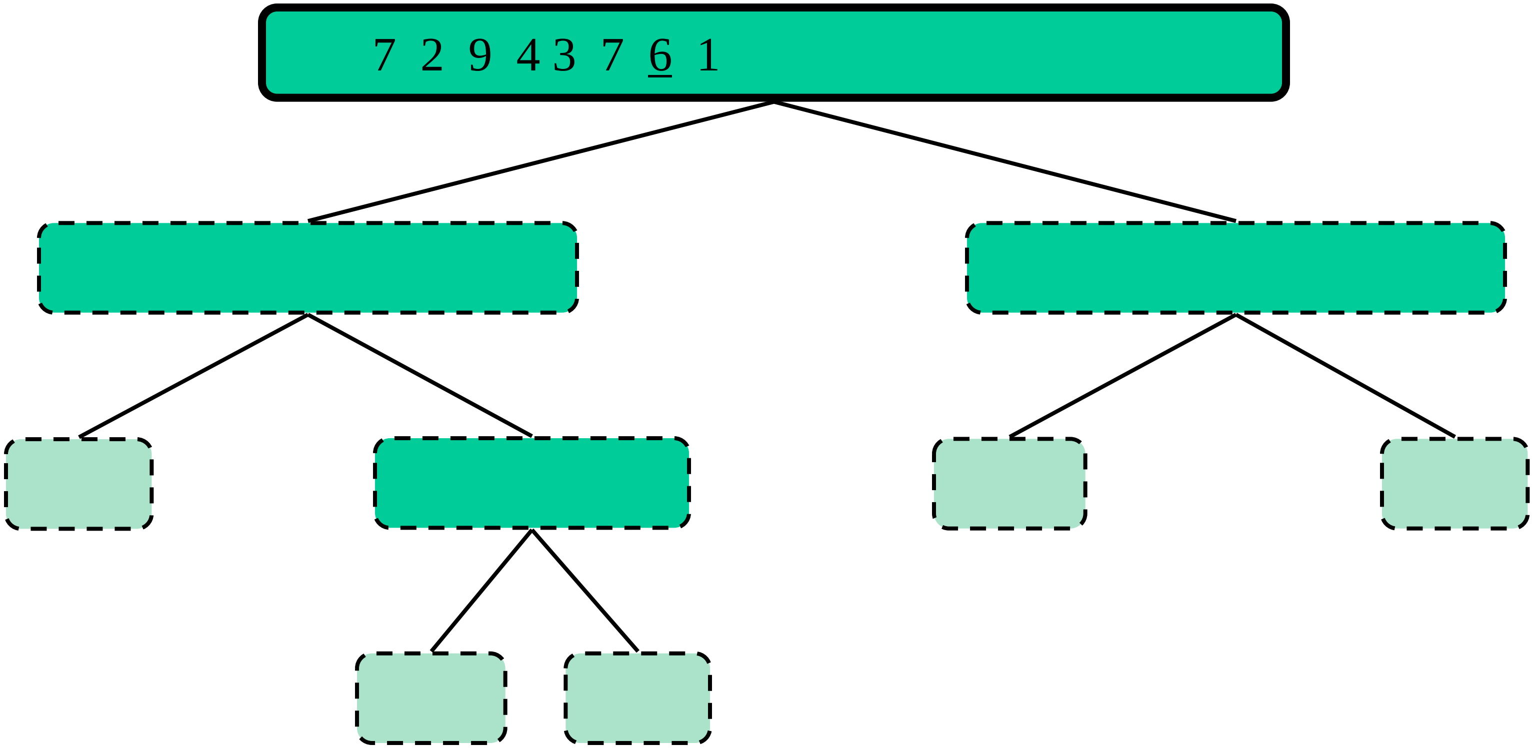
# Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - ➢ Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - ➢ The root is the initial call
  - ➢ The leaves are calls on subsequences of size 0 or 1



