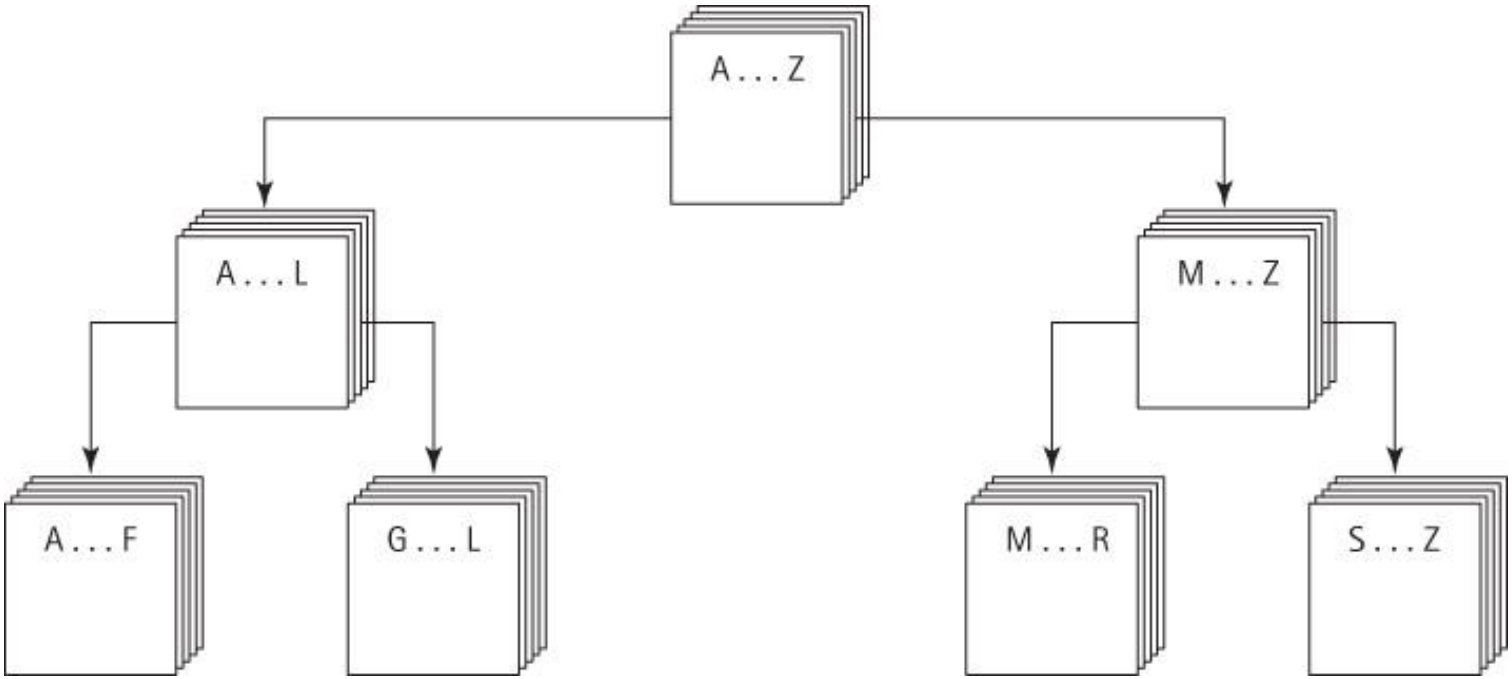
CS430-01
Introduction to Algorithms
5. Quick Sort

Michael Choi
Dept. of Computer Science
IIT

Quick Sort

- A divide-and-conquer algorithm
- Inherently recursive
- At each stage the part of the array being sorted is divided into two “piles”, with everything in the left pile less than everything in the right pile
- The same approach is used to sort each of the smaller piles (a smaller case).
- This process goes on until the small piles do not need to be further divided (the base case).



