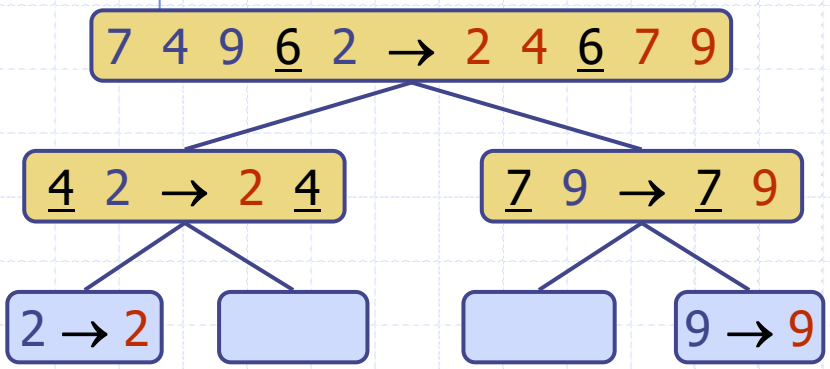
Lesson 5
QuickSort

Enlivening Hidden Laws of Nature to Manage Change

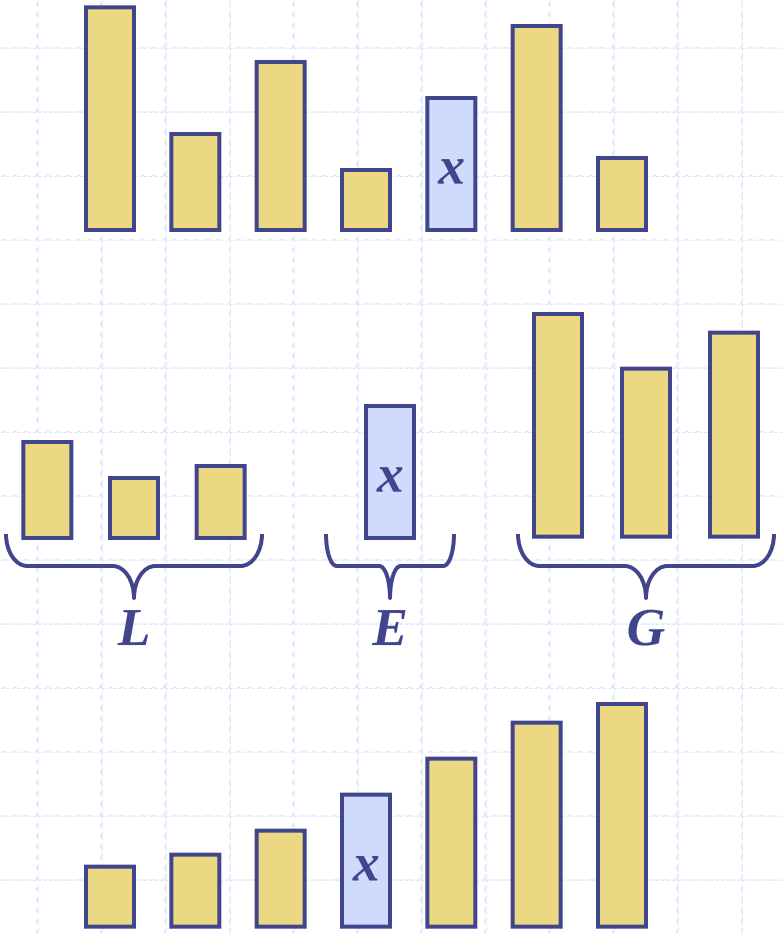


Wholeness of the Lesson

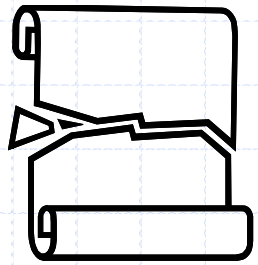
Quick Sort, another Divide and Conquer sorting algorithm, typically sorts lists even faster than Merge Sort. QuickSort achieves even greater sorting efficiency by replacing the crucial merge step of MergeSort, which requires repeated access to temporary storage, with a subtler pre-processing partition step, which eliminates the need for temporary storage. This technique illustrates the principle that subtler levels of the mind and of the universe are more powerful; when these can be harnessed, more can be accomplished.

Quick-Sort

- Quick-sort is a sorting algorithm based on the divide-and-conquer paradigm. Consider first a randomized version:
  - Divide: (Partition) 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
  - Conquer: Sort L and G
  - Combine: Join L, E and G



Partition



- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence. For a structure like a linked list, this takes $O(1)$ time, and so partition step takes $O(n)$ time.
- With a slight variation, the partition step for arrays is also $O(n)$.

Algorithm *partition*(S, p)

Input sequence S, pivot p

Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.

$L, E, G \leftarrow$ empty sequences

while !*S.isEmpty*()

$y \leftarrow$ *S.removeFirst()*

if $y < p$

L.insertLast(y)

else if $y = p$

E.insertLast(y)

else { $y > p$ }

G.insertLast(y)

return L, E, G

QuickSort in Pseudo-Code

Algorithm ***quickSort***(***S***)

Input sequence ***S***

Output S in sorted order

if(|S|=0 or |S|=1) then **return** S

$p \leftarrow$ ***pickPivot()***

(***L***,***E***,***G***) $\leftarrow$ ***partition***(***S***,***p***)

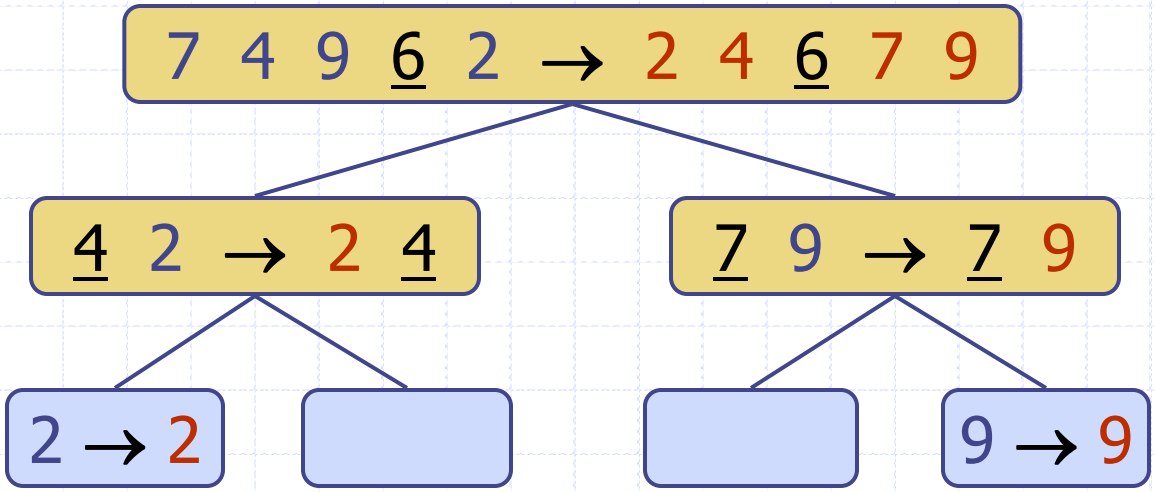
quickSort(***L***)

quickSort(***G***)

