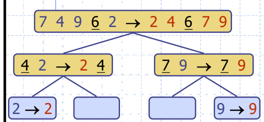
Lesson 10
QuickSort:
Enlivening Hidden Laws of Nature to Manage Change



Wholeness of the Lesson

Quick Sort is another Divide and Conquer sorting algorithm that typically sorts Sequences or Arrays even faster than Merge Sort. QuickSort achieves even greater sorting efficiency by replacing the cruder merge step of MergeSort, which requires repeated access to temporary storage, with a subtle 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.

1

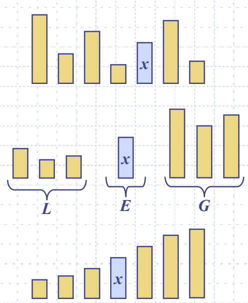
Quicksort

- Divide and Conquer Algorithm
  - The main idea is the moving of a single key (the pivot) to its ultimate location after each partitioning
  - That location is found by
    - moving the smaller values to the left of the pivot and
    - moving the larger values to the right of the pivot
    - the elements are not placed in sorted order in these two partitions to the left and right
- If sorted in place, no need for a combine step
- Earns its name based on its average behavior

2

Quick-Sort

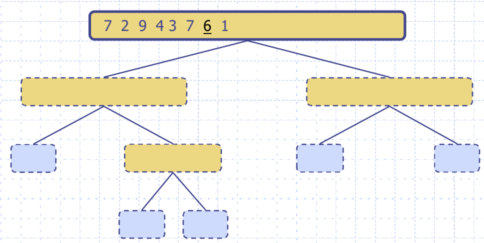
- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



3

Execution Example

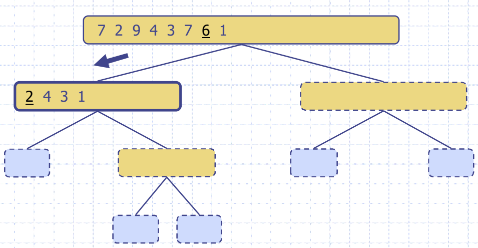
- Pivot selection



4

Execution Example (cont.)

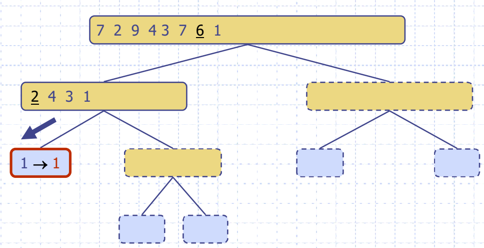
- Partition, recursive call, pivot selection



5

Execution Example (cont.)

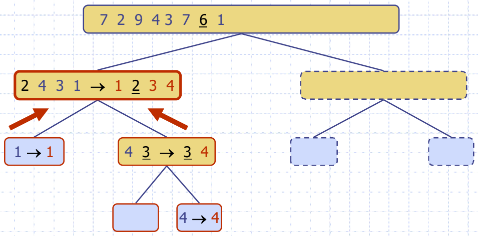
- Partition, recursive call, base case



6

Execution Example (cont.)

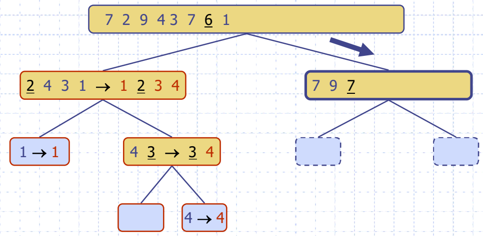
- Recursive call, …, base case, join



7

Execution Example (cont.)

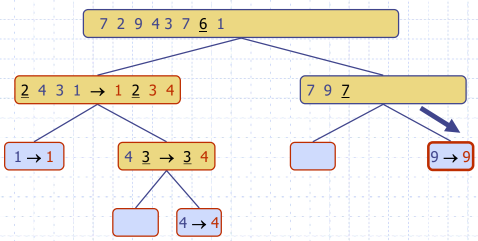
- Recursive call, pivot selection



8

Execution Example (cont.)

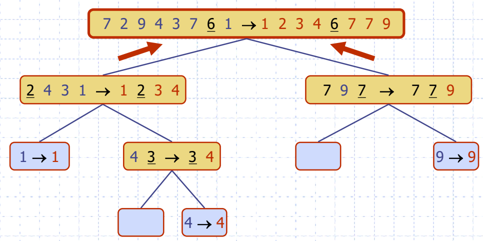
- Partition, …, recursive call, base case



9

Execution Example (cont.)