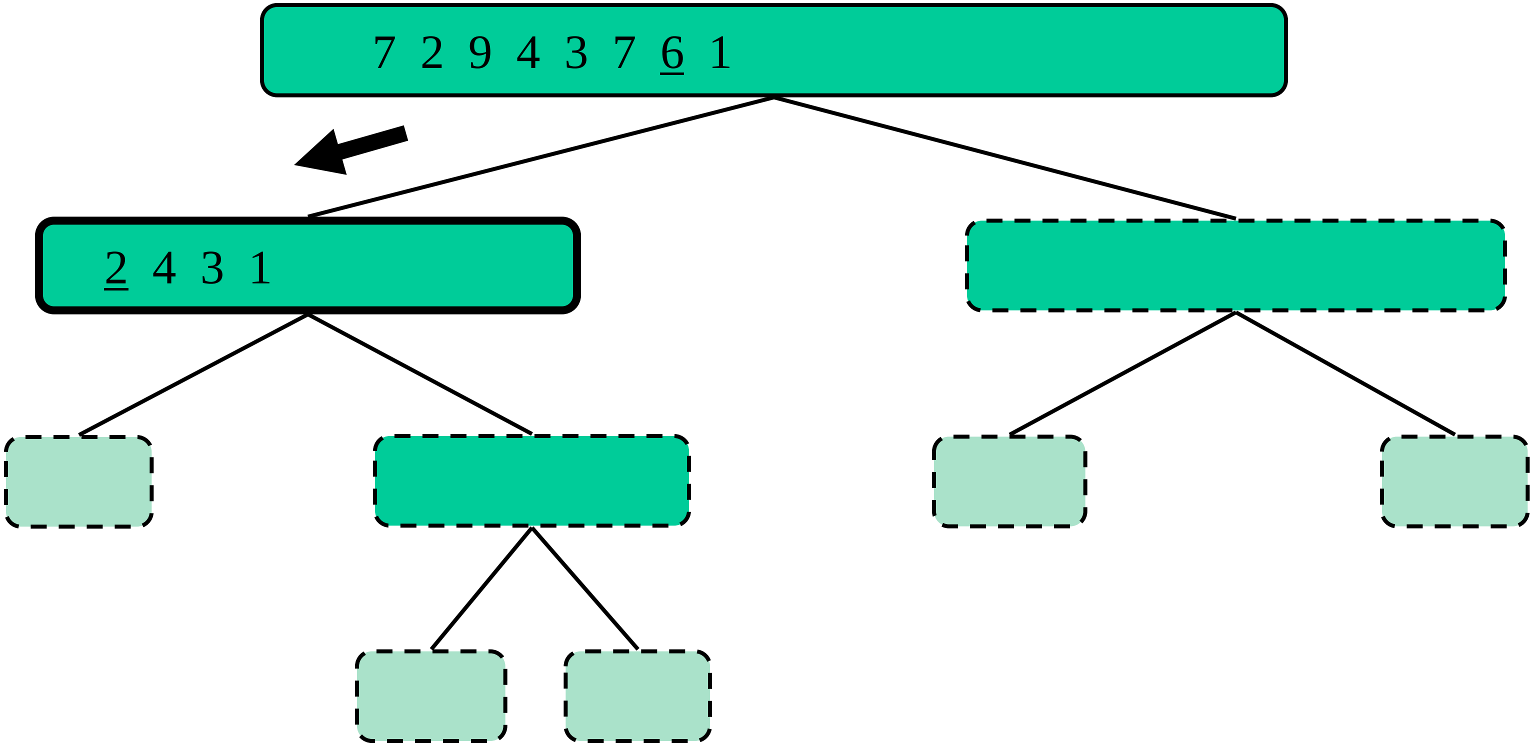
# Execution Example

- Pivot selection



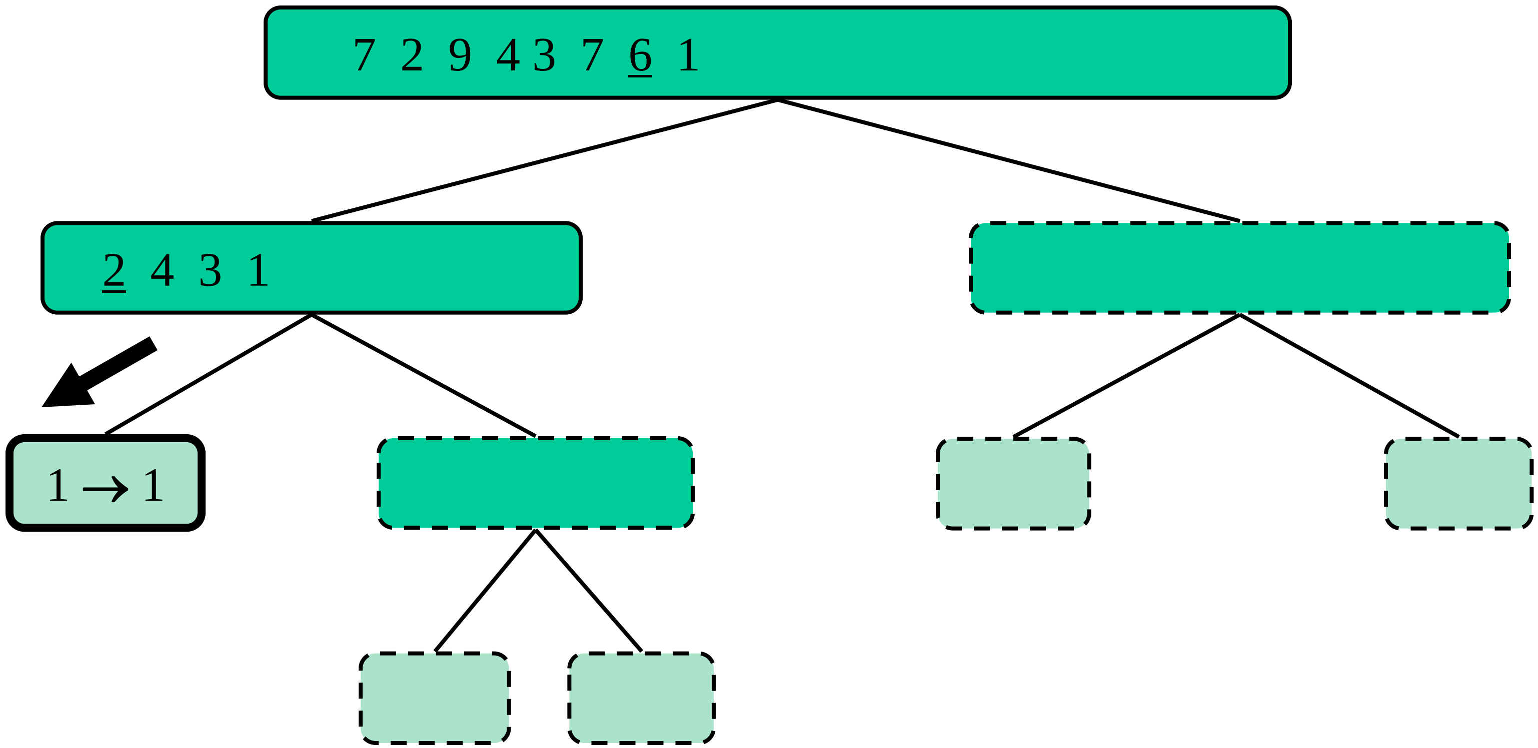
# Execution Example (cont.)

- Partition, recursive call, pivot selection



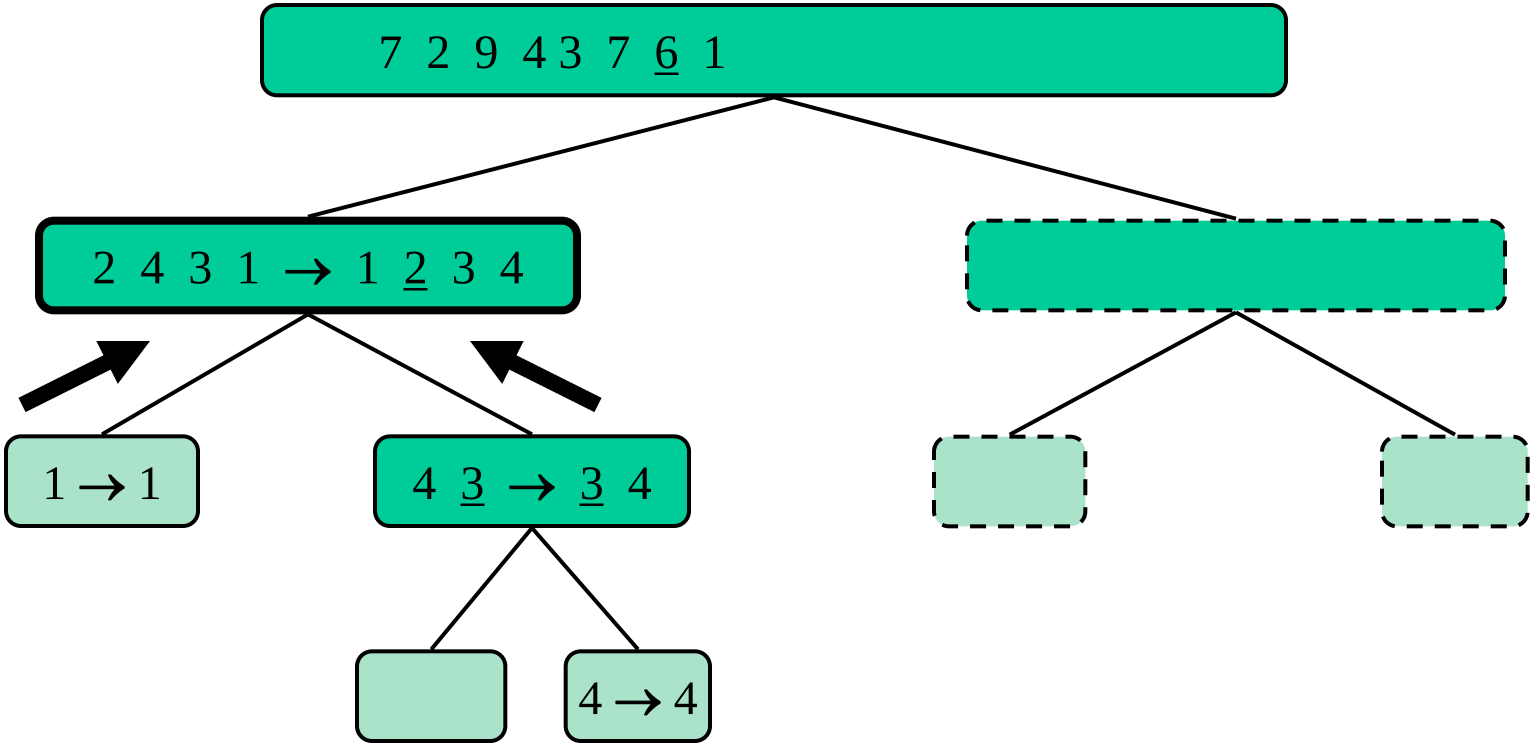
# Execution Example (cont.)