return $L \cup E \cup G$

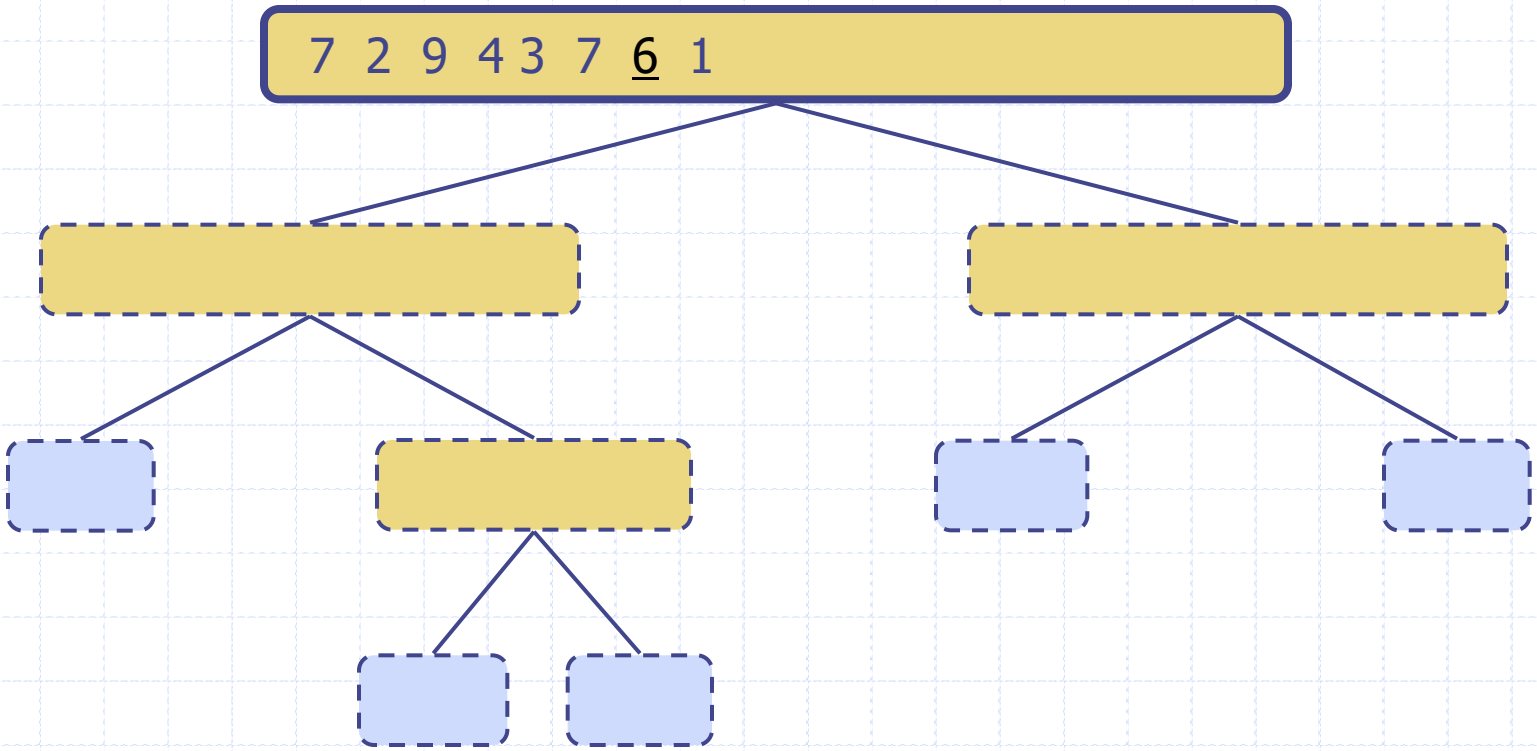
Quick-Sort Tree

- We can represent execution of quick-sort using a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution
    - 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



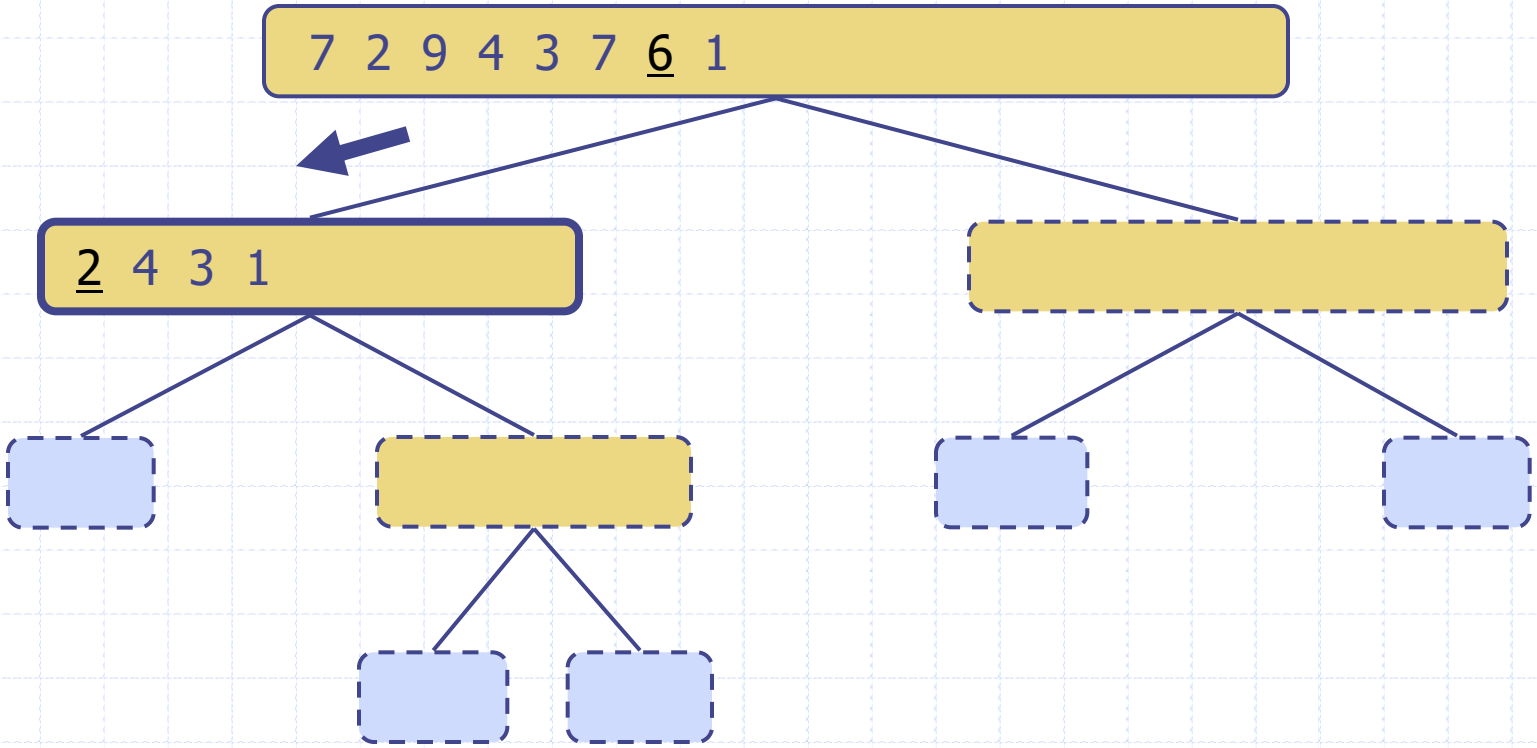
Execution Example

- Pivot selection



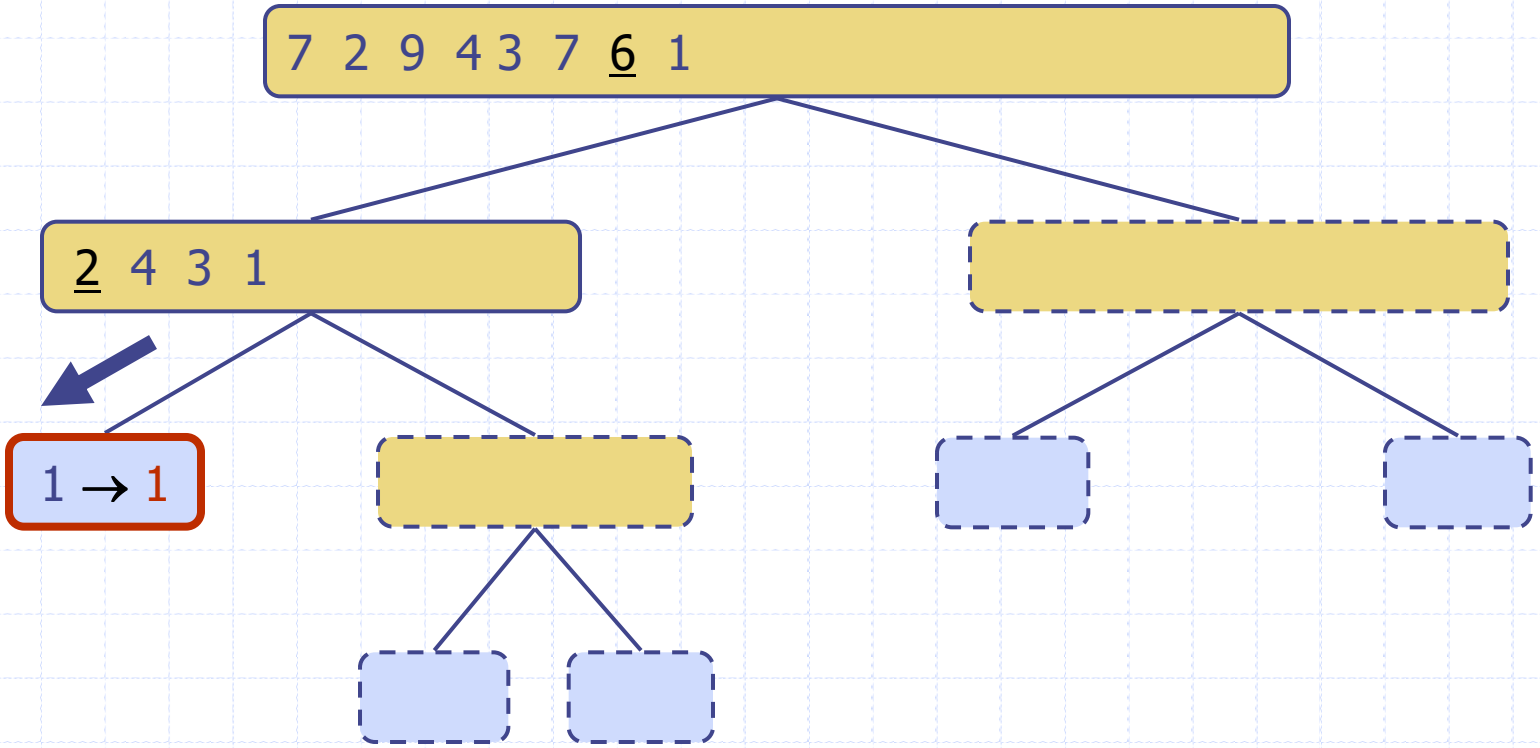
Execution Example (cont.)

