M. ALI NAWAZ ADIL

www.alinawaz.com/DAA

FA05-BS-0006

Design and Analysis of Algorithms
Muhammad Ali Jinnah University

Design, analysis, implementation & improvements in

Quicksort…

DESIGN & ANALYSIS OF ALGORITHMS

Design, analysis, implementation & improvement in Quicksort

Table of Contents

Sorting Algorithms

All sorting algorithms can be divided into one of two categories: those which are **comparison based** and those which are not. A comparison based algorithm is one that orders the data set by weighing the key value of one element against that of one or many other elements. Conversely a **non comparison based** sort puts the target data set into order without consideration of two or more data items.

The efficiency of each presented algorithm is discussed in detail. When considering the efficiency of a given algorithm it is useful to examine its performance in several cases. These include sorting data sets of various sizes, data sets already in sorted order, data sets in reverse sorted order, and data sets in random order. Sometimes the number of comparisons performed by a particular algorithm does not matter but the number of swaps must be minimized because swapping records is **expensive**. Other we need not worry about space but want an algorithm that sorts as quickly as possible. It is *vital* to know something about the data which you are sorting in order to choose the algorithm that best matches your needs.

The Quicksort

Over the years computer scientists have come up with many different algorithms for sorting data. The Inventor of Quicksort is Sir Charles A. R. Hoare (1980 ACM Turing Award). C.A.R. Hoare's Quicksort, however, is generally regarded as the most efficient and fastest sorting algorithm in the average case.

A Quicksort operates by selecting a value called the **pivot point** and then arranging the data being sorted such that all data items with a key value less than the that of the pivot point appear at the beginning of the data structure and all data items greater than or equal to the pivot value are moved towards the end of the data structure.

Thus, the data set to be sorted is **partitioned** into two pieces; the k items less in value than the pivot item's value, and the $(n - k)$ items greater than or equal to the pivot value. It is essential to note that neither of these two partitions are sorted as they are built; all we know after the partitioning operation is that every item to the "left" of the pivot is less in value than it, and every item "right" of the pivot point is greater than or equal to it.

The Quicksort continues, at this point, to process the two partitions discussed above. In each sub-partition a new pivot value is selected which allows yet another sub-division of the data set. This process of selecting a new pivot value and then sub-dividing a range of data into two more partitions continues again and again until the size of a resulting partition becomes small enough that it can be easily explicitly sorted. This point usually occurs when there are two or fewer items remaining in a partition because such ranges can be easily put into order with a very little effort.

Quicksort pseudo code

```
quicksort(A,p,r)

Input: A[p..r] array of elements (p, r start and end
indices)

1. if (p < r) then

2. q ?  partition(A, p, r)

3. quicksort(A, p, q)

4. quicksort(A, q+1,r)

5. endif
```

Picking a Pivot Value

As you might imagine, selecting a good pivot value is crucial to the success and performance of this sort. In the ideal case we want to pick the statistical median key in a partition as the pivot value and, thus, split the partition into equal halves.

The simplest way to pick a pivot value is to use the value of first data item. This method has the advantage of being a very fast but, unfortunately, operates on a sometimes faulty supposition. If the data being sorted is in near-sorted order then the first item in a given partition is very likely to have the lowest value of all items in said partition. This leads to unbalanced partitioning of the data set. To avoid infinite recursion usually Quicksort is implemented to partition into a set containing the pivot element and a set containing the *n*-1 others.

Below is an example pivot selecting routine written in C. It uses the greater of the first two distinct values in a partition as the pivot point and returns NONE if the partition does not need to be sorted any further.