Divide: Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

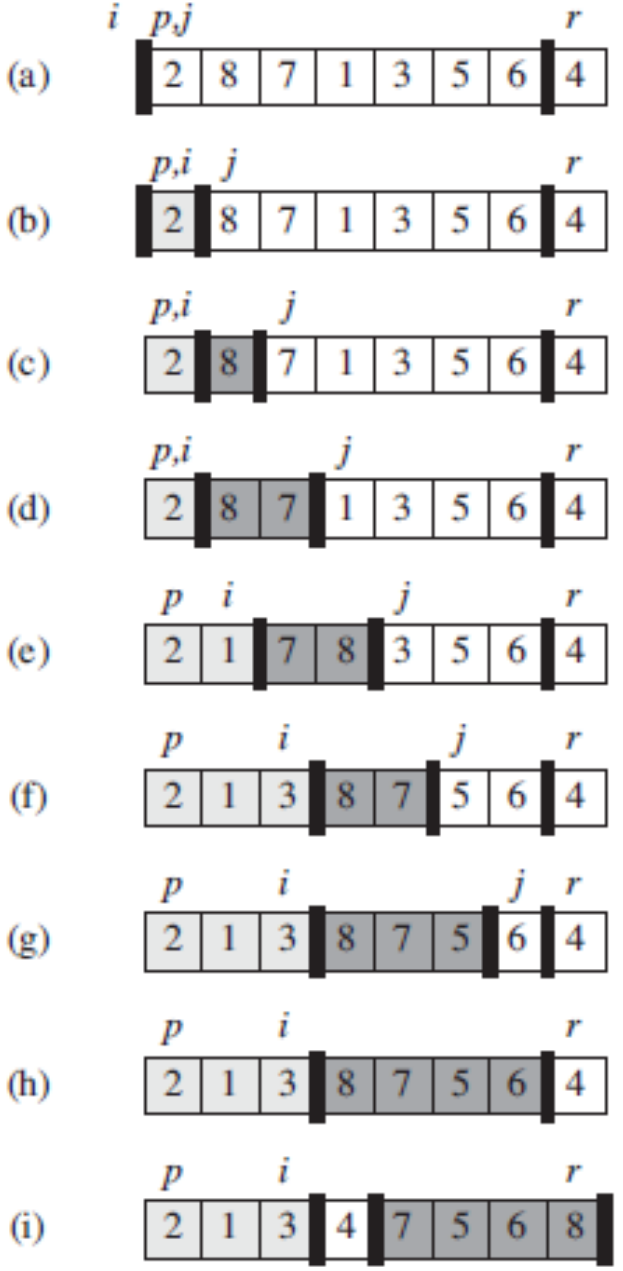
Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

Partitioning Array

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```



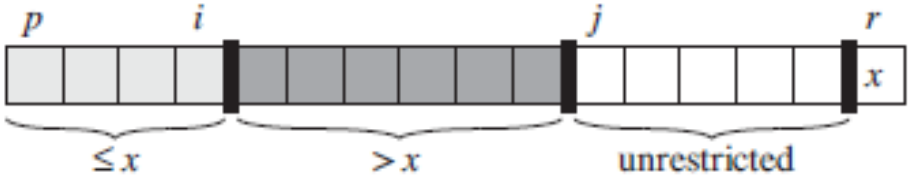


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j..r-1]$ can take on any values.

Initialization: Prior to the first iteration of the loop, $i = p - 1$ and $j = p$. Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$; the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$; the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