- Partition, recursive call, base case



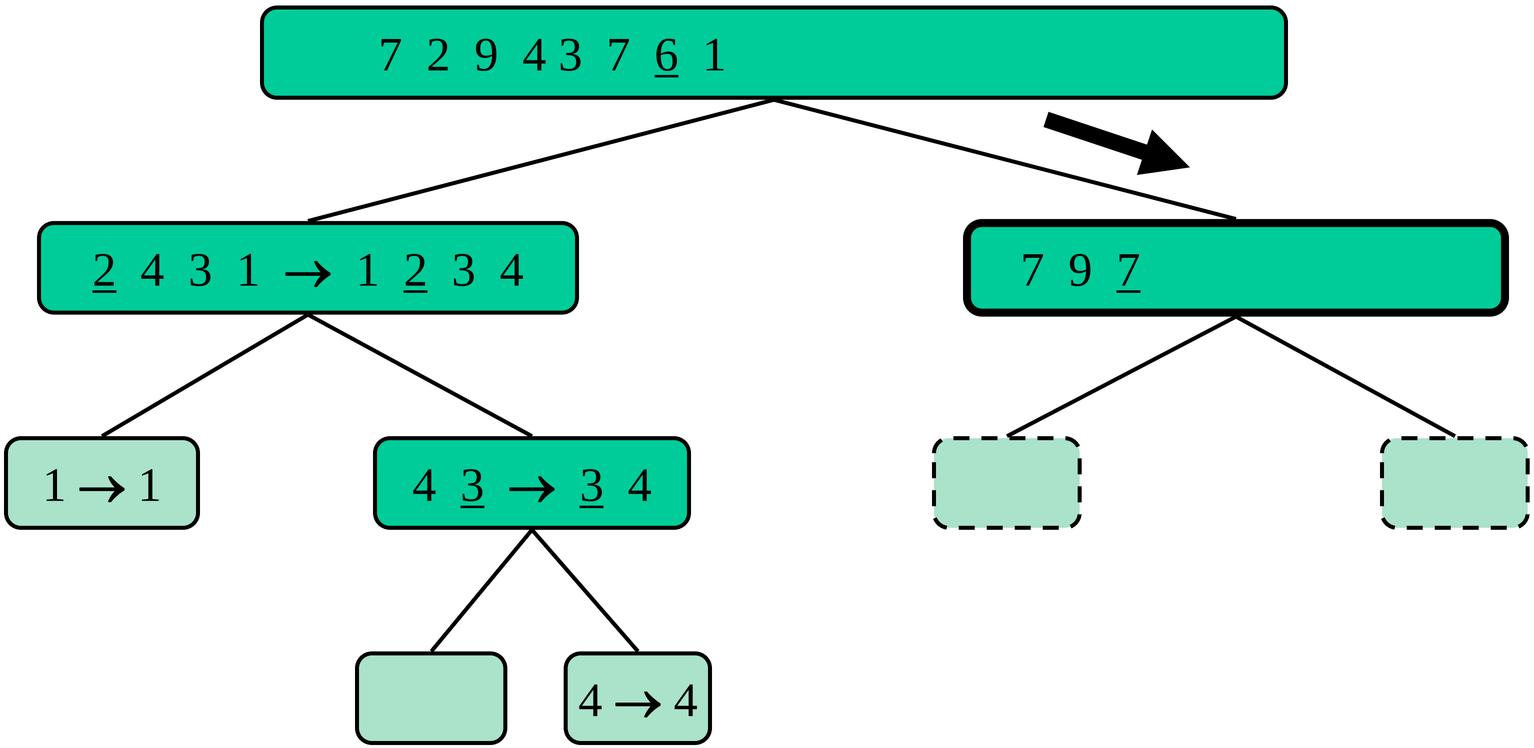
# Execution Example (cont.)

- Recursive call, …, base case, join



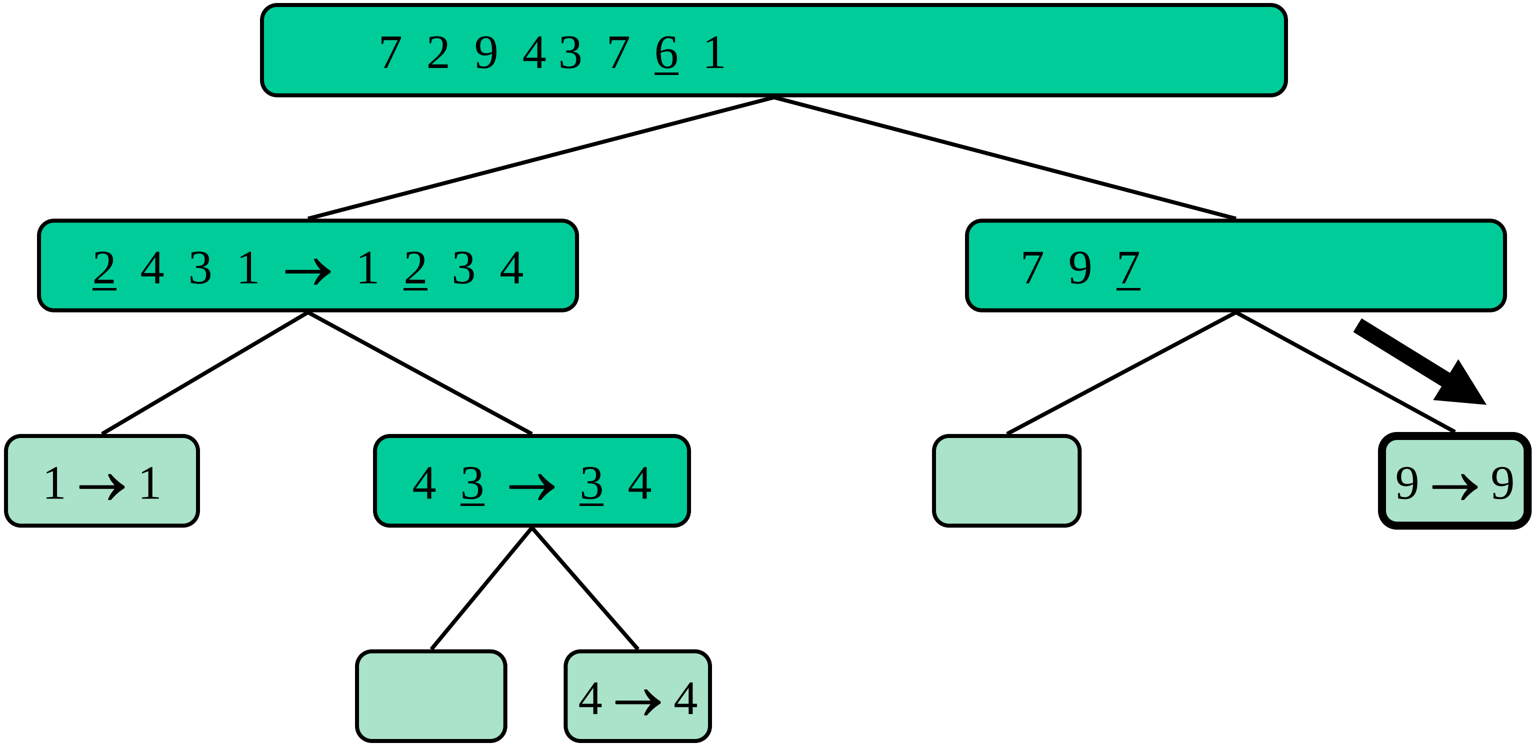
# Execution Example (cont.)

