CIS 5511. Programming Techniques

Sorting (2)

1. Quicksort

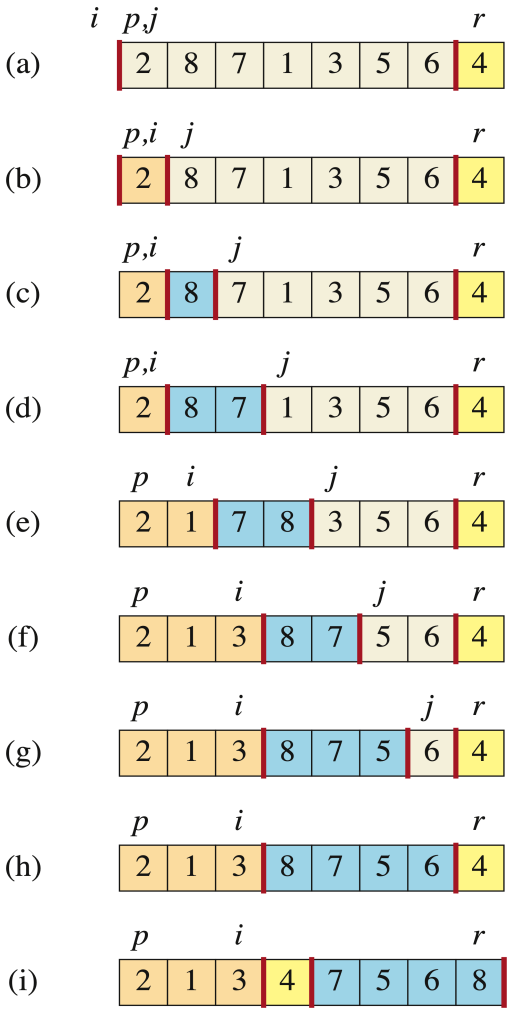
Quicksort first does a partition with a pivot value on the array to be sorted, then recursively sort the two subarrays separately.

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

The (Lomuto) partition algorithm separates the elements of array A between index p and r that are smaller than the pivot value from the elements larger than it.

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                     // index of a new slot in the low side
6          exchange A[i] with A[j]       // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

Example of partition:



Loop invariant:

At the beginning of each iteration of the loop of lines 3–6, for any array index k,

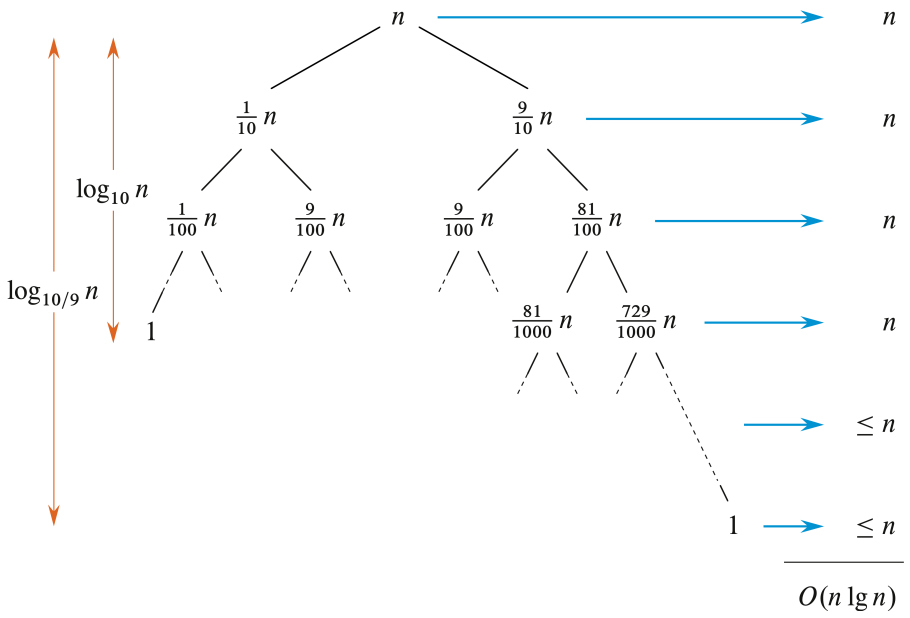
1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Hoare partition let two index values start at the two ends of the array, then move toward each other. In the process, a pair of elements are exchanged into their corresponding regions. The following algorithm uses the first element as the pivot and returns the index j such that elements in A[1..j] are not larger than elements in A[j+1..r]:

```
HOARE-PARTITION(A, p, r)
1   x = A[p]
2   i = p − 1
3   j = r + 1
4   while TRUE
5       repeat
6           j = j − 1
7       until A[j] ≤ x
8       repeat
9           i = i + 1
10      until A[i] ≥ x
11      if i < j
12          exchange A[i] with A[j]
13      else return j
```

This partition algorithm does not return the index of the pivot, so the quicksort algorithm should be modified accordingly.

