

## QUICKSORT

The quicksort algorithm, developed by C. A. R. Hoare, is a divide and conquer sorting algorithm.

- **Divide:** An array  $A[\text{left}, \text{right}]$  ( $\text{left} < \text{right}$ ) is partitioned into three subarrays.

Array (1):  $A[\text{left}, \text{middle}-1]$

Array (2):  $A[\text{middle}]$  (size is one)

Array (3):  $A[\text{middle}+1, \text{right}]$

This is done such that

- Every element in (1) is  $\leq$  every element in (2)
- Every element in (3)  $>$  every element in (2).

- **Conquer:** Recursively sort the arrays  $A[\text{left}, \text{middle}-1]$  and  $A[\text{middle}+1, \text{right}]$ .

To sort the array  $A[1..n]$ , a call  $Q\text{-sort}(A,1,n)$  is made.

```
function Q-sort(array A, integer index left, integer index right)
  if ( left < right )
    middle <- Partition(A, left, right)
    Q-sort(A, left, middle-1)
    Q-sort(A, middle+1, right)

function Partition(array A, integer index left, integer index right)
  {output is the integer index middle.  A is changed in place. }
  {for now, the partitioning pivot will be A[left] }

  pivot <-- A[left]
  L <-- left+1 {note that left < right}
  R <-- right
  while ( L <= R ) do
  begin
    while ( L <= right ) AND ( A[L] <= pivot ) do
      L <-- L + 1
    while ( R >= left ) AND ( A[R] > pivot ) do
      R <-- R - 1
    if ( L < R )
      swap (A[L],A[R]) {swap is a basic three-line function
                       to exchange the array elements }
      L <-- L+1
      R <-- R-1
  end(while)          {A[left] is the pivot.  A[R] <= pivot }
  swap(A[left],A[R])
  return R           {R is the middle. }
```

★ Example:

Let  $\text{middle} = R$ . At the end of the partition algorithm, the following is true.

1.  $\text{middle} \geq \text{left}$ . i.e.,  $\text{middle}$  is guaranteed to point into the array.
2. All elements in  $A[\text{left}, \text{middle}-1]$  are  $\leq A[\text{middle}]$ .
3. All elements in  $A[\text{middle}+1, \text{right}]$  are  $> A[\text{middle}]$ .

### Analysis of Quicksort

The Quicksort algorithm does several types of operations, including comparisons of array elements, comparisons of array indices (such as  $\text{left}$ ,  $\text{right}$ ,  $R$ , and  $L$ ), and swaps of data elements. For simplicity, we'll analyze the total number of comparisons of array elements done in the algorithm. Analysis of the other types of operations is similar.

All comparisons of array elements are done in the Partition function. How many comparisons of array elements are done when the input array contains  $n$  elements? Note that every element of the array except the leftmost element is compared with the pivot element exactly once, except possibly at the end when the outer while loop is entered with  $L=R$ . In the case that the outer loop is entered with  $L=R$ , both  $A[L]$  and  $A[R]$  are compared with pivot, so the element  $A[L] = A[R]$  is compared twice with the pivot. Therefore the number of comparisons in the Partition function is:



However, the Partition function can be optimized to ensure that the number of comparisons is always exactly  $n - 1$  (this is left as a homework exercise).

The next step is to understand what is the total number of comparisons of the algorithm. This depends on the value of  $middle$  at each recursive call. So, we need to make some assumptions about the value of  $middle$ .

Let  $C(n)$  be the number of comparisons, assuming that  $middle$  always lies exactly in the center of the array. That is, suppose that  $middle$  lies exactly half way between  $left$  and  $right$  on every recursive call of the algorithm. For simplicity, let  $n$  be odd and be of the form  $n = 2^k - 1$  for some  $k$ . Then,  $A[left, middle - 1]$  and  $A[middle + 1, right]$  both have  $(n - 1)/2$  elements. Therefore, we have that

★

$$C(n) =$$

Note that  $C(1) = 0$ . The iteration method can be used to solve this recurrence to obtain the result  $C(n) = \Theta(n \log n)$ . This turns out to be the best case for quicksort.

Next, suppose that  $middle = left$  in each recursive call. Thus, the rest of the elements are in  $A[middle + 1, right]$ , and  $A[left, middle - 1]$  is empty. Then, for the first of the two recursive calls in Q-Sort, the number of comparisons is equal to zero. In this case, if  $C(n)$  is the total number of comparisons done by the algorithm, we have

★

$$C(n) =$$

## Randomized Quicksort

We want to avoid the worst-case scenario of Quicksort, where *pivot* is either the smallest or largest element. The idea is to choose *pivot* uniformly at random from the array elements. Specifically, the partition function could be changed by adding the following two lines at the beginning:

```
choose i randomly and uniformly from the range [left,right]
exchange A[i] with A[left]
```

As a result, pivot is equally likely to be any element of the array. Let  $C(n)$  be the expected number of comparisons done on an array of  $n$  elements, averaging over all possible runs of the algorithm. In what follows, we'll also assume that the elements of the array are all distinct. Then,

$$C(n) = (n-1) + (\text{expected number of comparisons done in the two subproblems})$$

We want to continue get a recurrence for  $C(n)$ .