Picking Pivot value program in C.

```
quicksort(A,p,r)
/* NONE must be a value that cannot occur as a key */

#define NONE -1typedef key int;
typedef data struct
{
  key thekey;
  char *therest;
};

/* Return the pivot value or NONE if the partition
does not need to be sorted any further. */

key selectpivot(data *array, int left, int right)
{

key first = value(array[left]); /* the first key */
int lcv;                        /* loop control */

  for (lcv = left + 1; lcv <= right; lcv++)
  {
    if (value(array[lcv]) > first)
       return (value(lcv));

    else if (value(array[lcv]) < first)
       return first;
  }

/* if we get here the partition is homogeneous */

  return (NONE);
}

key value(data *item)
{
  return (item->thekey);
}
```

Above, the routine selects the value of a pivot point and returns it, or, if all the elements in the partition are equal in value, returns NONE to indicate that further sorting of this partition is not necessary. NONE must be a value that will not appear in the array. If you cannot predict which values will appear in your data set, it would be better to write the above routine to return the index of the pivot value rather than the value itself. Then it could return an invalid index number to specify a homogeneous partition.

Partitioning

Now, here is code to do the actual work of dividing an input range into two partitions based on a pivot point. Again, it is written in C:

Partitioning program in C.

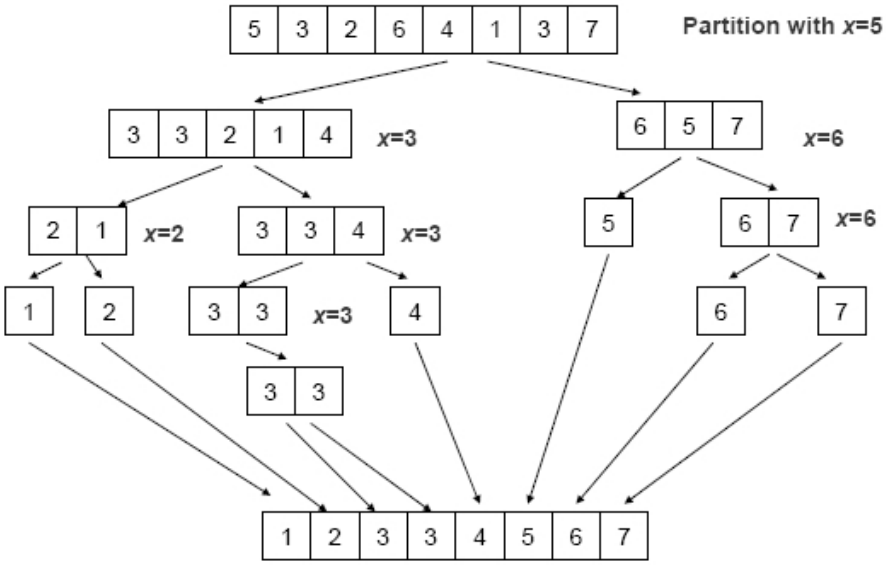
```
int partition (data *array, int left, int right, int
pivotval)
{
  do
  {
    swap (&array[left], &array[right]);
    while (value(array[left]) < pivotval)
       left++;

    while (value(array[right]) >= pivotval)
       right++;
  } while (right >= left);

/* this will be the value of the first element in the
right part */

  return (left);
}
```


The above routine cleverly moves inward from each end of the input range swapping data values that are on the wrong side of the pivot value until the two inward-moving indices meet in the middle. The initial swap, above, is not necessary; it is included only to make the do-loop more simple. In the worst case, ***n*-1** comparisons and **n/2** swaps are necessary to partition the data set.



Performance of Quicksort

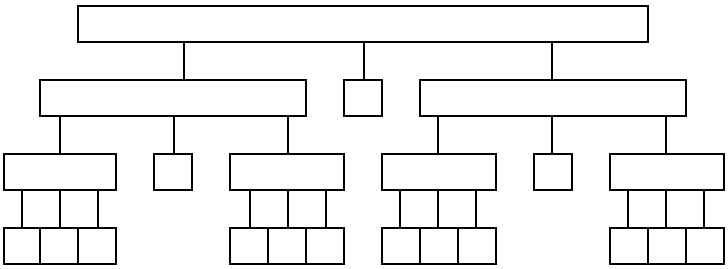
Best Case for Quicksort

Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take?

The best case for divide-and-conquer algorithms comes when we split the input as evenly as possible. Thus in the best case, each sub problem is of size $n/2$.

The partition step on each sub problem is linear in its size. Thus the total effort in partitioning the 2^k problems of size $n/2^k$ is $O(n)$.