- Recursive call, pivot selection



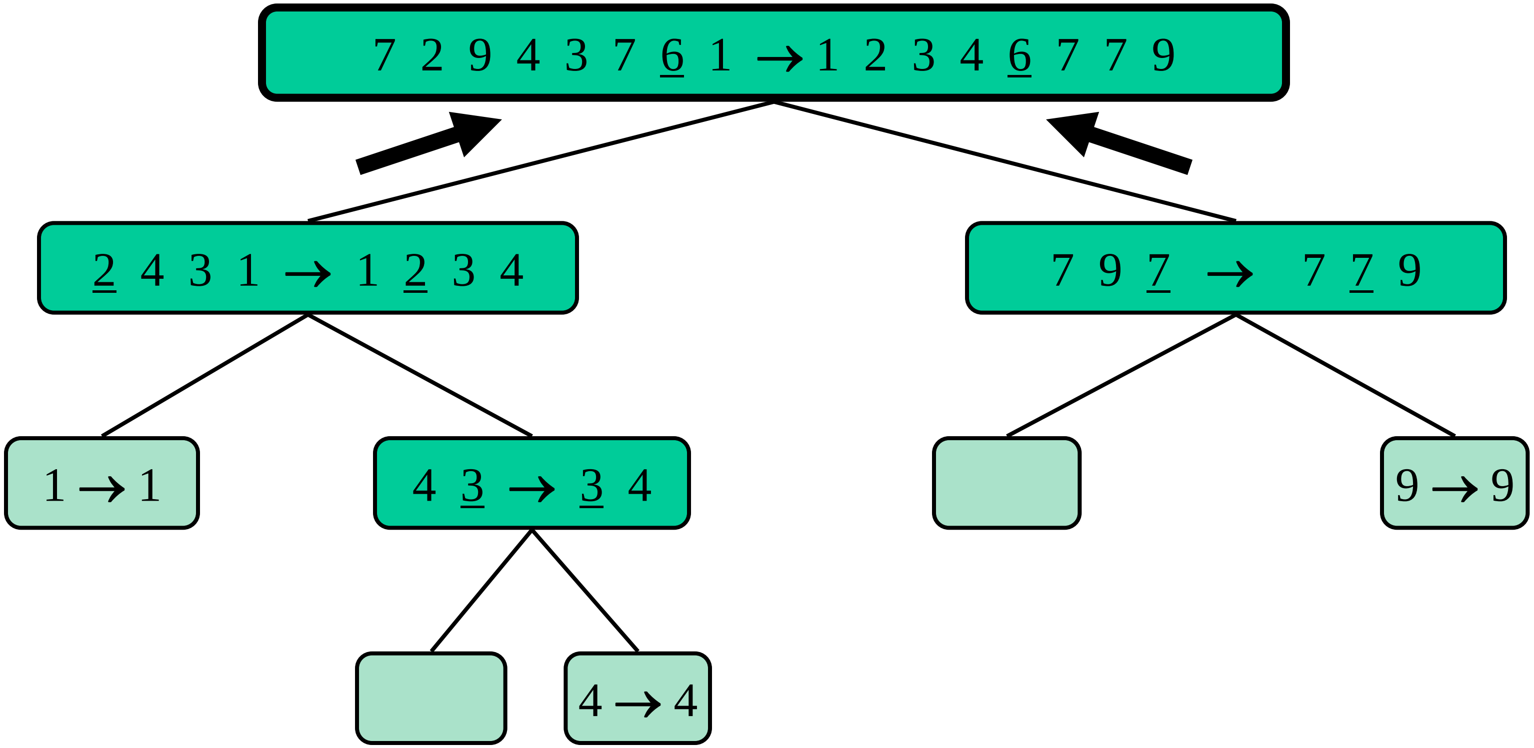
# Execution Example (cont.)