There are  $n$  possible outcomes for where *pivot* lies. (Here we use the assumption that the elements are all distinct.) If *pivot* has rank  $i$  where  $1 \leq i \leq n$ , then the expected number of comparisons done is equal to the expected number of comparisons done on a subproblem of size  $i - 1$  plus the expected number of comparisons done on a subproblem of size  $n - i$ . The expected number of comparisons done on the two subproblems is:

★

We expect that the solution to this recurrence lies between  $\Theta(n \log n)$  and  $\Theta(n^2)$ . Suppose we optimistically guess that the solution is  $O(n \log n)$  and try to prove this by induction.

**Claim 6**  $C(n) \leq an \lg n$  for  $n \geq 0$  and for some  $a \geq 0$ .

★ **Proof:** The basis case is when  $n = 1$ . We know that  $C(1) = 0$  and so the claim is trivially true.

**Induction:** Let  $n \geq 2$ . Assume that the claim is true for  $0 \leq i < n$ . Then

★

This proof can be extended to show that  $C(n) = \Theta(n \lg n)$ .

Quicksort is popular in practice:

- It is an in-place algorithm (as opposed to mergesort).

This means that the array can be sorted without needing significant extra storage.

- The expected number of exchanges (swaps) done is small.
- There are many practical improvements that make the basic algorithm faster; these are discussed in Hoare's paper. For example, (i) choose the pivot to be the median of three randomly chosen elements, rather than choosing it uniformly at random; (ii) use "sentinels" in the partitioning code. The following code appears in the partition function:

```
while (L<= right) and (A[L]<=pivot) do
  L <- L + 1
```

The comparison of  $L$  with `right` can be omitted if we have a *sentinel* i.e. a very large number in array position  $n + 1$ ; (iii) finally, Hoare's paper has ideas on reducing number of swaps, or exchanges.

## A Lower Bound for Sorting Algorithms

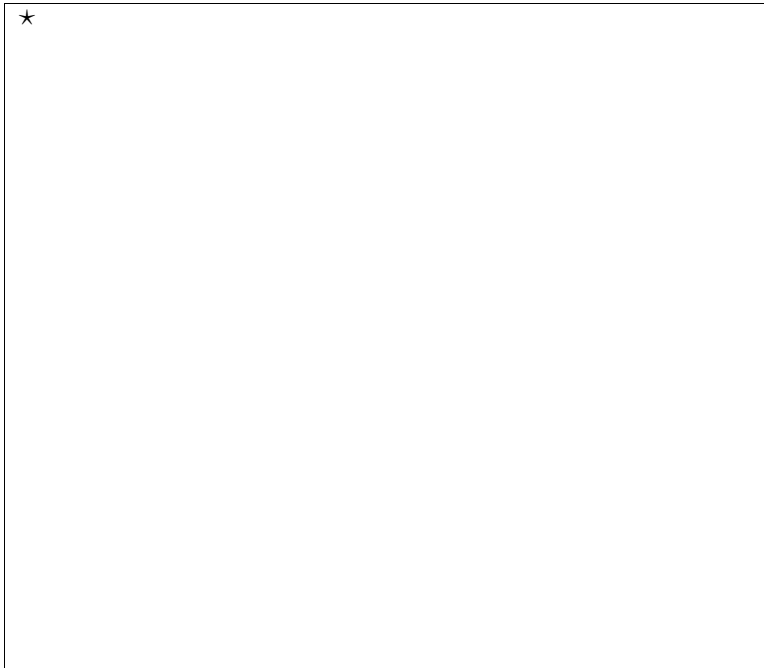
Let us begin by introducing binary decision trees

$h$  — number of levels (tree height)

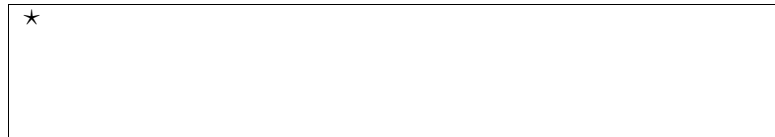
$N$  — number of nodes

$n$  — number of leaves

The lower bound here applies to comparison-based sorting algorithms (not to algorithms that examine bits of the element to be sorted.) We show that any such deterministic algorithm must do  $\Omega(n \log n)$  comparisons in the worst case. We represent sorting algorithms on inputs with  $n$  elements using a tree. e.g. for insertion sort on 3 elements.



How many leaves does a comparison tree for sorting  $n$  elements have? Hint: each different permutation of the elements occurs at least once as a leaf.



Define the **height of a tree** to be the number of **edges** on the longest path from the **root** to a leaf. Note that the height of a decision tree for a given sorting algorithm on inputs of size  $n$  equals the worst-case number of comparisons that the algorithm does on an input of size  $n$ .

Since a comparison tree is a binary tree, if it has height  $h$ , the number of leaves in the tree is at most  $2^h$ . Therefore, if  $h$  is the height of a comparison tree for sorting  $n$  elements, we must have  $n! \leq 2^h$ . We use this to show  $h = \Omega(n \lg n)$ .