- Partition, recursive call, pivot selection



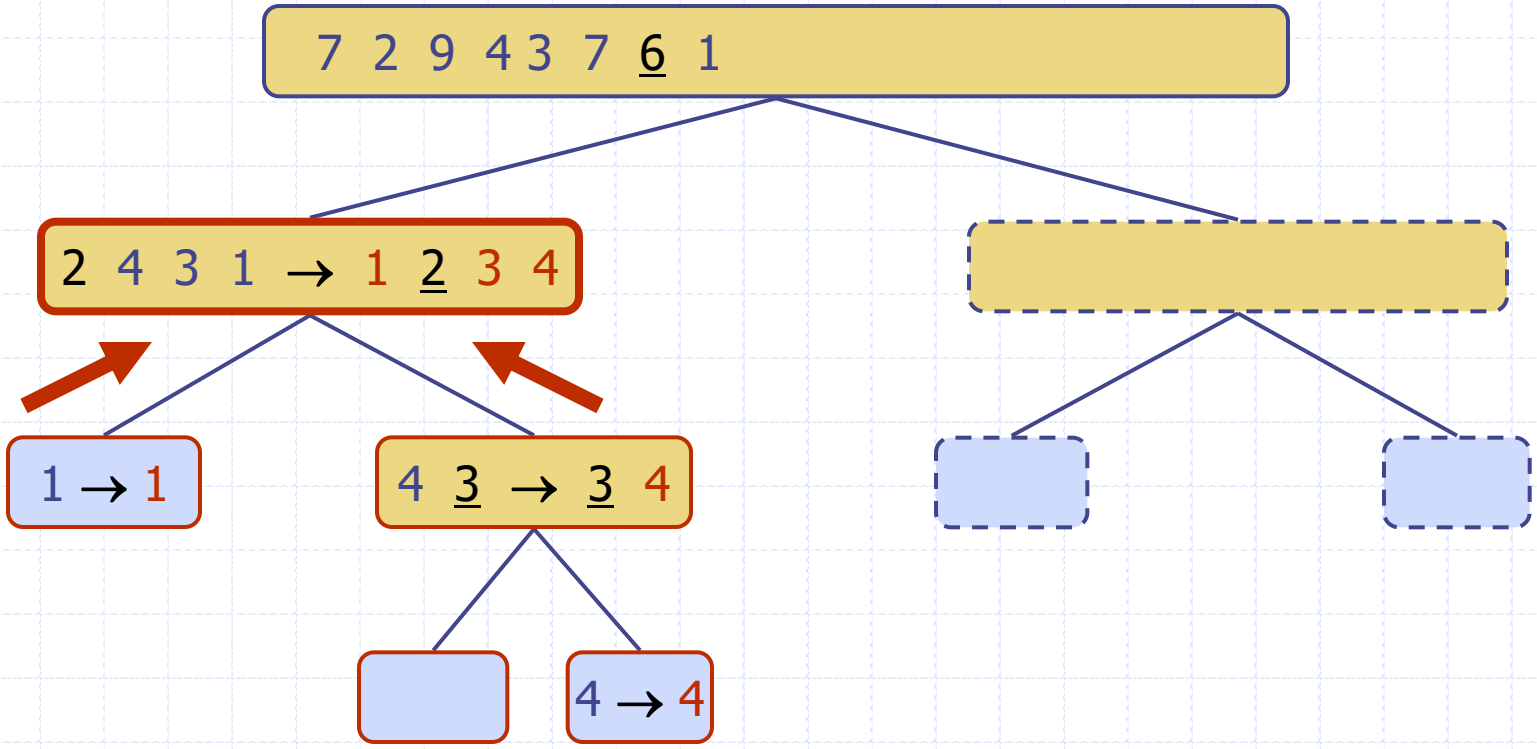
Execution Example (cont.)

- Partition, recursive call, base case



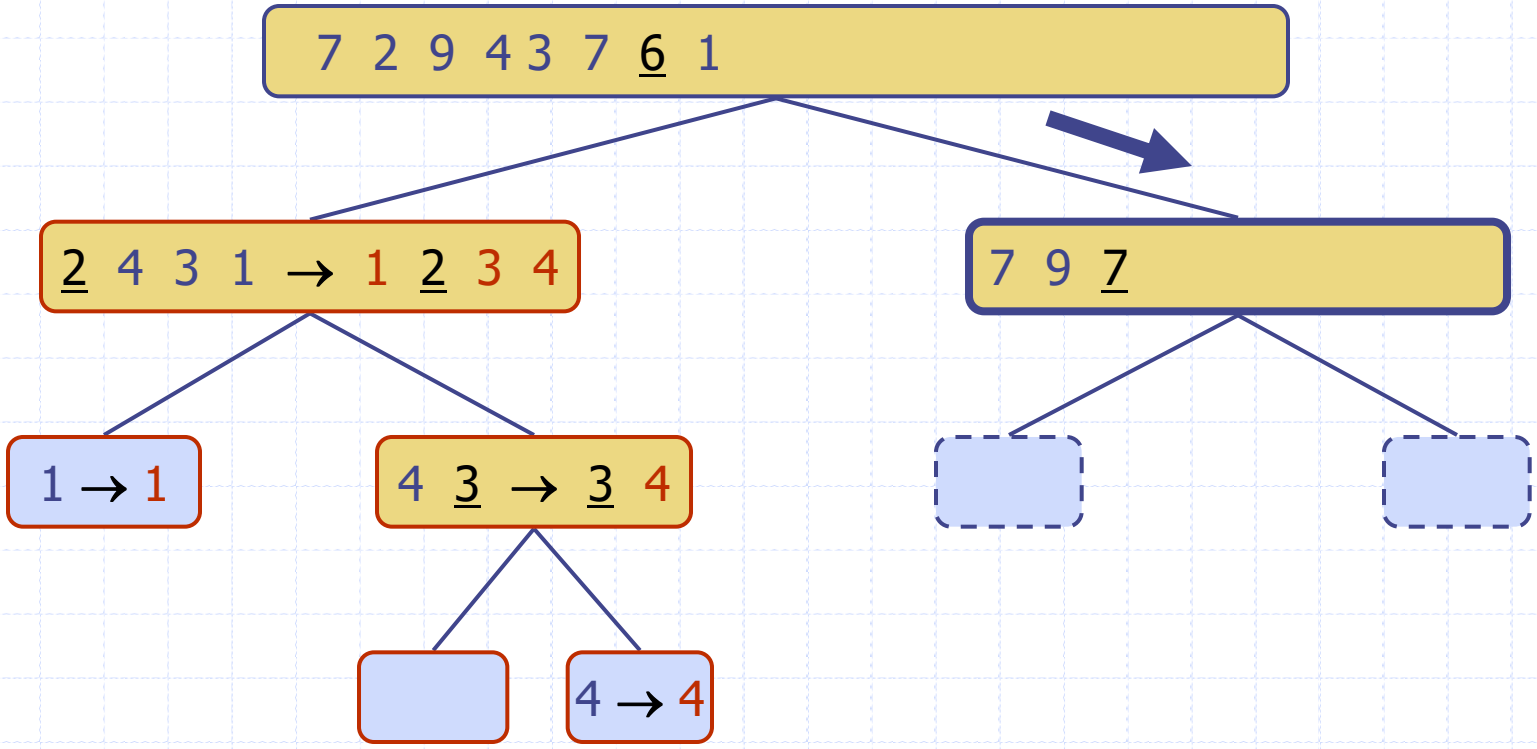
Execution Example (cont.)

