CSE 326: Data Structures
Quicksort
Comparison Sorting Bound

Brian Curless

Spring 2008

Announcements (5/14/08)

- Homework due at beginning of class on Friday.
- Section tomorrow:
  - Graded homeworks returned
  - More discussion of project, Java and hash tables
  - …
- Reading for this lecture: Chapter 7.

Quicksort

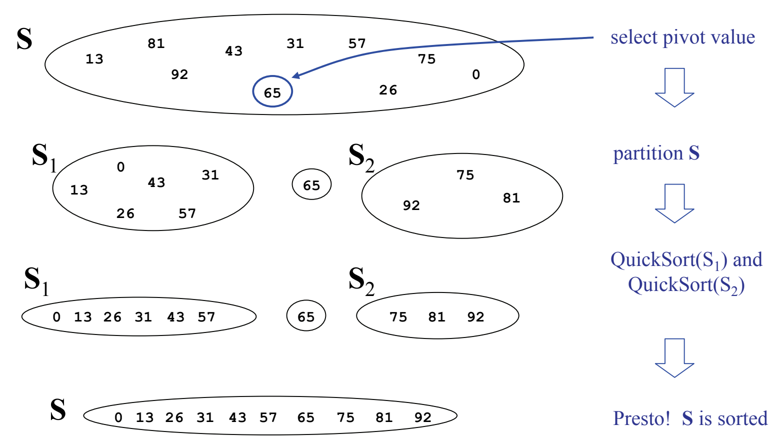
Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does.

Here's the idea for sorting array **S**:

1. Pick an element v in **S**. This is the ***pivot*** value.
2. Partition **S**-{v} into two disjoint subsets, $\mathbf{S}_1$ and $\mathbf{S}_2$ such that:
   - elements in $\mathbf{S}_1$ are all $\leq v$
   - elements in $\mathbf{S}_2$ are all $\geq v$
3. Return concatenation of QuickSort($\mathbf{S}_1$), v, QuickSort($\mathbf{S}_2$)

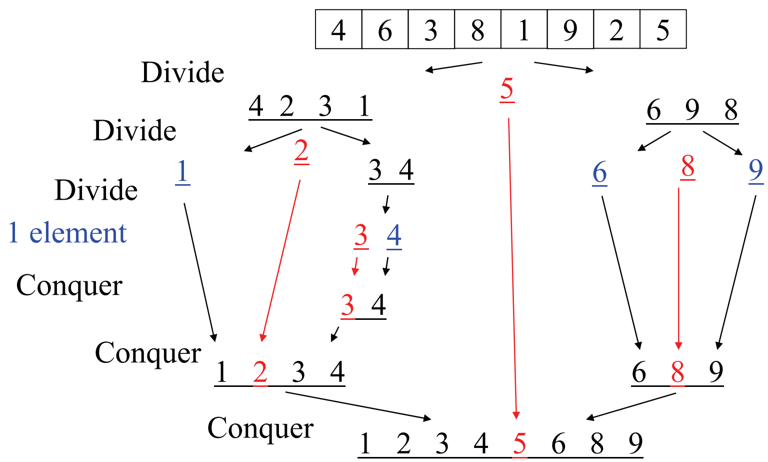
Recursion ends when Quicksort( ) receives an array of length 0 or 1.

The steps of Quicksort



[Weiss]

Quicksort Example



Pivot Picking and Partitioning

The tricky pieces are:

- **Picking the pivot**
  - Goal: pick a pivot value that will cause $|S_1|$ and $|S_2|$ to be roughly equal in size.
- **Partitioning**
  - Preferably in-place
  - Dealing with duplicates.

Picking the Pivot

Median of Three Pivot

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

`medianOf3Pivot(…)`

0	1	4	9	7	3	5	2	6	8

Choose the pivot as the median of three.

Place the pivot and the largest at the right and the smallest at the left.

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays such that:
  - elements in left sub-array are ≤ pivot
  - elements in right sub-array are ≥ pivot
- Can be done in-place with another “two pointer method”
  - Sounds like mergesort, but here we are *partitioning*, not sorting…
  - …and we can do it in-place.