- Partition, …, recursive call, base case



# Execution Example (cont.)

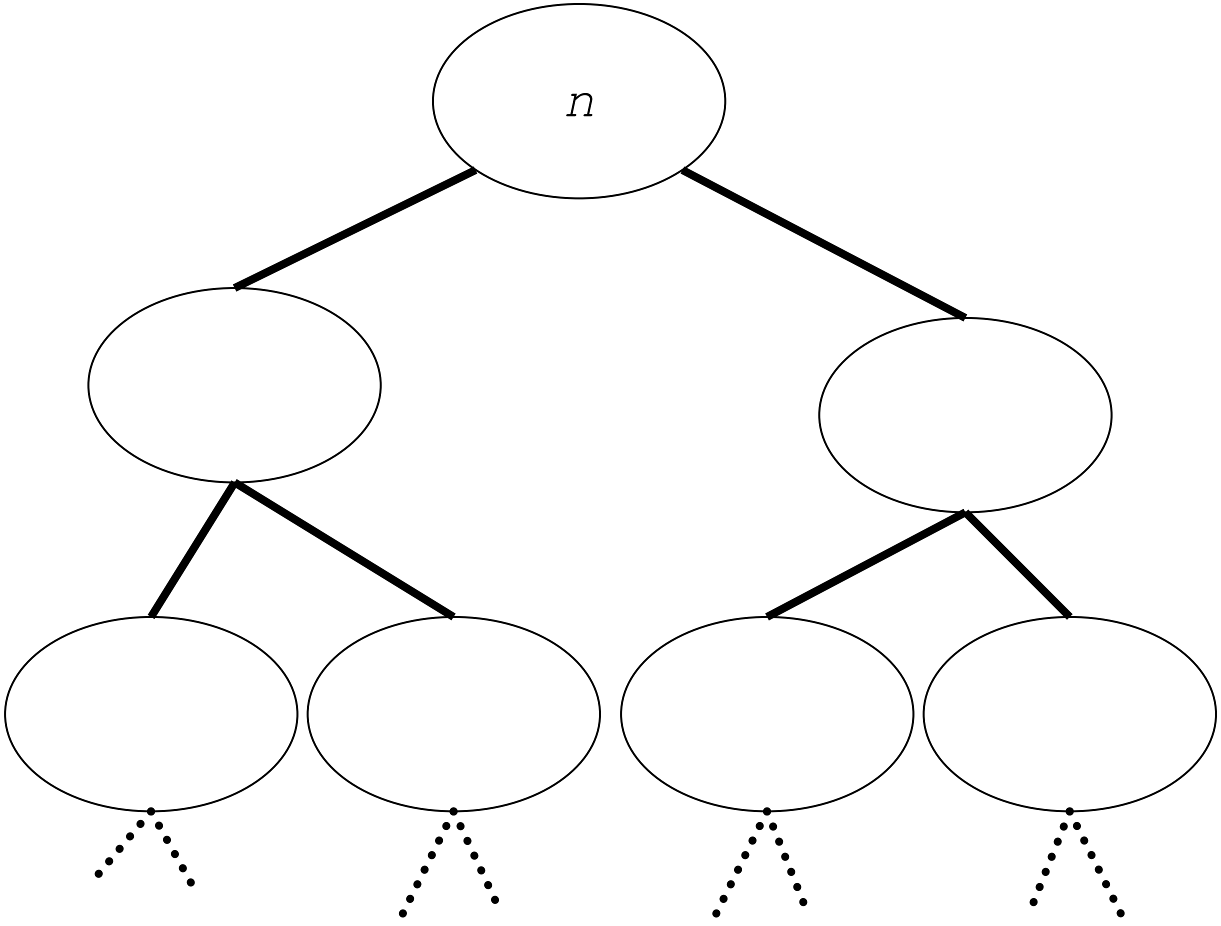
- Join, join



# Quicksort: running time analysis

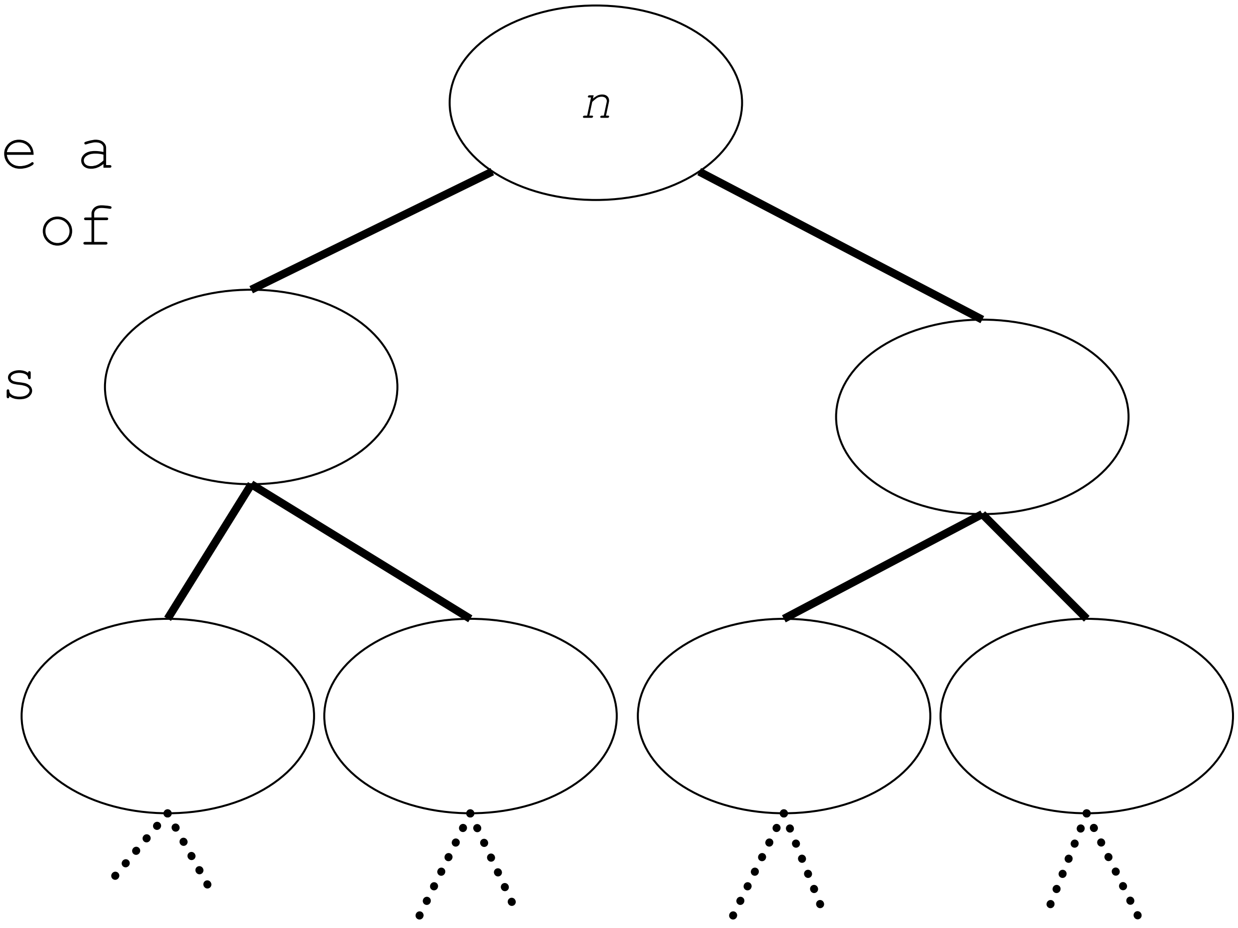
- How long can a branch in the Quicksort tree be?
- What is the worst-case running time of Quicksort?
- What sequences require the worst-case running time?
- What is the best-case running time?
- Why is Quicksort called *quick* sort?

# How long can a branch in the Quicksort tree be in the worst case?



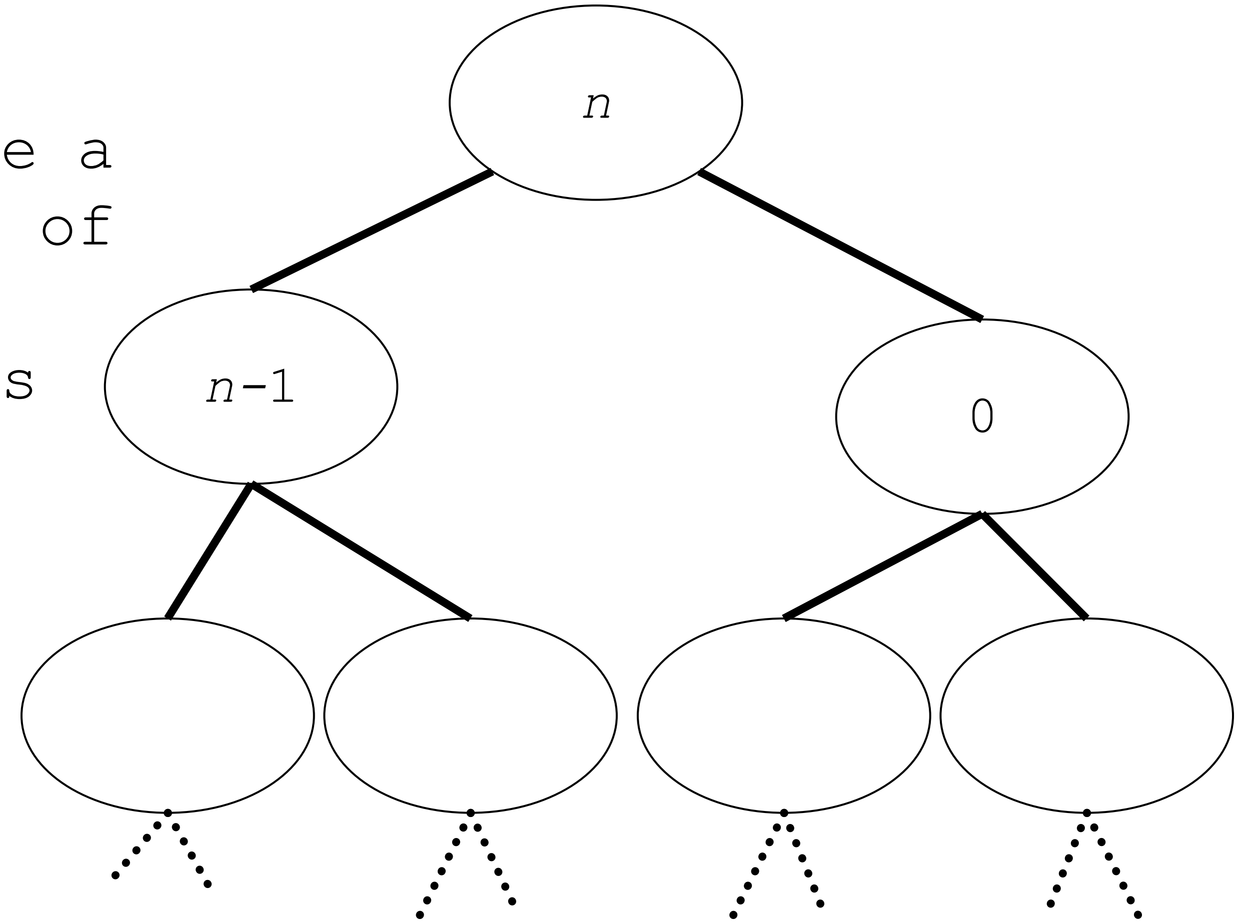
# The pivot $x$ and the length of sequences $L$ and $G$