- Recursive call, …, base case, join



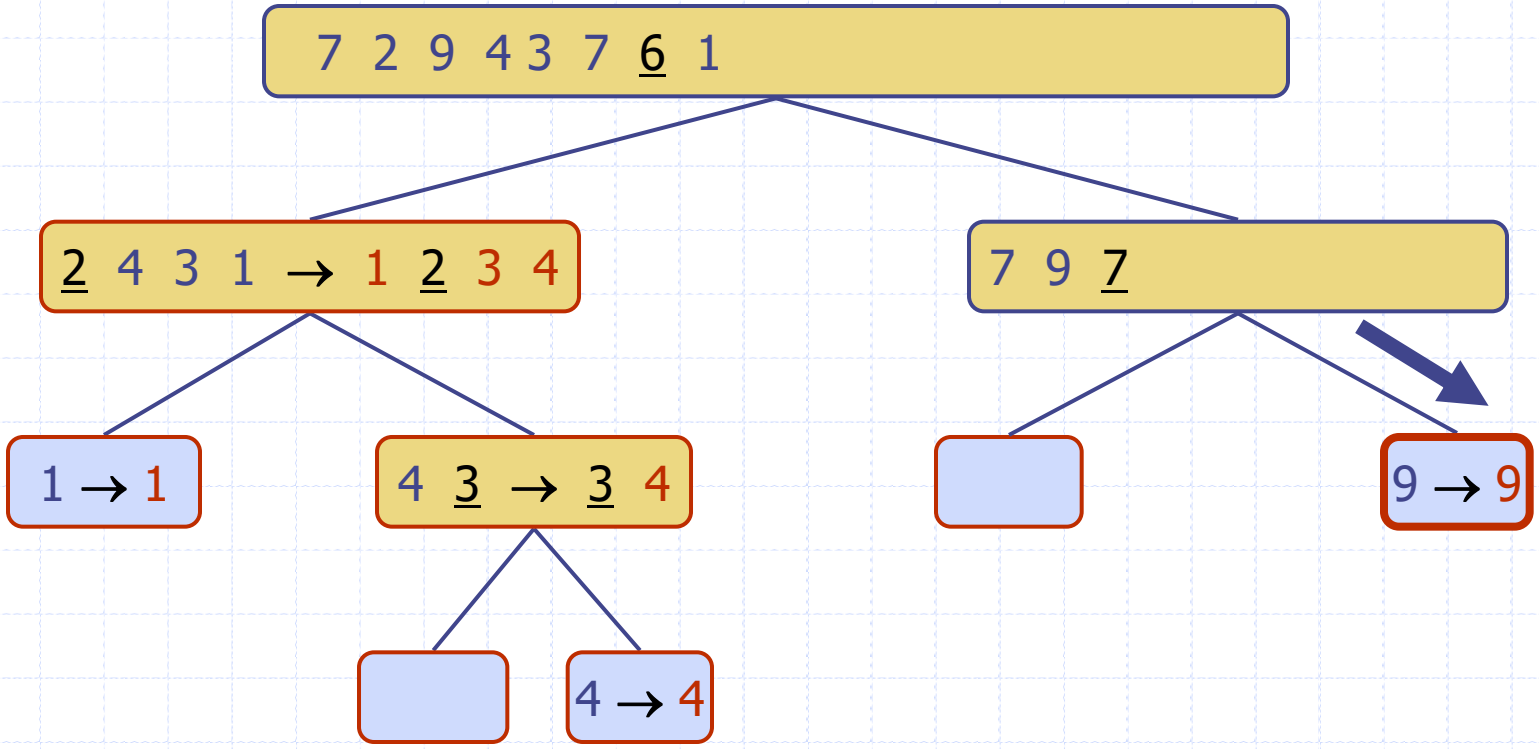
Execution Example (cont.)

- Recursive call, pivot selection



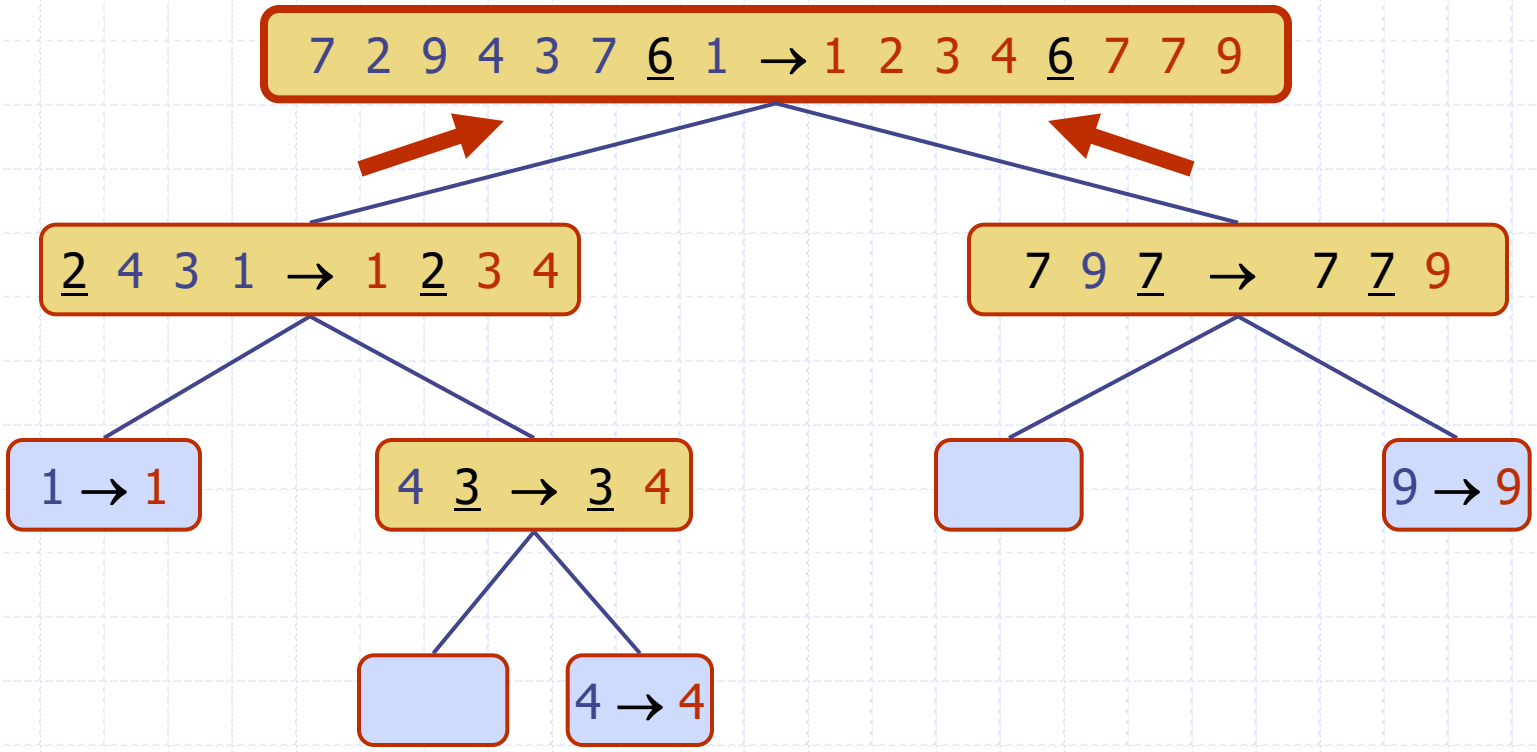
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join

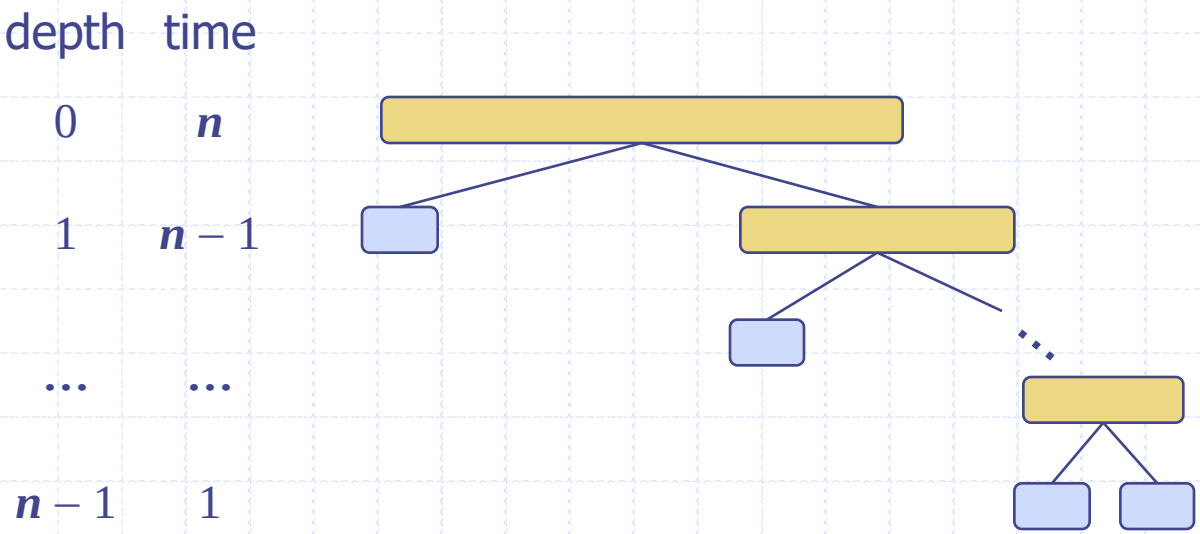


Worst-case Running Time

- The worst case for quick-sort occurs when the pivot selected is always the unique minimum (or maximum) element
- In that case, one of L and G has size $n-1$ and the other has size 0
- The running time is proportional to the sum