Performance

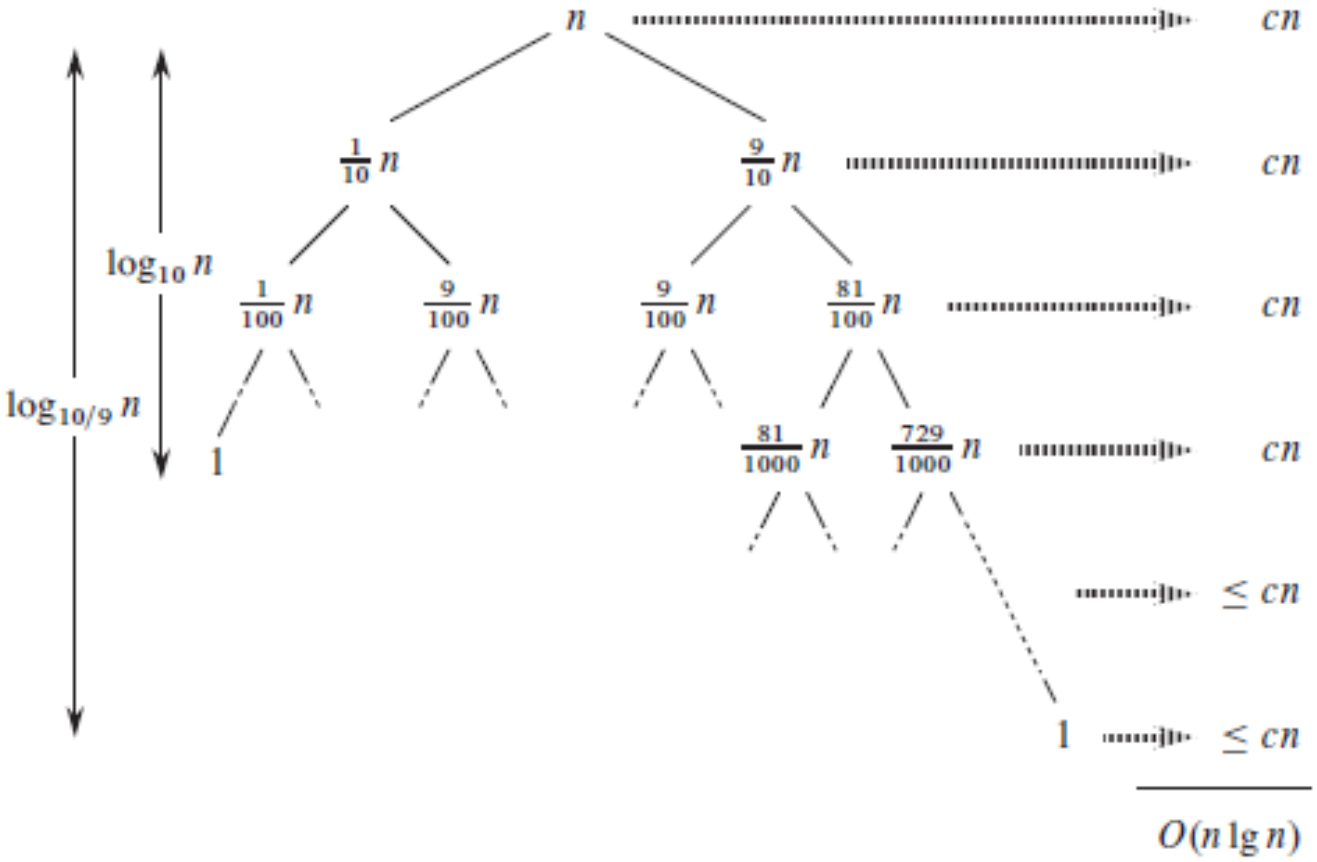


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Quick Sort Summary

Method quickSort (first, last)

Definition: Sorts the elements in sub array values[first]..values[last].

Size: last - first + 1

Base Case: If size less than 2, do nothing.

General Case: Split the array according to splitting value.
quickSort the elements <= splitting value.
quickSort the elements > splitting value.

The Quick Sort Algorithm

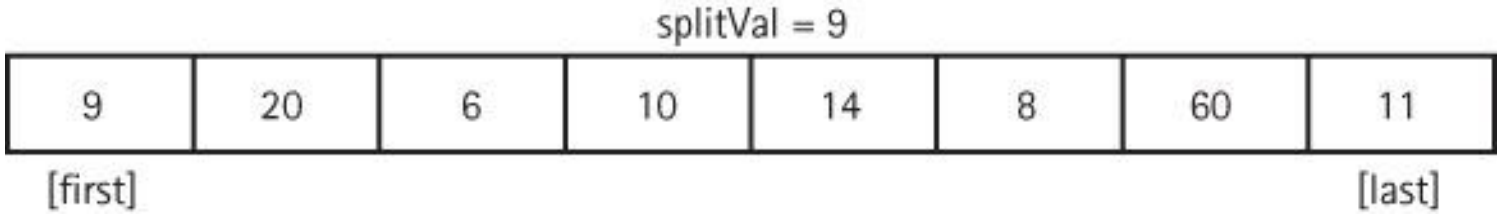
```
quickSort
if there is more than one element in values[first]..values[last]
    Select splitVal
    Split the array so that
        values[first]..values[splitPoint – 1] <= splitVal
        values[splitPoint] = splitVal
        values[splitPoint + 1]..values[last] > splitVal
    quickSort the left sub array
    quickSort the right sub array
```