Let $x$ be a largest of all elements



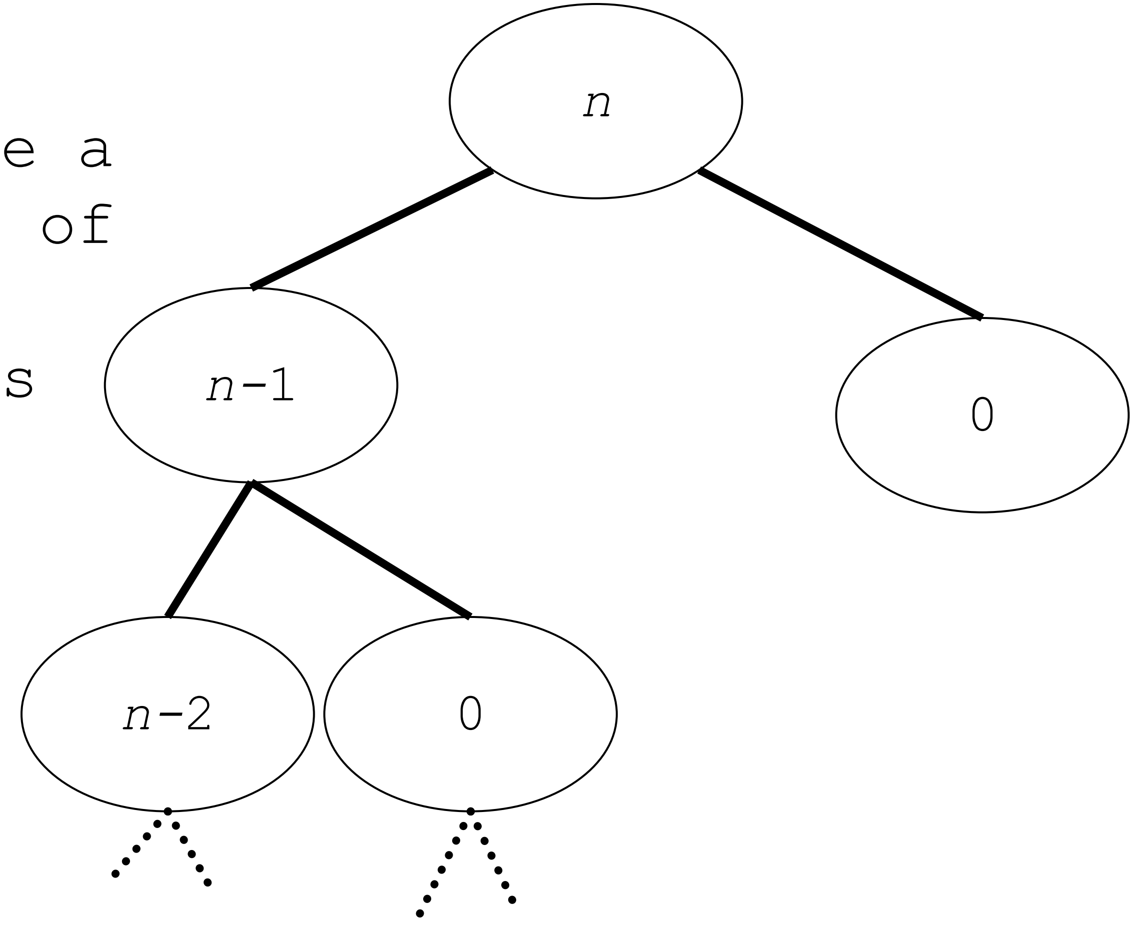
# The pivot element and the length of sequences *L* and *G*

Let *x* be a largest of all elements



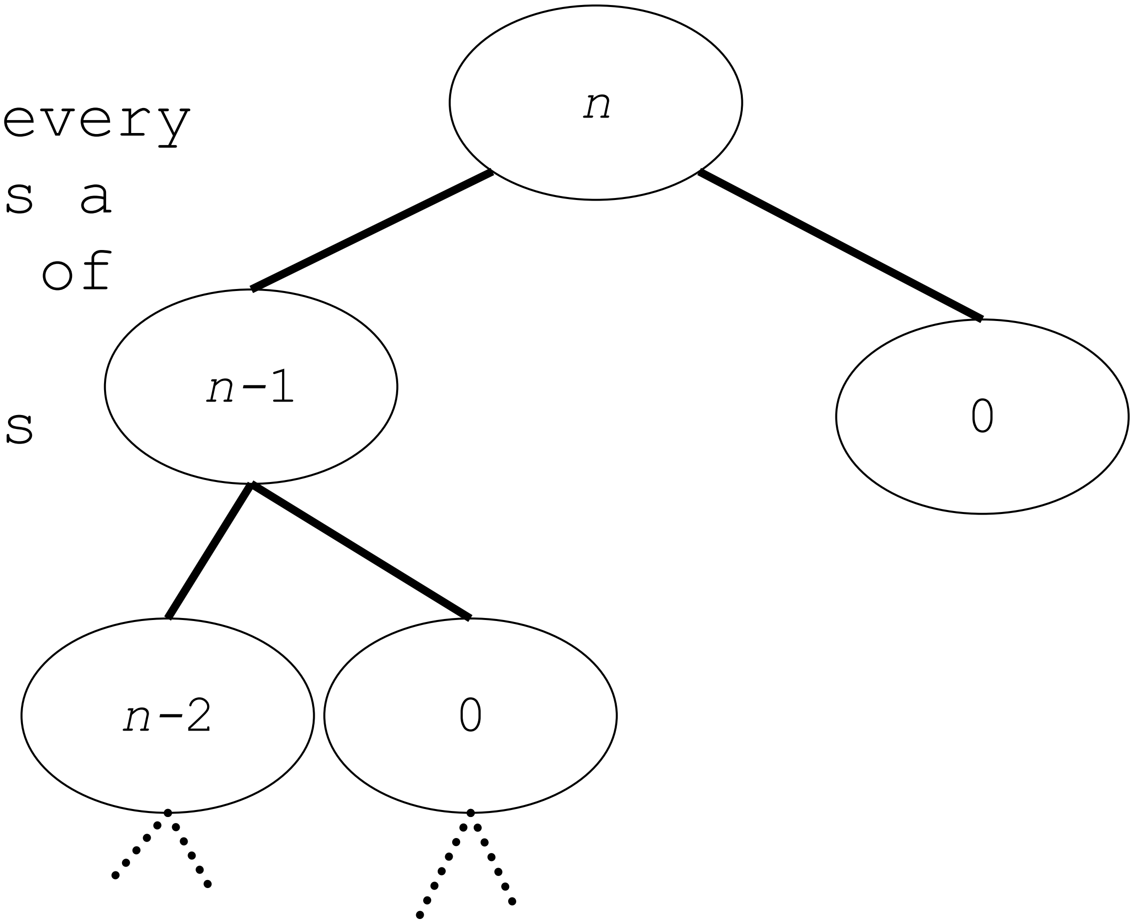
# The pivot element and the length of sequences *L* and *G*