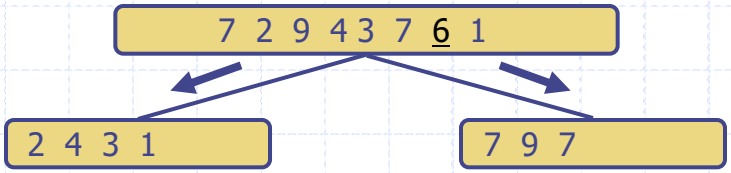
$$n + (n-1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $\Theta(n^2)$

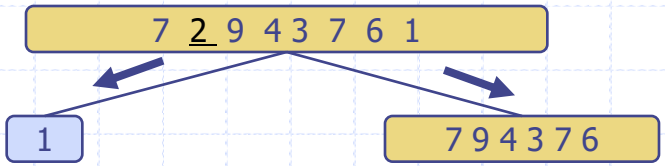


Expected Running Time

- Consider a recursive call of quick-sort on an array of size n
  - **Good self-call:** the sizes of L and G are each less than $3n/4$ (normal division)
  - **Bad self-call:** one of L and G has size greater than or equal to $3n/4$



Good call



Bad call

- A self-call is good with probability at least 1/2 [NOTE: alg is randomized]
  - At least 1/2 of the possible pivots cause good self-calls :

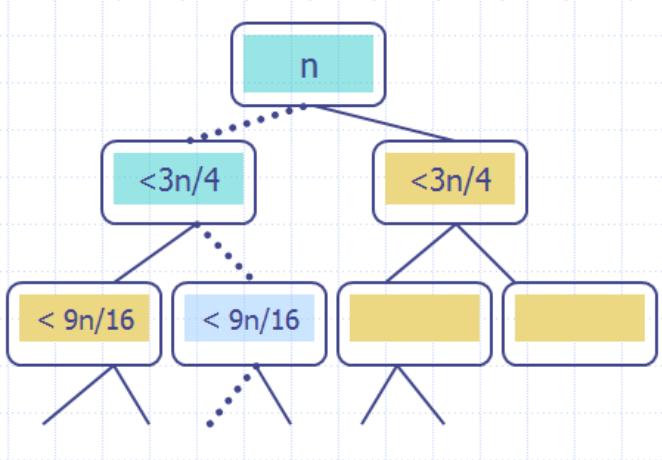
The "Good Case" Analysis

- A good case occurs if every self-call is good
- The height of recursion tree is one less than the number of terms of the descending sequence

  $n, (3/4)n, (3/4)^2n, \ldots, 1, 0.$

By exercise, there are $2 + \lfloor \log_{4/3} n \rfloor$ terms, so the height of the recursion tree is $1 + \lfloor \log_{4/3} n \rfloor$, which is O(log n)

- At each level of the recursion tree, total processing time is O(n)
- Therefore, total running time in the good case is O(nlog n)



General Average Case Analysis

- Good self-calls don't occur every time – but we know probability of a good self-call is ½.
- If all self-calls are good, height of tree is $m = 1+\lfloor \log_{4/3} n \rfloor$. This implies that in general, recursion ends after m good self calls.
- The expected height of average case recursion tree is the expected number of self-calls required to obtain m good self-calls.
- Question: "What is expected number of self-calls to get m good self-calls?" We are going to answer this using a simple coin-flipping experiment from Probability.

The coin-flipping experiment

- Repeatedly flip a coin. How many flips are required to get "heads"?

  In the worst case, what is the answer?

  Answer: In the worst case, "heads" never comes up.

- We can see that in this case, worst case analysis is not useful. Our next question will be: What is the expected number of flips required to get heads?

  Answer: The expected number of flips required to get "heads" is 2.

The coin-flipping experiment

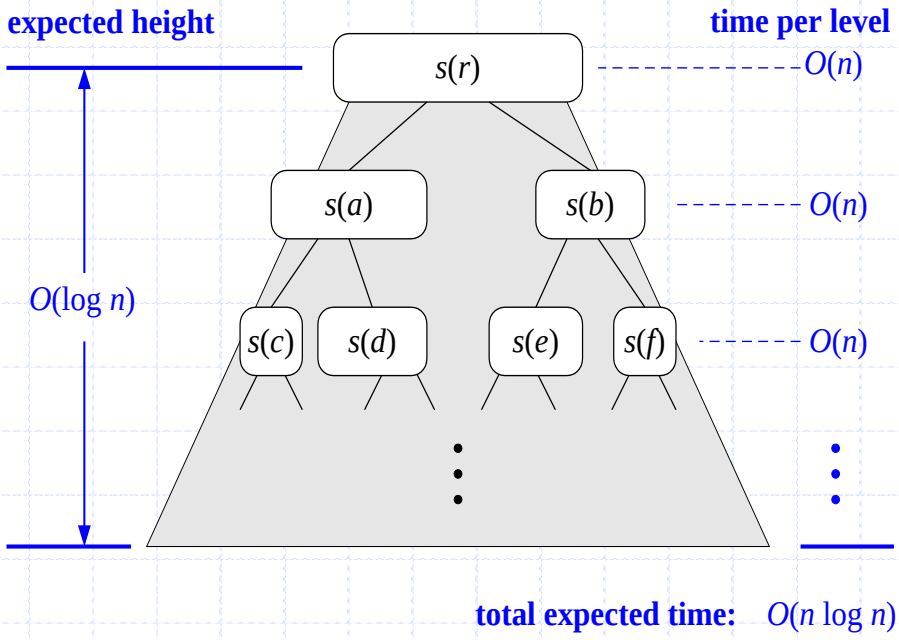
- Theorem [Expected number of trials for success]. Suppose an experiment is performed repeatedly, and the probability of "success" is p (where p is a real number between 0 and 1), and is not influenced by previous successes or failures. Then the expected number of trials to obtain a success is 1/p.
  - For example, the expect number of die rolls required to get a 1 is 6. (since the probability to get a 1 is 1/6.)
  - The expected number of flips required to get "heads" is 2 (since the probability to get heads is 1/2.)
  - Likewise, for any integer k>0, The expected number of flips required to get exactly k "heads" is 2k.

This leads to the following Theorem:

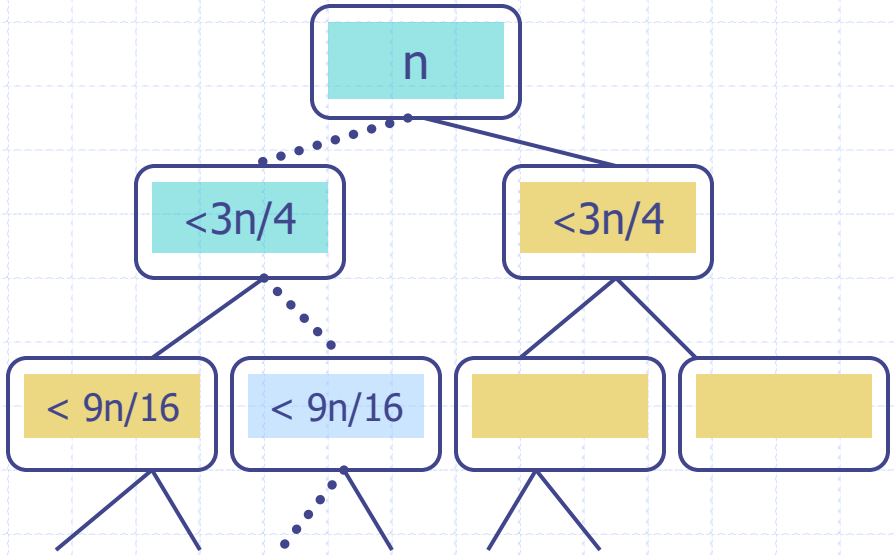
- Theorem [Expected number of trials for k successes]. Suppose k is a positive integer. Suppose an experiment is performed repeatedly, and the probability of "success" is p (where p is a real number between 0 and 1), and is not influenced by previous successes or failures. Then the expected number of trials to obtain k successes is k/p.

General Average Case Analysis

- "What is expected number of self-calls to get m good self-calls?" is like "What is expected number of flips to get m heads?" In both cases, answer is 2m. In our case, m is O(log n), so 2m is also O(log n).
- Therefore, in recursion tree, since there are O(n) processing steps at each level, and since expected height is O(log n), we conclude that average case running time is O(nlog n).