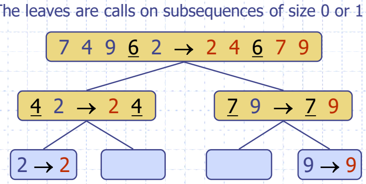
- Join, join



10

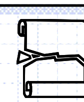
Quick-Sort Tree

- 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



11

Partition



- We partition an input sequence as follows:
  - We examine every element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- For sequence of size n, the partition step takes $O(n)$ time.

```
Algorithm partition(S, p)
    Input List S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty lists
    x ← S.remove(p)
    while (S.size() > 0) do
        y ← S.remove(0)
        if y < x then
            L.insertLast(y)
        else if y > x then
            G.insertLast(y)
        else { y = x }
            E.insertLast(y)
    return (L, E, G)
```


12

QuickSort in Pseudo-Code

```
Algorithm QuickSort(S)
    Input list S
    Output S in sorted order

    if (S.size() > 1) then
        (L,E,G) ← partition(S)
        L ← QuickSort(L)
        G ← QuickSort(G)
        S ← L ∪ E ∪ G
    return S
```

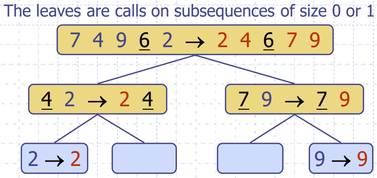
```
Algorithm partition(S)
    Input linked list S
    Output lists L, E, G of the elements
        of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty lists
    p ← pickPivot()
    x ← p.element()
    while (S.size() > 0) do
        y ← S.remove (S.first())
        if y < x then
            L.insertLast(y)
        else if y > x then
            G.insertLast(y)
        else { y = x }
            E.insertLast(y)
    return (L, E, G)
```

Quick-Sort 13

13

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



Quick-Sort 14

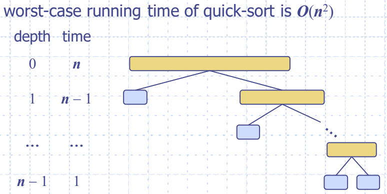
14

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 $O(n^2)$

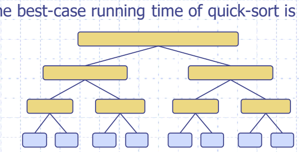


Quick-Sort 15

15

Best-case Running Time

- The best case for quick-sort occurs when the pivot selected is always the median of the array. (Median of the array is the middle element after we sort the array.)
- In that case, both of L and G have size nearly $n/2$.
- The height h of the Quick-Sort tree is $O(\log n)$
  - at each recursive call we divide the sequence in half (n a power of 2)
- The overall amount of work done at each level is $O(n)$
- Thus, the best-case running time of quick-sort is $O(n\log n)$



16

16

Average-case Running time

- The worst-case running time of quick-sort is O(n²); the best-case running time of quick-sort is O(nlog n). We would expect the average-case running time is between them.
- As a matter of fact, using more advanced methods, it can be shown that the average-case running time is O(n log n).

Quick-Sort 17

17

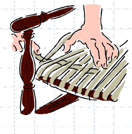
Likelihood of Worst Case

- Using the probabilistic techniques, one can show that probability of QuickSort having its worst case running time on input array of size n is less than 1/n².
- In another words, the average-case O(n log n) running time of QuickSort is extremely likely to occur.

Quick-Sort 18

18

In-Place Quick-Sort

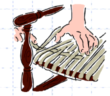


- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

Algorithm *inPlaceQuickSort*(S, l, r)
Input sequence S, ranks l and r
Output sequence S with the elements of rank between l and r rearranged in increasing order
if $l < r$ **then**
  $p \leftarrow$ *inPlacePartition*(S, l, r)
  inPlaceQuickSort(S, l, $p - 1$)
  inPlaceQuickSort(S, $p + 1$, r)

19

In-Place Partitioning



- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

j k
6 2 5 1 0 7 3 5 9 2 7 9 8 9 7 8 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element > pivot.
  - Scan k to the left until finding an element < pivot.
  - Swap elements at indices j and k

j k
6 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 8 9

20

In Place Version of Partition

Algorithm *inPlacePartition*(S, *lo*, *hi*)
  Input Sequence S and ranks lo and hi, $0 \le lo \le hi < S.size()$
  Output the pivot is now stored in S at its sorted rank

```
p ← a random integer between lo and hi
swapElements(S, lo, p)
pivot ← S.elemAtRank( lo )
j ← lo + 1
k ← hi
while  j ≤ k  do
    while  j ≤ k  ∧ S.elemAtRank( j ) < pivot  do
        j ← j + 1
    while  k ≥ j  ∧ S.elemAtRank( k ) ≥ pivot  do
        k ← k – 1

    if j < k then
        swapElements(S, j, k)
        j ← j + 1
        k ← k – 1

swapElements(S, lo, k)  // move pivot to sorted rank
return k
```


21

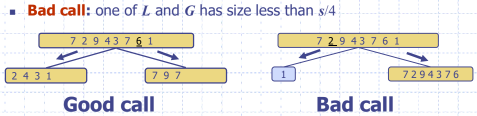
Other Choices of Pivot

- *Choosing pivot at random.* Half the time we get 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 $(i+u)/2$.

22

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are both at least $s/4$
  - **Bad call:** one of L and G has size less than $s/4$



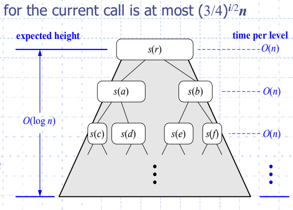
- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:

$\underbrace{1\ 2\ 3\ 4}_{\text{Bad pivots}}\ \underbrace{5\ 6\ 7\ 8\ 9\ 10\ 11\ 12}_{\text{Good pivots}}\ \underbrace{13\ 14\ 15\ 16}_{\text{Bad pivots}}$

23

Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors (half) are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



24

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 (9+0)/2 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 8 or fewer), the work of sorting is handed off to an InsertionSort routine.