Partioning In-place

Setup: `i` = start and `j` = end of un-partioned elements:

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

Advance `i` until element ≥ pivot:

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

Advance `j` until element ≤ pivot:

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

If `j` > `i`, then swap:

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

Partioning In-place

					i			j		
Advance `i` :	0	1	4	2	**7**	3	5	9	6	8
					i		j			
Advance `j` :	0	1	4	2	7	3	**5**	9	6	8
					i		j			
Swap :	0	1	4	2	**5**	3	**7**	9	6	8
							i j			
Advance `i` :	0	1	4	2	5	3	**7**	9	6	8
						j	i			
Advance `j` :	0	1	4	2	5	**3**	7	9	6	8
						j	i			
`i` > `j`, swap in pivot, partition done!	0	1	4	2	5	3	**6**	9	**7**	8

$S_1 \leq$ pivot pivot $S_2 \geq$ pivot

Partition Pseudocode

```
Partition(A[], left, right) {
  v = A[right]; // Assumes pivot value currently at right
  i = left;     // Initialize left side, right side pointers
  j = right-1;

  // Do i++, j-- until they cross, swapping values as needed
  while (1) {
    while (A[i] < v) i++;
    while (A[j] > v) j--;
    if (i < j) {
      Swap(A[i], A[j]);
      i++; j--;
    }
    else
      break;
  }
  Swap(A[i], A[right]); // Swap pivot value into position
  return i;             // Return the final pivot position
}
```

Complexity for input size *n*?

Quicksort Pseudocode

Putting the pieces together:

```
Quicksort(A[], left, right) {
  if (left < right) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex – 1);
    Quicksort(A, pivotIndex + 1, right);
  }
}
```


QuickSort: Best case complexity

```
Quicksort(A[], left, right) {
  if (left < right) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex – 1);
    Quicksort(A, pivotIndex + 1, right);
  }
}
```


QuickSort: Worst case complexity

```
Quicksort(A[], left, right) {
  if (left < right) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex – 1);
    Quicksort(A, pivotIndex + 1, right);
  }
}
```


QuickSort: Average case complexity

Turns out to be **O(*n* log *n*)**.

See Section 7.7.5 for an idea of the proof.
Don’t need to know proof details for this course.

Many Duplicates?

An important case to consider is when an array has many duplicates.

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

`medianOf3Pivot(…)`

0	6	6	6	6	6	6	6	6	8

Partitioning with Duplicates

Setup: `i` = start and `j` = end of un-partioned elements:

	i						j		
0	6	6	6	6	6	6	6	6	8

Advance `i` until element ≥ pivot:

	i						j		
0	**6**	6	6	6	6	6	6	6	8

Advance `j` until element ≤ pivot:

	i						j		
0	6	6	6	6	6	6	**6**	6	8

If `j` > `i`, then swap:

	i						j		
0	**6**	6	6	6	6	6	**6**	6	8

Partitioning with Duplicates

Advance `i,j`:

		i				j			
0	6	**6**	6	6	6	**6**	6	6	8

Swap:

		i				j			
0	6	**6**	6	6	6	**6**	6	6	8

Advance `i,j`:

			i		j				
0	6	6	**6**	6	**6**	6	6	6	8

Swap:

			i		j				
0	6	6	**6**	6	**6**	6	6	6	8

Advance `i,j`:

				i j					
0	6	6	6	**6**	6	6	6	6	8

Finish:

				i j					
0	6	6	6	**6**	6	6	6	**6**	8

Partitioning with Duplicates:Take Two

Start `i` = start and `j` = end of un-partioned elements:

	i						j		
0	6	6	6	6	6	6	6	6	8

Advance `i` until element **>** pivot (and in bounds):

							i j		
0	6	6	6	6	6	6	**6**	6	8

Advance `j` until element **<** pivot (and in bounds):

	j						i		
0	**6**	6	6	6	6	6	**6**	6	8

Finish:

	j						i		
0	6	6	6	6	6	6	**6**	**6**	8

Is this better?

Partitioning with Duplicates: Upshot

It's better to stop advancing pointers when elements are equal to pivot, and then just do swaps.

Complexity of quicksort on an array of identical values?

Can we do better?

Important Tweak

Insertion sort is actually better than quicksort on small arrays. Thus, a better version of quicksort:

```
Quicksort(A[], left, right) {
  if (right - left ≥ CUTOFF) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex - 1);
    Quicksort(A, pivotIndex + 1, right);

  } else {
    InsertionSort(A, left, right);
  }
}
```

CUTOFF = 10 is reasonable.

Properties of Quicksort

- $O(N^2)$ worst case performance, but $O(N \log N)$ average case performance.
- Pure quicksort not good for small arrays.
- No iterative version (without using a stack).
- "In-place," but uses auxiliary storage because of recursive calls.
- Stable?
- Used by Java for sorting arrays of primitive types.

How fast can we sort?

Heapsort, Mergesort, and Binary Tree Sort all have $O(N \log N)$ **worst** case running time.

These algorithms, along with Quicksort, also have $O(N \log N)$ **average** case running time.

Can we do any better?