Overview Of Avg Case Analysis



1. **Any branch of self-calls in which each successive self-call cuts input size by a factor of <¾ each time must have height $\log_b n$ where b = 4/3. Call a self-call that cuts input size by a factor of <¾ a "good self-call"**
2. **Think of "success" as "a good self-call occurs". Probability of "success" is at least ½**
3. **Let $m = 1+\log_b n$. On any path from root to leaf, expected number of self-calls to get m good self-calls is like expected number of coin flips to get m heads: E = 2m which is O(log n). Therefore, expected height of each branch is O(log n), so expected height of recursion tree is O(log n)**
4. **Running time at each level is O(n) => average case running time is O(nlog n).**

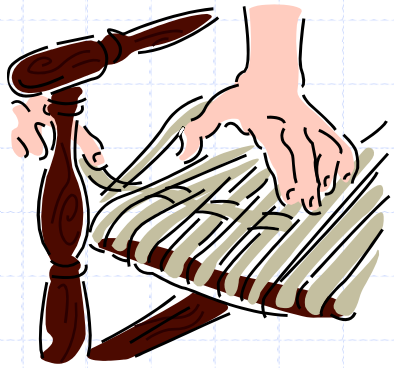
Likelihood of Worst Case

- **Using the probabilistic technique of "Chernoff bounds", one can show that prob of quickSort having its worst case running time on input array of size n is less than $1/n^2$.**
- **In another words, the average-case O(nlog n) running time of QuickSort is extremely likely to occur.**

Main Point

In average case analysis of QuickSort, it is observed that the actual number of self-calls required to complete sorting does not deviate much from the number of *good* self-calls that occur in the process. In a sense, the *good* self-calls cause the process as a whole to unfold as efficiently as possible, and the fact that they occur so frequently follows from laws governing random behavior. In a similar way, when the home of natural law is enlivened in individual awareness, Maharishi explains that it is "more difficult to make mistakes" – life is spontaneously lived in a way that does not create harm or problems.

An Implementation: In-Place QuickSort



- Quick-sort usually implemented to run in-place
- In the partition step, we use swap operations to rearrange the elements of the input sequence so that the current subarray has elements less than or equal to the pivot on the left side, and elements greater than or equal to pivot on right side.
- This refinement slightly improves the algorithm described earlier, but makes average case analysis a bit more complicated in the case in which there are duplicates

Algorithm *inPlaceQuickSort*(*S*, *l*, *r*)
  Input array *S*, positions *l* and *r*
  Output array *S* with the
    elements from positions *l* to *r*
    rearranged in increasing order
  if *l* > *r*
    return
  k ← a random integer between *l* and *r*
  swap(k,r) //place pivot at the right
  x ← *S.elemAtPos*(*r*) ***//the pivot***
  i ← *inPlacePartition*(*x*) //new pos of piv
  inPlaceQuickSort(*S*, *l*, *i* – 1)
  inPlaceQuickSort(*S*, *i* + 1, *r*)

In-Place Partitioning

1. Move pivot to the far right (pos = r)
2. Begin with pointers i, j with i at pos l and j at pos $r - 1$
3. Move i to the right past all values < pivot
4. Move j to the left past all values > pivot
5. If stuck, then swap the values and repeat 3, 4, allowing i to move one to the right and j to the left when their turns come (equivalently: after swap, immediately increment i and decrement j by 1)
6. Stop as soon as j crosses (moves to the left of) i or i crosses (moves to the right of) j
7. After stopping, swap the pivot and value at position i.

Partition Example

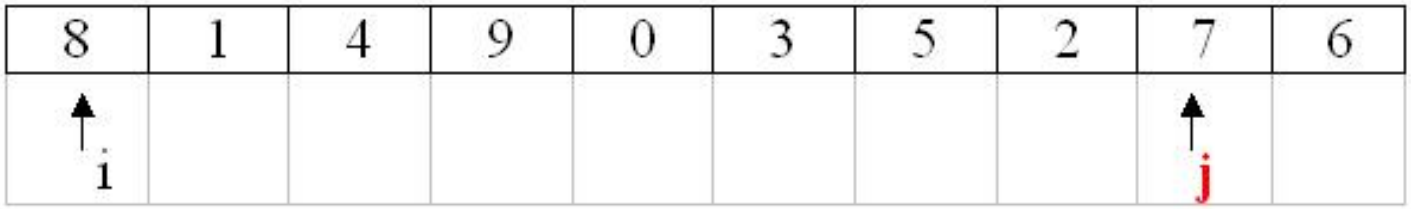
8	1	4	9	6	3	5	2	7	0

Randomly pick pivot -- say it's 6. Swap with rightmost element.

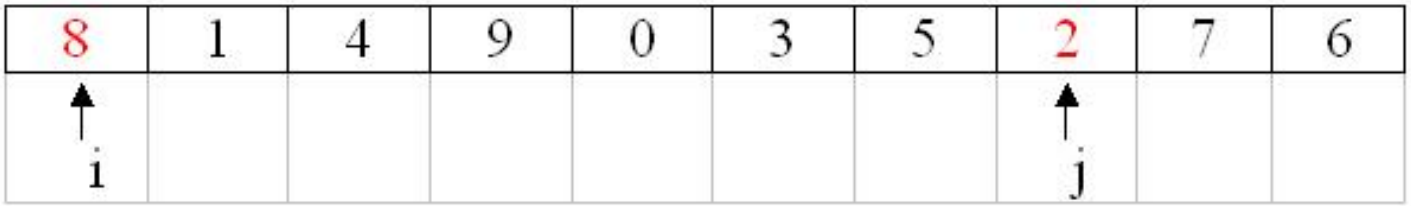
8	1	4	9	0	3	5	2	7	6
i								j	

i now moves right as long as it points to values < 6

Partition Example continued

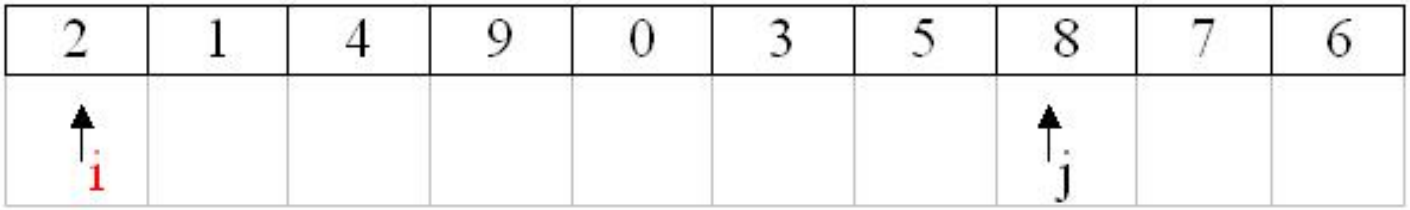


i is stuck at 8. Now j moves to the left as long as it points to a number > 6

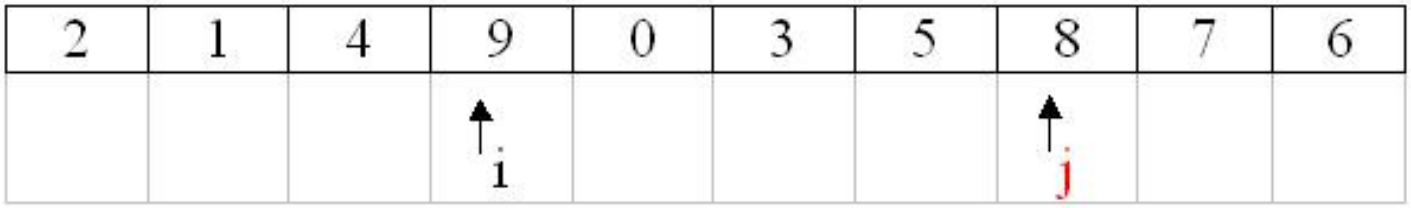


j is now stuck at 2; since i is also stuck, swap 2 and 8. Afterwards, when it is i's turn to move, it automatically is moved one step; likewise for j

Partition Example continued

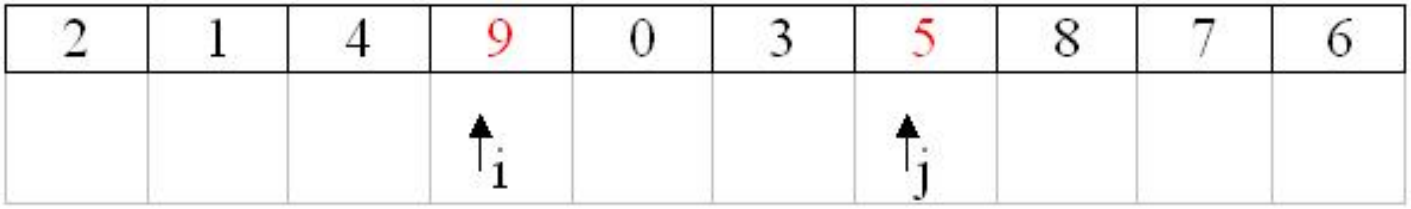


Now i moves to the right as long as value is < 6

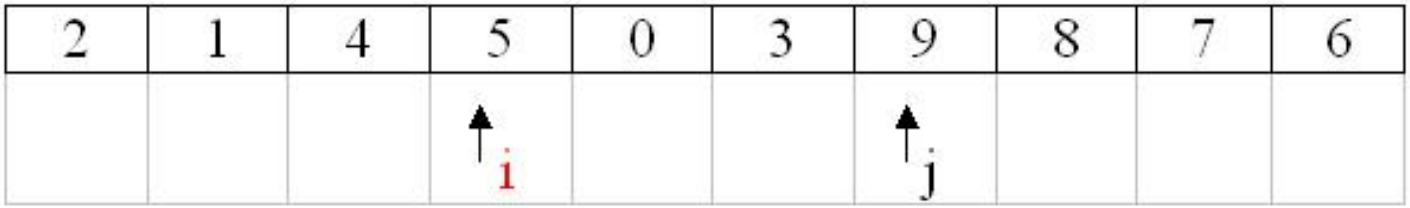


i is now stuck at 9; j moves one to the left for free and continues as long as value is > 6