The best case of Quicksort has $T(n) \le 2T(n/2) + \Theta(n) = O(n \lg n)$. The result is the same even if the partition is uneven, where $T(n) = T(n/a) + T(n(1 - 1/a)) + \Theta(n) = O(n \log_a n) = O(n \lg n)$.



The worst case of quicksort happens when the pivot values are always at the end of the array --- in that case it becomes selection sort, and the running time is $T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$. A worst case is when the input is already sorted. To prevent the worst-case from consistently happening, pivot values can be randomly picked, or selected from a small number of elements (e.g., using indices 1, n/2, n).

In average case, when the partitions are sometimes good and sometimes bad, the expected running time will be closer to the best-case than to the worst case, so this algorithm is usually considered as $O(n \lg n)$.

To reduce the overhead of recursion, the second recursive call in Quicksort can be replaced by an iteration:

```
TRE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and then sort the low side.
3      q = PARTITION(A, p, r)
4      TRE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

2. Comparison sorts

All the previous sorting algorithms belong to "comparison sort", where sorting is done only by comparisons among the values to be sorted.

The major comparison sorting algorithms are summarized in the following table in their recursive forms, with time function *T(n) = D(n) + a*T(f(n)) + C(n)*:

sort(first, last)	*Linear: 1 * T(n – 1)*	*Binary: 2 * T(n / 2)*
with preprocessing *D(n) = O(n)* *C(n) = O(1)*	SelectionSort select(last); **sort(first, last – 1)**;	QuickSort partition(first, last); **sort(first, middle – 1)**; **sort(middle+1, last)**;
with post-processing *D(n) = O(1)* *C(n) = O(n)*	InsertionSort **sort(first, last – 1)**; insert(last);	MergeSort **sort(first, middle)**; **sort(middle+1, last)**; merge(first, middle, last);

For a sorting problem with n different items, there are $n!$ possible input, in terms of the relative order of the items (and the actual values of items does not matter), and each needs a different processing path. Therefore it is enough to analyze an array containing integers 1 to n. Comparison sorts can be viewed in terms of *decision trees* to distinguish the cases. If we see each comparison as a node, then a sequence of comparisons forms a binary tree. The execution of an algorithm corresponding to a path from the root to a leaf, and each leaf corresponds to a different execution path for a specific order.

The decision tree corresponding to a (comparison-based) sorting algorithm must have $n!$ leaves to distinguish all possible inputs, and the worst case corresponds to the longest path from the root to a leaf. Since a binary tree of height h has no more than 2^h leaves, we have $n! \le 2^h$, that is, $\lg n! \le h$, while the former is $\Theta(n \lg n)$ (see Page 67, Equation 3.28). Therefore, $h = \Omega(n \lg n)$.

This result gives us a lower bound of the comparison-based sorting problem, independent of the algorithm used to solve the problem.

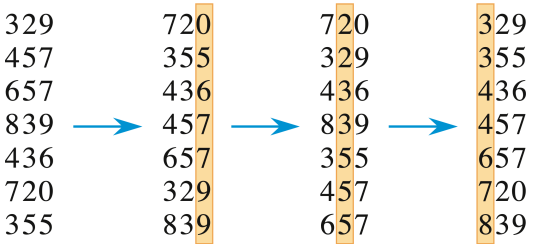
As for the best case, it is easy to see that each item must be compared at least once, so the

lower bound is Ω(*n*), which is also the lower bound of the average case.

3. Sorting in linear time

If we change the definition of the sorting problem by assuming more information beside the relative order of the values, then it is possible to get linear sorting algorithms.

- Counting sort works if the key only takes finite possible values (such as an integer from 0 to *k*). In the simplest case, the number of occurrence of each value is counted, then the value is repeated the same number of times in the output.
- Radix sort processes numbers one digit at a time while keeping the order already built. The worst-case cost is *O(w * n)*, where *n* is the number of keys, and *w* is the key length. Example:



- The Dutch national flag problem is also linearly-solvable using three-way-partition.