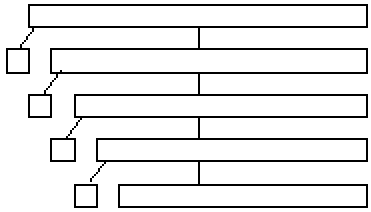
The recursion tree for the best case looks like this:



The total partitioning on each level is ***O*(*n*),** and it take **log n** levels of perfect partitions to get to single element sub problems. When we are down to single elements, the problems are sorted. Thus the total time in the best case is **T (nlog$_2$n)**.

Worst Case for Quicksort

Suppose instead our pivot element splits the array as unequally as possible. Thus instead of *n*/2 elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.



Now we have n-1 levels, instead of **log n**, for a worst case time of **T (n^2)**, since the first *n*/2 levels each have >= n/2 elements to partition.

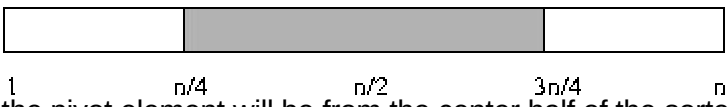
Thus the worst case time for Quicksort is worse than Heap sort or Merge sort.

To justify its name, Quicksort had better be good in the average case. Showing this requires some fairly intricate analysis.

The divide and conquer principle applies to real life. If you will break a job into pieces, it is best to make the pieces of equal size!

Intuition: The Average Case for Quicksort

Suppose we pick the pivot element at random in an array of n keys.



Half the time, the pivot element will be from the center half of the sorted array.

Whenever the pivot element is from positions *n*/4 to 3*n*/4, the larger remaining sub array contains at most 3*n*/4 elements.

Analysis

The performance of the whole Quicksort algorithm depends on how well **selectpivot** does at picking a good pivot point. The worst case for the given **selectpivot** procedure is when the data is already sorted. The higher of the first two elements in a data partition will simply divide the range into an one-element left side and an (*x*-1) element right side. This causes the Quicksort to run in quadratic time, n^2 to be precise.

Another limitation of the Quicksort is that it tends to be an excellent choice for large arrays but performs badly with very small ones.

The best and average case running efficiency of a Quicksort is **T ($nlog_2n$)** In order to Quicksort one element takes no comparisons. That is:

***T*(1) = 0**

Now, in order to Quicksort *n* elements we have to select a pivot point, partition the *n* elements, and recurs on the two partitions. Assume the *i* element is chosen to be the pivot point (where i $\in$ n). In order to partition the *n* elements will take, at most, (*n*-1) comparisons at which point the quicksort will recurs on both of the two partitions. One partition will consist of the first *i*-1elements and the other will be the contain *n*-*i* elements (assuming the pivot value itself goes with the right partition). Thus, to sort *n* items we need, at most, (*n*-1) comparisons in the partitioning phase plus however long it takes to sort each resulting partition:

***T*(*n*) = (*n*-1) + *T*(*i*-1) + *T*(*n*-*i*)**

Randomization

Suppose you are writing a sorting program, to run on data given to you by your worst enemy. Quicksort is good on average, but bad on certain worst-case instances.

If you used Quicksort, what kind of data would your enemy give you to run it on? Exactly the worst-case instance, to make you look bad.

But instead of picking the median of three or the first element as pivot, suppose you picked the pivot element at random.

Now your enemy cannot design a worst-case instance to give to you, because no matter which data they give you, you would have the same probability of picking a good pivot!

Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:

``With high probability, randomized quicksort runs in **T (n log n)** time.''