Partition Example continued

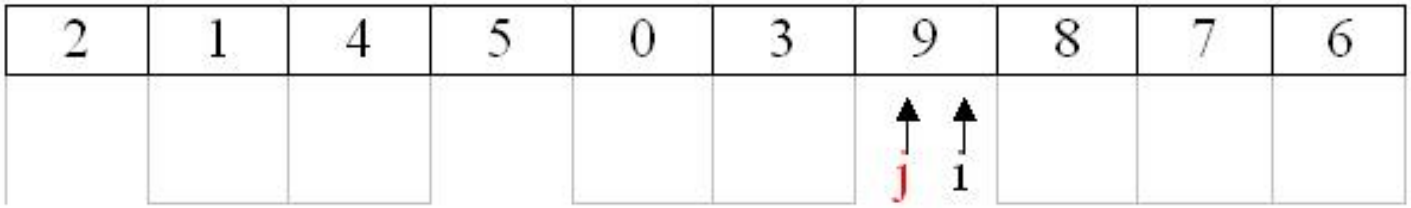


Now i and j are stuck, so swap 5 and 9

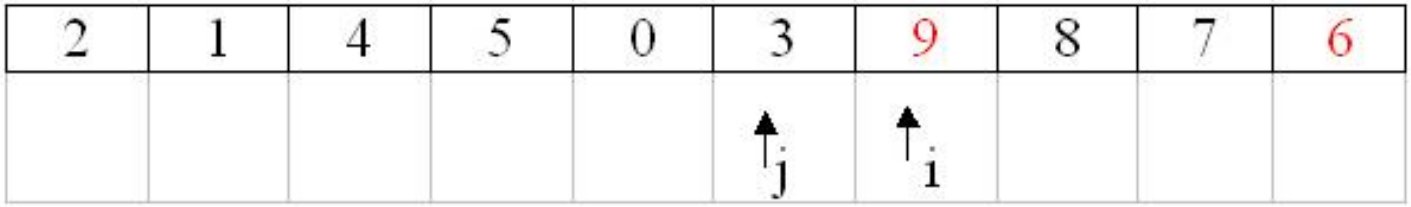


i now moves to the right as long as value < 6

Partition Example continued

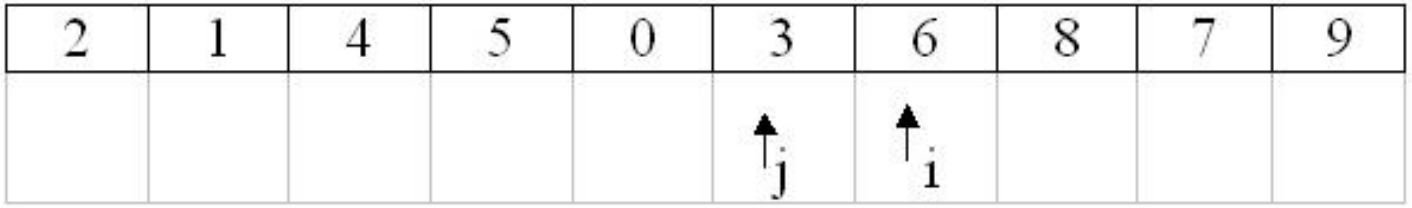


i is now stuck at 9; j moves to the left as long as values are > 6 but STOPS as soon as it moves 1 to the left of i



j stops 1 position to the left of i; now do a swap to place pivot into position i.

Partition Example continued



Now elements < pivot 6 are to the left of 6 and elements > pivot 6 are to the right of 6

Handling Duplicates And Non-Stability of QuickSort

- In the partition algorithm, if the left pointer encounters a value equal to the pivot, it halts; and the right pointer behaves in the same way. This strategy has the effect of evenly distributing duplicates into the left and right halves of the partition.

(continued)

- However, this strategy also makes QuickSort a non-stable sorting algorithm.
- Example: Consider sorting by numeric key the following sequence: (1,a), (1,b), (1,c), (1,d). Assume the pivot always happens to be value at the far right. Using the algorithm, one discovers that after the first partition step, the array looks like this:

  [(1,c), (1,b) | (1,d) | (1,a)]

  For the rest of the sort, (1,a) will always remain to the right of (1,b) and (1,c). Therefore, positions of duplicates are modified by QuickSort. Therefore, QuickSort it is not stable.

Other Choices of Pivot

- *Choosing pivot at random.* Usually a good choice. Repeated calls to random number generator could slow it down a little.
- *Choosing first or last element as pivot.* This is a dangerous approach: using first element as pivot when data is already sorted leads to worst case. If data is known to be random (or is randomized), this is a good choice.
- *Median of Three.* Many consider this the best alternative. If i = lower pos, u = upper pos, pick the median of elements at positions i, u, and $\lfloor (i+u)/2 \rfloor$.

Example of Median of Three

- Input Array *a*

8	1	4	9	3	5	2	7	0	6
0	1	2	3	4	5	6	7	8	9

  - Position $\lfloor(9+0)/2\rfloor$ is position 4
  - a[0] = 8, a[4] = 3, a[9] = 6
    => median of 8,3,6 is 6
  - Therefore, for this call of the function, use 6 as the pivot

Implementing QuickSort

- *Small elements.* As in MergeSort, QuickSort is often implemented so that whenever the input to a recursive call involves only a small number of elements (say 20 or fewer), the work of sorting is handed off to an InsertionSort routine
- *Demo*: QuickSort.java

Comparison With MergeSort

- MergeSort's O(nlog n) worst-case running time makes it reliable, but in practice QuickSort is faster.
- Reason for QuickSort' s faster speed: MergeSort makes many copies of portions of array.

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$\Theta(n^2)$	◆slow (good for small inputs)
insertion-sort	$\Theta(n^2)$	◆slow (good for small inputs)
bubble-sort	$\Theta(n^2)$	◆should never be used
quick-sort	$O(n \log n)$ expected	◆fastest (good for large inputs) ◆ *but…* not stable
merge-sort	$O(n \log n)$	◆stable ◆ fast (good for huge inputs)

The Selection Problem

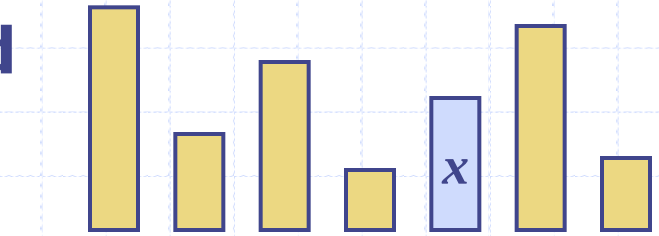
- Given an integer k and n elements x_1, x_2, …, x_n, taken from a total order, find the k-th smallest element in this set.
- Of course, we can sort the set in O(n log n) time and then index the k-th element.

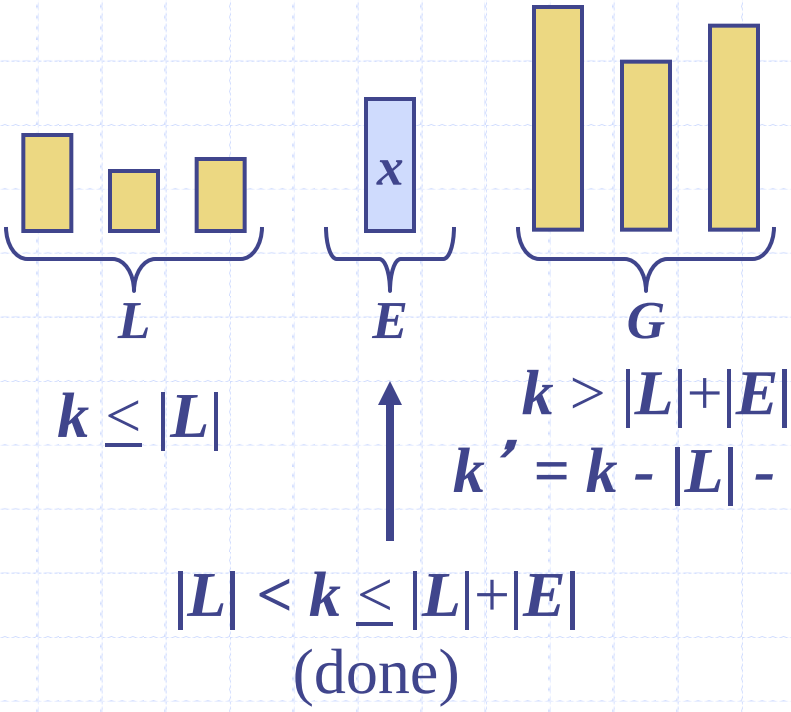
k=3 $\quad$ 7 4 9 <u>6</u> 2 → 2 4 <u>6</u> 7 9

- Can we solve the selection problem faster?