Let $x$ be a largest of all elements

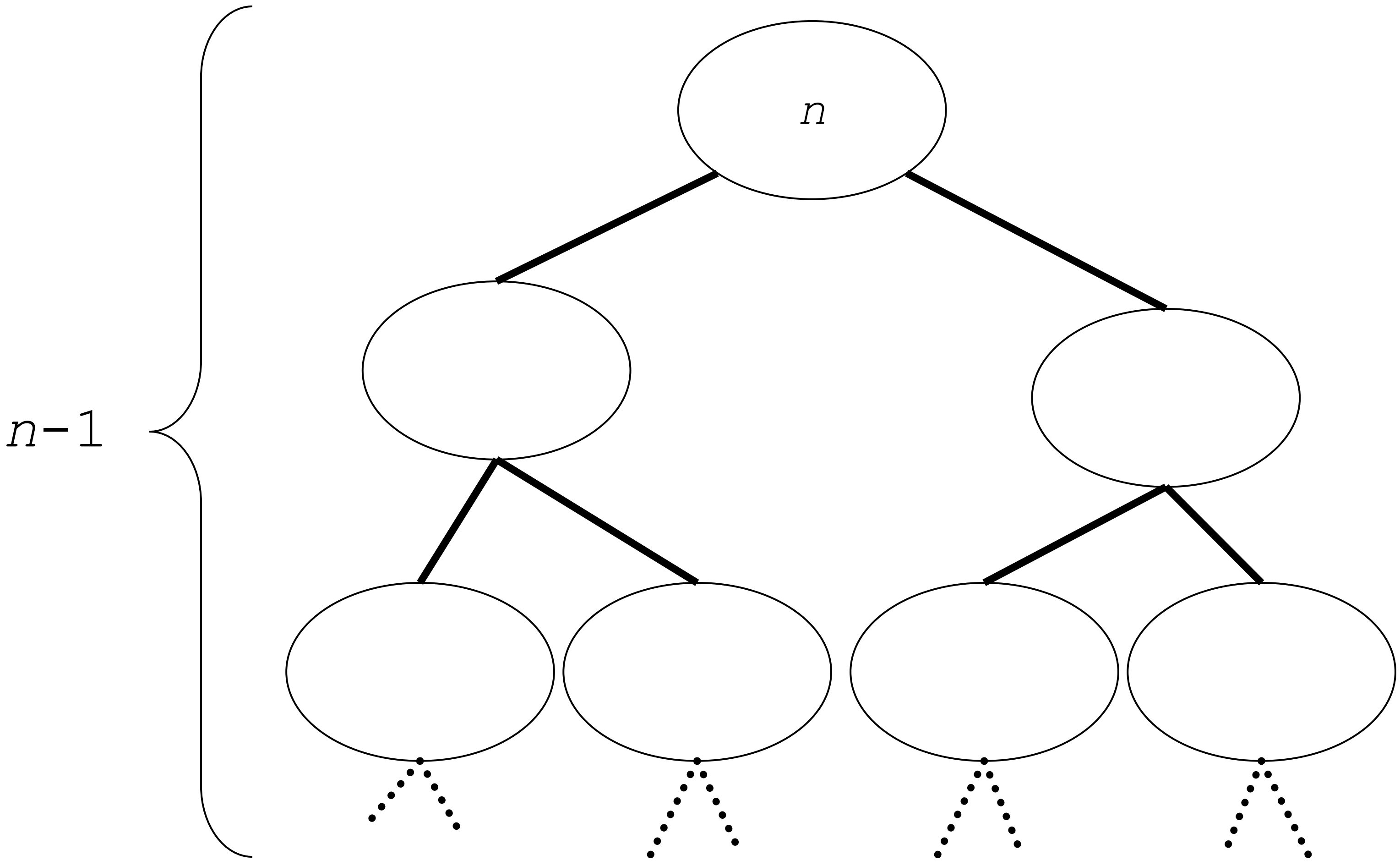


# What sequences require the longest branch?

Assume every pivot is a largest of all the elements to sort



# How long can be a branch in the quick-sort tree?



# What sequences require the worst-case running time?

- Sorted sequences

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

# What is the worst-case running time of Quicksort?

Create $L$, $G$ and $E$ in each level of the "tree".

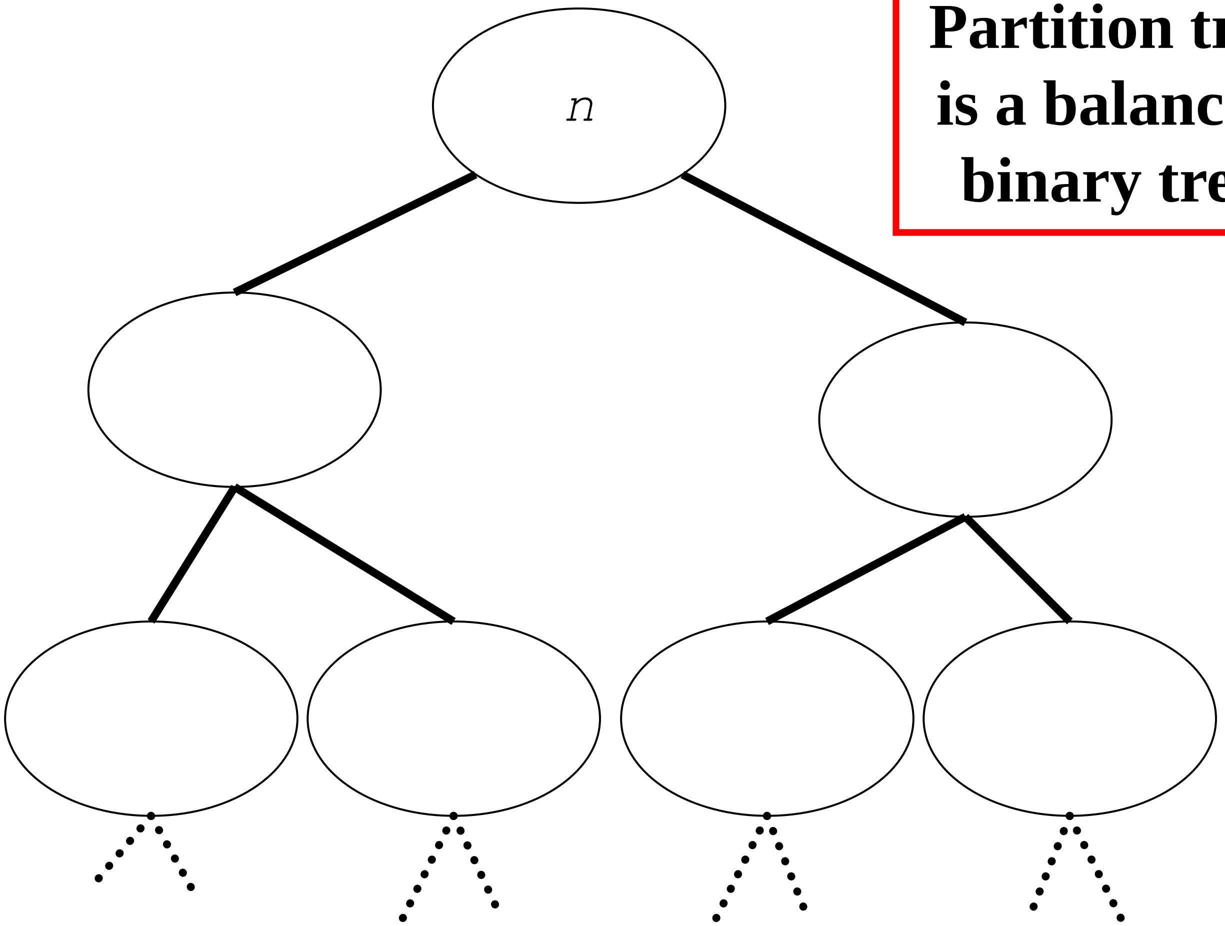
$$\sum_{i=1}^{n-1} i \text{ is } O(n^2)$$