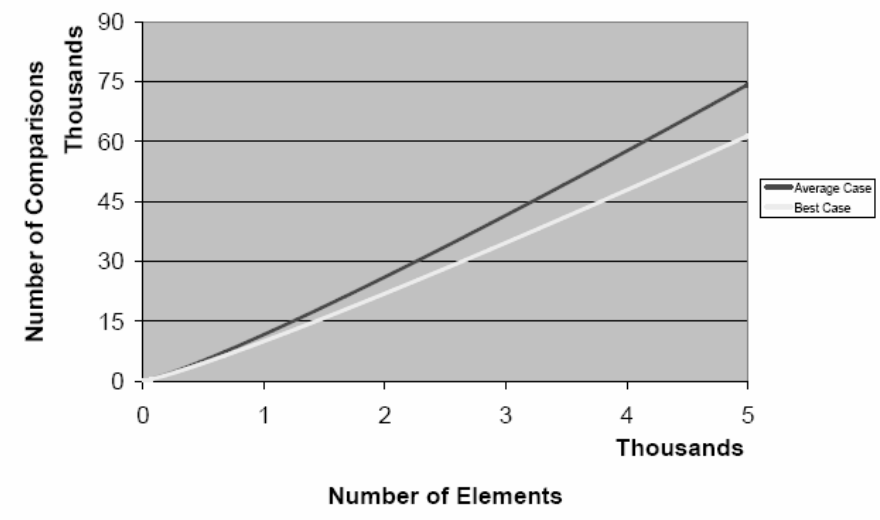
Where before, all we could say is:

``If you give me random input data, quicksort runs in expected **T (n log n)** time.''

Since the time bound how does not depend upon your input distribution, this means that unless we are extremely unlucky (as opposed to ill prepared or unpopular) we will certainly get good performance.

Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

Complexity of Quick Sort



Improvement Strategies

Some improvements can be made to the Quicksort by bolstering its weaknesses. The Quicksort does not perform well for small data sets. While your first instinct may be to ignore this limitation because you are ``always going to be sorting large arrays'' you should be remember that, as the algorithm divides your large array into smaller partitions, eventually it will reach a point that is sorting a small array. The performance of the Quicksort can be enhanced if, for small partitions, it does not call itself recursively. Rather, calling another sort algorithm to handle the small data set can substantially decrease running time.

Another way to accomplish this same optimization is to just stop sorting when your partitions reach a certain (small) size. Your final product will be a data set that is in ``almost'' sorted order. A few data items will be out of place here and there but never by much. Such an ``almost sorted'' array can then be fed through an Insertion Sort and put into final order. Because the Insertion Sort runs in near linear time for ``almost sorted'' data, this last step will normally prove to be much faster than continuing to sort every little partition recursively with a Quicksort.

Yet another Quicksort improvement strategy which pertains especially to recursive implementations of the algorithm is to always process the smallest partition first. This results in more efficient use of the call stack and overall faster execution.

Source Code in C.

```
/* Here is a full implementation of un optimized
recursive Quicksort in C.  */

/* get_pivot - return the index of the selected pivot
value */

int get_pivot (int low, int hi) {

/* safety net, this should not happen */

  if (low == hi) return(data[low]);

/* return the greater of the first two items in the
range  */

  return( (data[low] > data[low+1]) ? low : (low+1));

}
```

```
/* swap - given two pointers to integers, swap their
contents */

void swap (int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  num_swaps++;
}

/* q_sort - Quicksort a data range */

void q_sort (int low, int hi) {

  int pivot_index;
/* index in the data set of the pivot */

  int pivot_value;
/* the value of the pivot element     */

  int left, right;

/* select the pivot element and remember its value */

  pivot_index = get_pivot(low, hi);
  pivot_value = data[pivot_index];

  /* do the partitioning */

  left = low; right = hi;

  do {

  /* move left to the right bypassing elements already
on the correct side */

    while ((left <= hi) && (data[left] < pivot_value))
{
      num_comps++;
      left++;
    }

    num_comps++;

  /* move right to the left bypassing elements already
on the correct side */
```

```
    while ((right >= low) && (pivot_value <
data[right])) {
      num_comps++;
      right--;
    }
    num_comps++;

   /* if the pointers are in the correct order then
they are pointing to two items that are on the wrong
side of the pivot value, swap them... */

    if (left <= right) {
      swap(&data[left], &data[right]);
      left++;
      right--;
    }

  } while (left <= right);

  /* now recurse on both partitions as long as they
are large enough */

  if (low < right) q_sort(low, right);
  if (left < hi) q_sort(left, hi);
}
```

Bibliography

- http://www.**fearme**.com
- http://www.**daniweb**.com
- http://www.**cs.sunysb**.edu