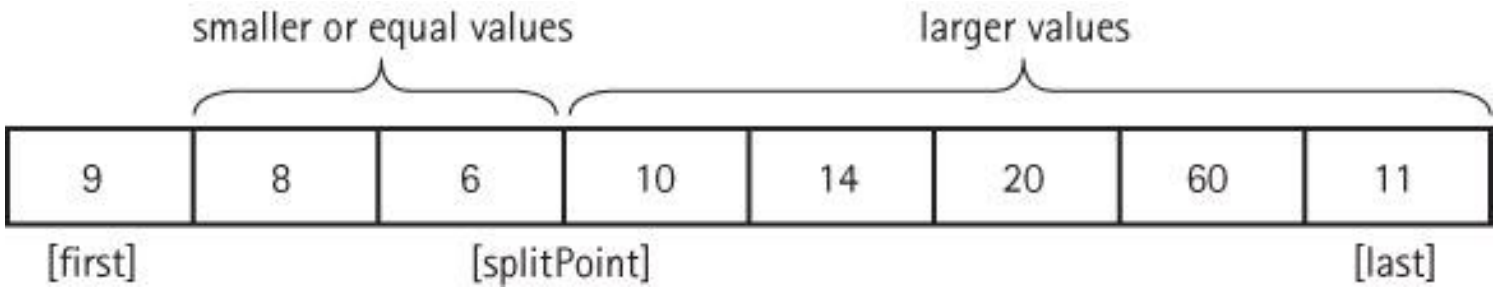
The algorithm depends on the selection of a “split value”, called `splitVal`, that is used to divide the array into two sub arrays.

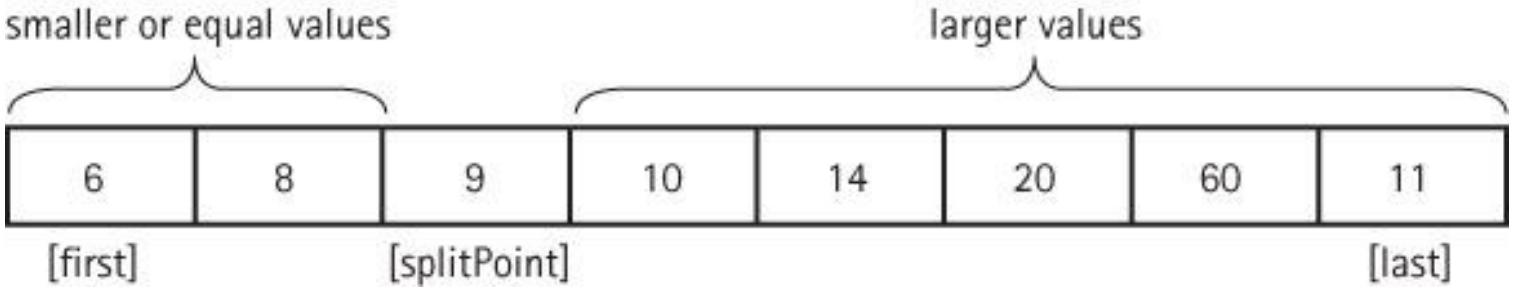
How do we select `splitVal`?

One simple solution is to use the value in `values[first]` as the splitting value.