Comparison With MergeSort and HeapSort

- MergeSort's O(nlog n) worst-case running time makes it reliable, but in practice QuickSort is the fastest sort for arrays.
- QuickSort is faster than MergeSort for arrays because
  - MergeSort on an array makes many copies of portions of array.
    - Temp is copied back into the array segment
- QuickSort is faster than HeapSort because
  - Locality of reference so fewer cache misses which decreases the constants.
- The problem with the current version occurs when there are a lot of equal keys

What About Duplicates? In Place Version of Partition

```
Algorithm inPlacePartition(S, lo, hi)
    Input Sequence S and indices lo and hi, 0 ≤ lo ≤ hi < S.size()
    Output the pivot is now stored in S at its sorted rank

    p ← a random integer between lo and hi
    swapElements(S, lo, p)
    pivot ← S.elemAtRank( lo )
    j ← lo + 1
    k ← hi
    while j ≤ k do
        while j ≤ k ∧ S.elemAtRank( j ) < pivot do
            j ← j + 1
        while k ≥ j ∧ S.elemAtRank( k ) ≥ pivot do
            k ← k – 1

        if j < k then
            swapElements(S, j, k)

    swapElements(S, lo, k)  // move pivot to sorted rank
    return k
```


In Place Version of Partition that handles many duplicates

```
Algorithm inPlacePartition(S, lo, hi)
    Input Sequence S and indices lo and hi, 0 ≤ lo ≤ hi < S.size()
    Output elements equal to pivot are now stored in S at their sorted rank

    p ← a random integer between lo and hi
    pivot ← S.elemAtRank( lo )
    j ← lo
    k ← hi
  while j ≤ k do
      while j ≤ k ∧ S.elemAtRank( j ) < pivot do
          j ← j + 1
      while k ≥ j ∧ S.elemAtRank( k ) ≥ pivot do
          k ← k – 1
      if j < k then
          swapElements(S, j, k)
  firstEq ← j
  k ← hi
  while j ≤ k do
      while j ≤ k ∧ S.elemAtRank( j ) = pivot do
          j ← j + 1
      while k ≥ j ∧ S.elemAtRank( k ) > pivot do
          k ← k – 1
      if j < k then
          swapElements(S, j, k)

  return (firstEq, k)        // first and last index of elements equal to pivot
```


Main Point

1. QuickSort sorts an input array by first (randomly) picking a *pivot*, then partitioning the array into three pieces *L, E, G,* obtained by and putting the elements smaller than the pivot into *L* and the elements larger than the pivot into *G.* The algorithm then recursively sorts *L* and *G;* the final output concatenates the three partitions, *L* U *E* U *G.* QuickSort's inplace partitioning strategy avoids the extra temporary memory required by MergeSort. Thus, despite its worst case running time (which is highly unlikely), QuickSort has been shown to be the fastest sorting algorithm for large arrays that fit in main memory. *Science of Consciousness*: This advantage in efficiency in QuickSort reflects the general principle that accessing subtler levels of intelligence results in action in the outer level of life that meets more directly and consistently with success.

Main Point

2. In Quicksort, the pivot key is the focal point and controls the whole of the sorting process; after being used to partition the input into two smaller subsequences, the pivot (or all that equal the pivot) is (are) placed in its correct location and these two subsequences are recursively sorted.

   Science of Consciousness: Research studies have shown that the ability to maintain broad awareness and sharp focus is cultured through regular practice of the TM technique.

31 Merge and Quick Sort

31

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
insertion-sort	$O(n^2)$ or $O(n+k)$	◆ excellent for small inputs ◆ fast for 'almost' sorted inputs
merge-sort	$O(n \log n)$	◆ excels in sequential access ◆ for huge data sets
quick-sort	$O(n \log n)$ expected	◆ in-place, randomized ◆ excellent generalized sort
heap-sort	$O(n \log n)$	◆ in-place ◆ fastest/fewest comparisons
bucket-sort radix-sort	$O(n+N)$ $O(d(n+N))$	◆ if integer keys & keys known ◆ faster than quick-sort

32

32

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. MergeSort is an extremely efficient sorting algorithm for sorting large data sets.
2. By eliminating dependency on temporary storage, QuickSort achieves (typically) an even faster running time for sorting large data sets that fit in main memory and have efficient random access (locality of reference).

Quick-Sort 33

33

3. *Transcendental Consciousness* is the silent field that lies beyond the field of thought and perception.
4. *Impulses Within The Transcendental Field.* Beyond the influence of the ups and downs of manifest life, the dynamics of the transcendental field are free to move in any direction with frictionless flow. Enlivening this level in individual awareness brings success and efficiency to the field of action.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, one is free of all dependency on externals. External life is seen clearly to be nothing but the Self.

Quick-Sort 34

34