Quick-Select

- Quick-select is a **randomized** selection algorithm for finding the *k*th smallest element in a list:
  - Pick Pivot: 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
  - Search: depending on k, either answer is in E, or we need to recurse in either L or G

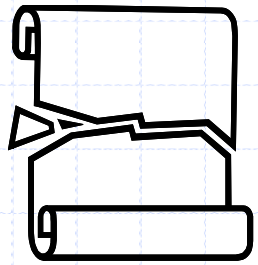




QuickSelect Pseudo-Code

Algorithm *QuickSelect*(*S*,*k*)
  Input sequence ***S*,** rank *k*
  Output *k*th smallest element of S
  $p \leftarrow$ *pickPivot()*
  (*L*,*E*,*G*) $\leftarrow$ *partition*(*S*,*p*)
  if |L| $<$ k $\leq$ |L| + |E| **then**
    return any element of ***E***
  else if k $\leq$ |L| **then**
    return *QuickSelect*(*L*, k)
  else {k > |L| + |E|}
    return *QuickSelect*(*G*, k – |*L*| – |*E*|)

Partition – same as QuickSort

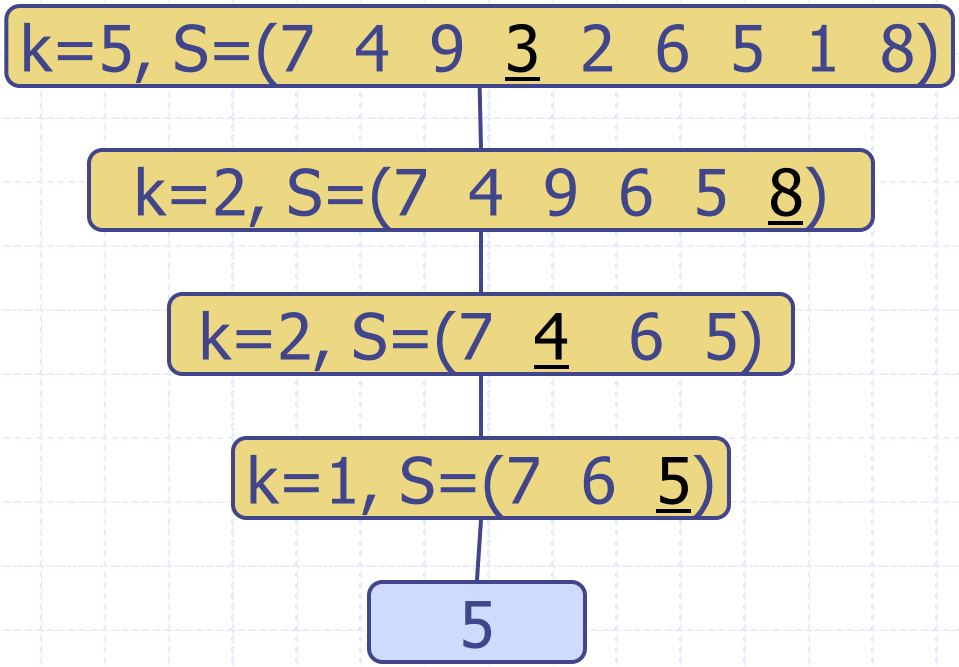


- We partition an input sequence as in the quick-sort algorithm:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-select takes $O(n)$ time
- As before, an in-place version can be formulated, and arrays can be used.

Algorithm *partition*(S, p)
  Input sequence S, pivot p
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < p$
      $L.insertLast(y)$
    else if $y = p$
      $E.insertLast(y)$
    else { $y > p$ }
      $G.insertLast(y)$
  return L, E, G

Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence



Worst-case Running Time

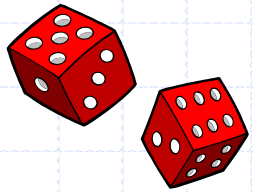
- A worst case for QuickSelect occurs when the input array is sorted, k = 1, and the pivot is always chosen to be the rightmost element
- In this case, the subsequence L always has size just 1 less than previous input and so T(n) is proportional to

  $1 + 2 + 3 + \ldots + (n-1)$

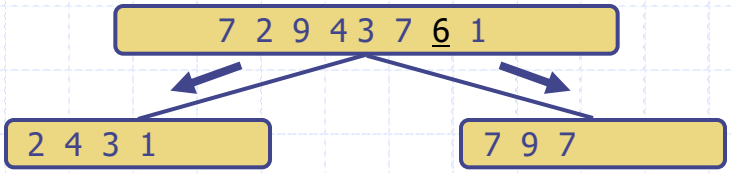
  which is $\Theta(n^2)$

FACT: There is a variation of QuickSelect that runs in O(n) in the worst case, but the constants prevent the algorithm from being efficient in practice.

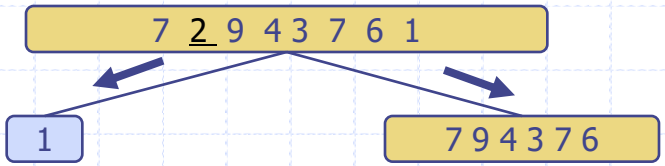
Expected Running Time



- Consider a recursive call of quick-select on a sequence of size s
  - **Good self-call:** the sizes of L and G are each less than $3s/4$
  - **Bad self-call:** one of L and G has size greater than or equal to $3s/4$

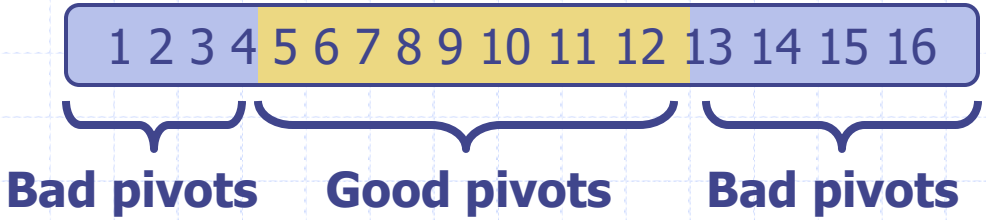


Good call



Bad call

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Analysis of QuickSelect

Algorithm *ModifiedQuickSelect*(*S*, *k*)
  Input sequence *S*, rank *k*
  Output *k*th smallest element of S
  p ← *pickPivot()*
  if *p* **is a good pivot then**
    (*L*, *E*, *G*) ← *partition*(*S*, *p*)
    if |L| < k ≤ |L| + |E| **then**
      return any element of *E*
    else if k ≤ |L| **then**
      return *QuickSelect*(*L*, k)
    else {k > |L| + |E|}
      return *QuickSelect*(*G*, k − |*L*| − |*E*|)
  else {p is a bad pivot} //**treat as wasted call**
    (*L*, *E*, *G*) ← *partition*(*S*, *p*)
    if |L| < k ≤ |L| + |E| **then**
      return any element of *E*
    else if k ≤ |L| **then**
      return *QuickSelect*(*S*, *k*)
    else {k > |L| + |E|}
      return *QuickSelect*(*S*, *k*)

- Good-case Analysis (all pivots good)
  $T(n) = O(n) + T(3n/4)$
- Alternate Bad pivot with Good pivot:
  $T(n) = O(n) + O(n) + T(3n/4)$
- Alternating case accurately represents average case because expected number of self-calls to get a good self-call is 2.
- Therefore:
  $T(n) \leq T(3n/4) + cn$
- Using Master Formula: a=1, b=4/3, c=c, k=1, b^k=4/3 yields
  T(n) is O(n) in the average case.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. The naïve algorithm for finding the kth smallest element in a sequence of n elements requires $\Omega(n^2)$ steps. A clever alternative is to sort the sequence first and then return the value at position k. This approach runs in $\Omega(n \log n)$
2. Using a Divide and Conquer strategy, QuickSelect locates the kth smallest element in O(n) steps, on average.

3. *Transcendental Consciousness* is the field of all possibilities and the home of all knowledge. Contact with this field brings insight into new possibilities and opens awareness to expanded knowledge.
4. *Impulses Within The Transcendental Field.* The unmanifest foundation of the observable world is the lively self-interaction within pure consciousness. Within this field, creation emerges in the collapse of unboundedness to a point, and expansion from point to infinity, with infinite frequency.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, the unmanifest dynamics at the source of creation are appreciated as impulses of one's own being.