Quick Sort Steps





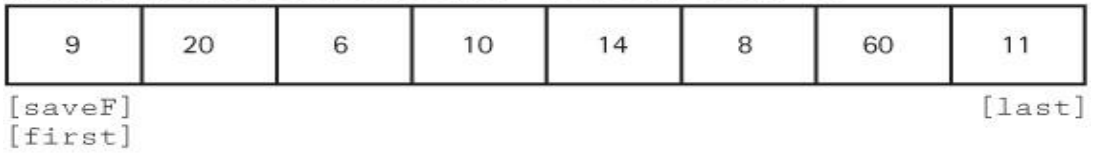


quickSort

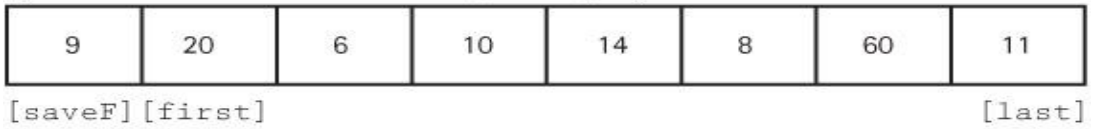
```
static void quickSort(int first, int last)
{
  if (first < last)
  {
    int splitPoint;

    splitPoint = split(first, last);
    // values[first]..values[splitPoint - 1] <= splitVal
    // values[splitPoint] = splitVal
    // values[splitPoint+1]..values[last] > splitVal

    quickSort(first, splitPoint - 1);
    quickSort(splitPoint + 1, last);
  }
}
```

The split operation

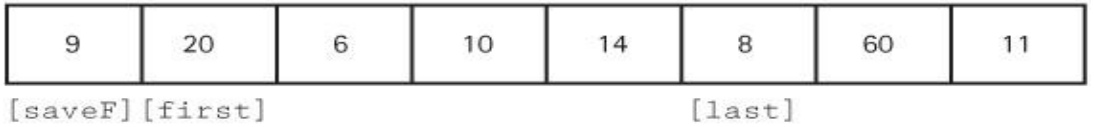
(a) Initialization. Note that `splitVal = values[first] = 9`.



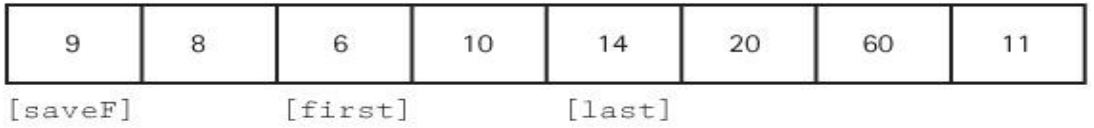
(b) Increment `first` until `values[first]>splitVal`



(c) Decrement `last` until `values[last]<= splitVal`

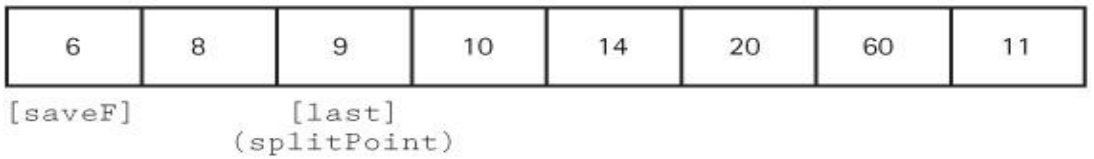


(d) Swap `values[first]` and `values[last]`; move `first` and `last` toward each other



(e) Increment `first` until `values[first]>splitVal` or `first>last`. Decrement `last` until `values[last]<= splitVal` or `first>last`

(f) `first>last` so no swap occurs within the loop. swap `values[saveF]` and `values[last]`



Analyzing Quick Sort

- On the first call, every element in the array is compared to the dividing value (the "split value"), so the work done is O(N).
- The array is divided into two sub arrays (not necessarily halves)
- Each of these pieces is then divided in two, and so on.
- If each piece is split approximately in half, there are O($\log_2 N$) levels of splits. At each level, we make O(N) comparisons.
- So Quick Sort is an O($N \log_2 N$) algorithm.

Drawbacks of Quick Sort

- Quick Sort isn't always quicker.
  - There are $\log_2 N$ levels of splits if each split divides the segment of the array approximately in half. As we've seen, the array division of Quick Sort is sensitive to the order of the data, that is, to the choice of the splitting value.
  - If the splits are very lopsided, and the subsequent recursive calls to quickSort also result in lopsided splits, we can end up with a sort that is $O(N^2)$.
- What about space requirements?
  - There can be many levels of recursion "saved" on the system stack at any time.
  - On average, the algorithm requires $O(\log_2 N)$ extra space to hold this information and in the worst case requires $O(N)$ extra space, the same as Merge Sort.

Quick Sort

- Despite the drawbacks remember that Quick Sort is VERY quick for large collections of random data

- Animation 1
- Animation 2