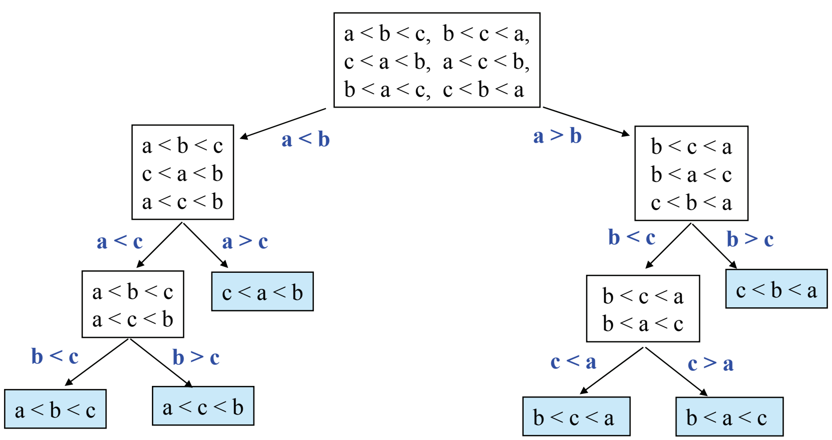
Sorting Model

- Recall our basic assumption: we can <u>only compare two elements at a time</u>
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given *N* elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c (*N* = 3)

Permutations

- How many possible orderings can you get?
  - Example: a, b, c (*N* = 3)
  - (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - 6 orderings = 3•2•1 = 3! (i.e., “3 factorial”)
  - All the possible permutations of a set of 3 elements
- For *N* elements
  - *N* choices for the first position, (*N*-1) choices for the second position, …, (2) choices, 1 choice
  - *N*(*N*-1)(*N*-2)⋯(2)(1)= <u>*N*! possible orderings</u>

Decision Tree

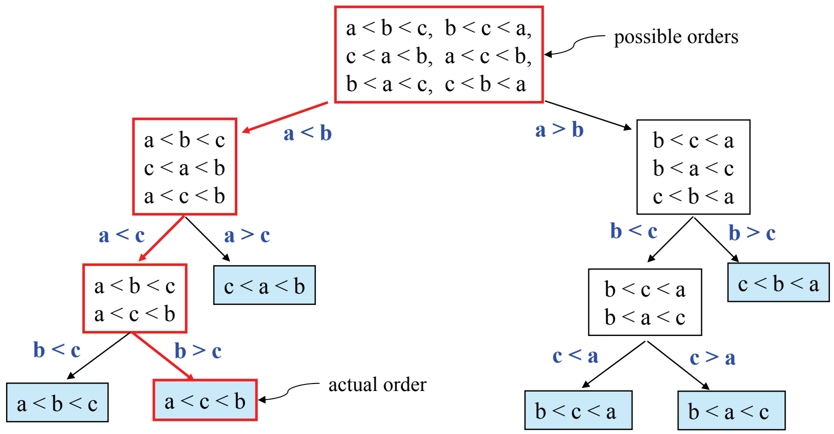


The leaves contain all the possible orderings of a, b, c.

Decision Trees

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
    - i.e., the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for *N* distinct elements?
- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

Decision Tree Example

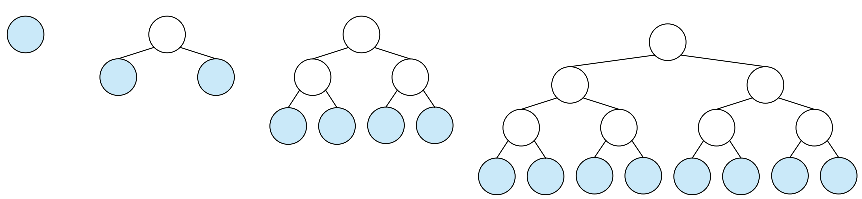


Decision Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - ie, by making comparisons
  - Each decision reduces the possible solution space by one half
- We will focus on worst case run time. Observations:
  - Worst case run time is ≥ maximum number of comparisons.
  - Maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree.

How many leaves on a tree?

Suppose you have a binary tree of height *h*. How many leaves in a perfect tree?



We can prune a perfect tree to make any binary tree of same height. Can # of leaves increase?

Lower bound on Height

- A binary tree of height h has at most 2^h leaves
  - Can prove formally by induction
- A decision tree has *N*! leaves. What is its minimum height of that tree?

Lower Bound on log(*N*!)

$\Omega(N \log \underline{N})$

Worst case run time of any comparison-based sorting algorithm is $\mathbf{\Omega(N \log N)}$.

Can also show that **average case** run time is also $\mathbf{\Omega(N \log N)}$.

Can we do better if we don't use comparisons? (Huh?)