Concatenate $L$, $G$ and $E$ in each level of the "tree".

$$\sum_{i=1}^{n} i \text{ is } O(n^2)$$

O($n^2$)

# When is Quicksort fastest?




**Partition tree is a balanced binary tree**

# A best case running time for Quicksort

$$O(n \log n)$$

# Randomized Quicksort

- Even though the worst case running time of Quicksort is quadratic, in practice Quicksort is a very efficient sorting algorithm.

- Consider the expected running time of "Randomized Quicksort" where the index of the pivot is chosen randomly.

# Randomized Quicksort

- ***Theorem*.** The **expected** running time of **randomized** Quicksort on a sequence of size $n$ is $O(n \log n)$.

- Randomized means choosing a pivot randomly from the set of elements to be sorted.

- How can we prove this theorem?
- To obtain $O(n \log n)$ **expected** time, we need to split up at least a fraction of $n$ of all the elements. Why that is the case we show a little later in the course.
- Suppose we can show that we can split up a ¼ $n$ elements not every time, but every other time we choose a pivot randomly, then we are done.

# Random Pivot Selection

- Suppose our set of elements is sorted


- A "good" pivot is one that is in the red range
- How can we choose a pivot from the red range when the array is not sorted yet?
- Let us select a pivot randomly from the input set
- What are the chances that the pivot is in the red range?
  - 50 %
  - Probability ½
  - Basic coin toss
- Thus, every other time we choose a "good pivot" if we choose one randomly

# Proof

- Now we have to estimate the height of the recursion tree, given that we we split up at least ¼ elements every other time.
- Suppose that we split up ¼ elements every time

$$\frac{1}{4}|S| \le |L| \le \frac{3}{4}|S| \qquad \frac{1}{4}|S| \le |G| \le \frac{3}{4}|S|$$

- Then the Quicksort recursion-tree is bounded in height by $\log_{4/3} n$
- Since we only choose a good pivot every other time the Quicksort recursion-tree is bounded in height by $2\log_{4/3} n$

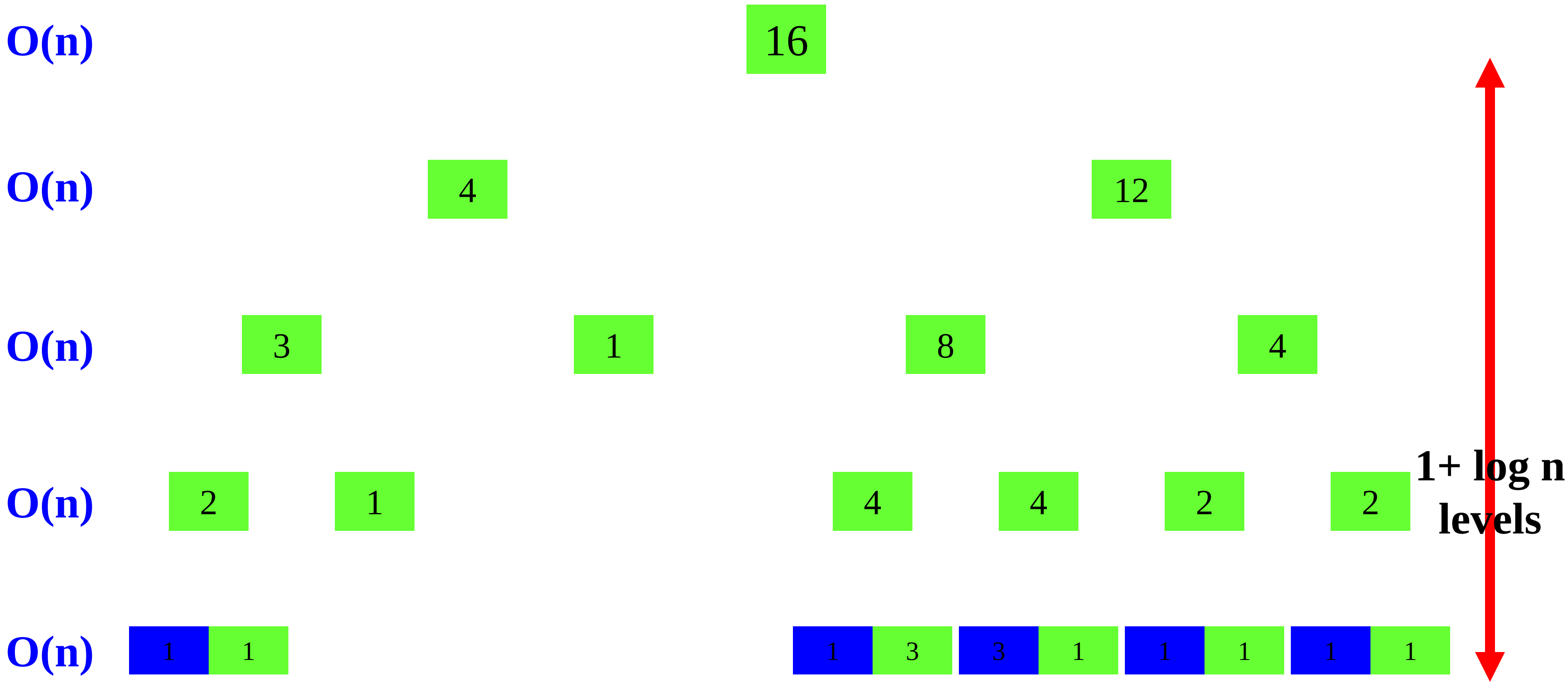
# Proof

- How many pivots do you have to pick to get $\log_{4/3} n$ good ones? $2\log_{4/3} n$
- What is the probability to pick a good pivot? $\frac{1}{2}$
- How many good pivots exist? $\frac{1}{2}n$

# Proof

- Thus the tree has an expected bound in height of $2\log_{4/3} n$
- **Thus, the resulting expected running time for Randomized Quicksort is *O*(*n* log *n*)**

# Height of Recursion Tree




$$T(n) = n\log_{4/3} n = n\frac{\log_2 n}{\log_2 4/3} = cn\log_2 n \in O(n\log n)$$