Sorting Considerations

We consider sorting a list of records, either into ascending or descending order, based upon the value of some field of the record we will call the sort key.

The list may be contiguous and randomly accessible (e.g., an array), or it may be dispersed and only sequentially accessible (e.g., a linked list). The same logic applies in both cases, although implementation details will differ.

When analyzing the performance of various sorting algorithms we will generally consider two factors:

- the number of sort key comparisons that are required
- the number of times records in the list must be moved

Both worst-case and average-case performance is significant.

Internal or External?

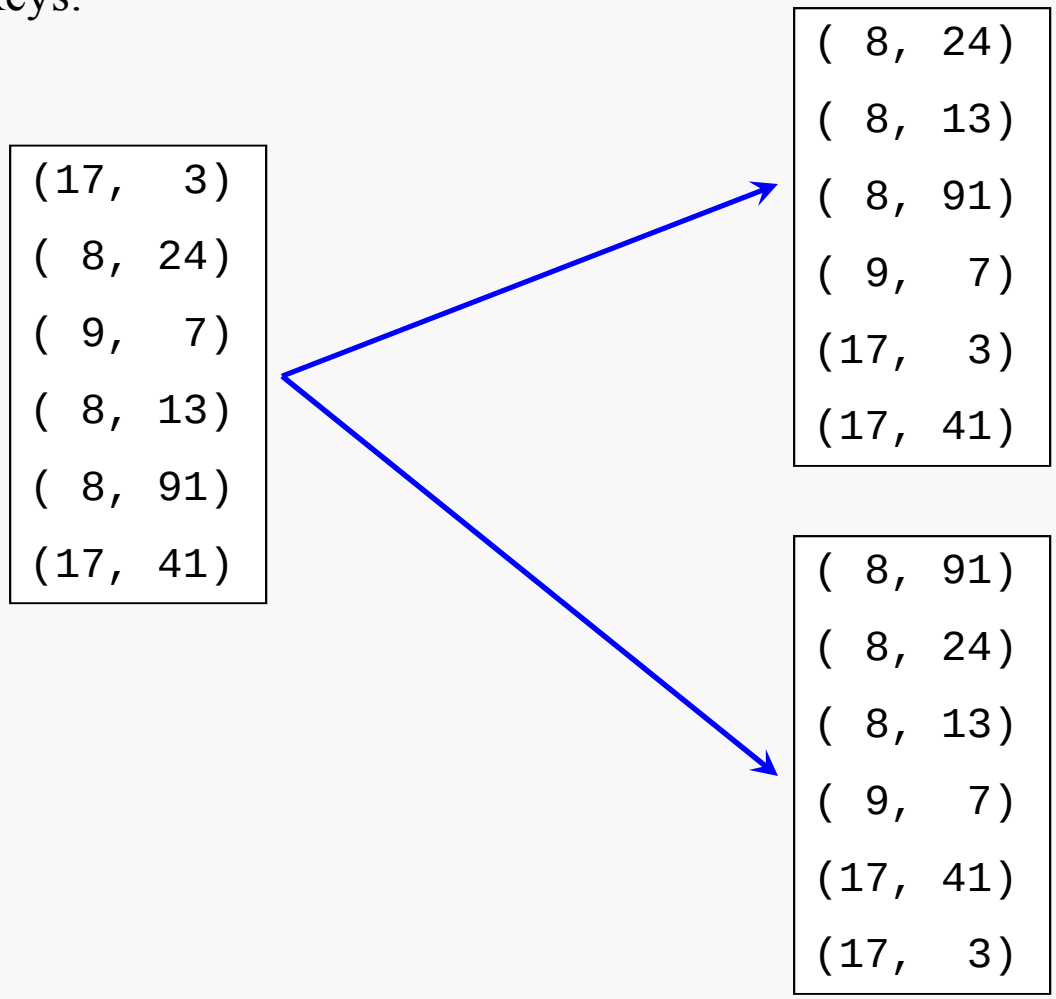
In an internal sort, the list of records is small enough to be maintained entirely in physical memory for the duration of the sort.

In an external sort, the list of records will not fit entirely into physical memory at once. In that case, the records are kept in disk files and only a selection of them are resident in physical memory at any given time.

We will consider only internal sorting at this time.

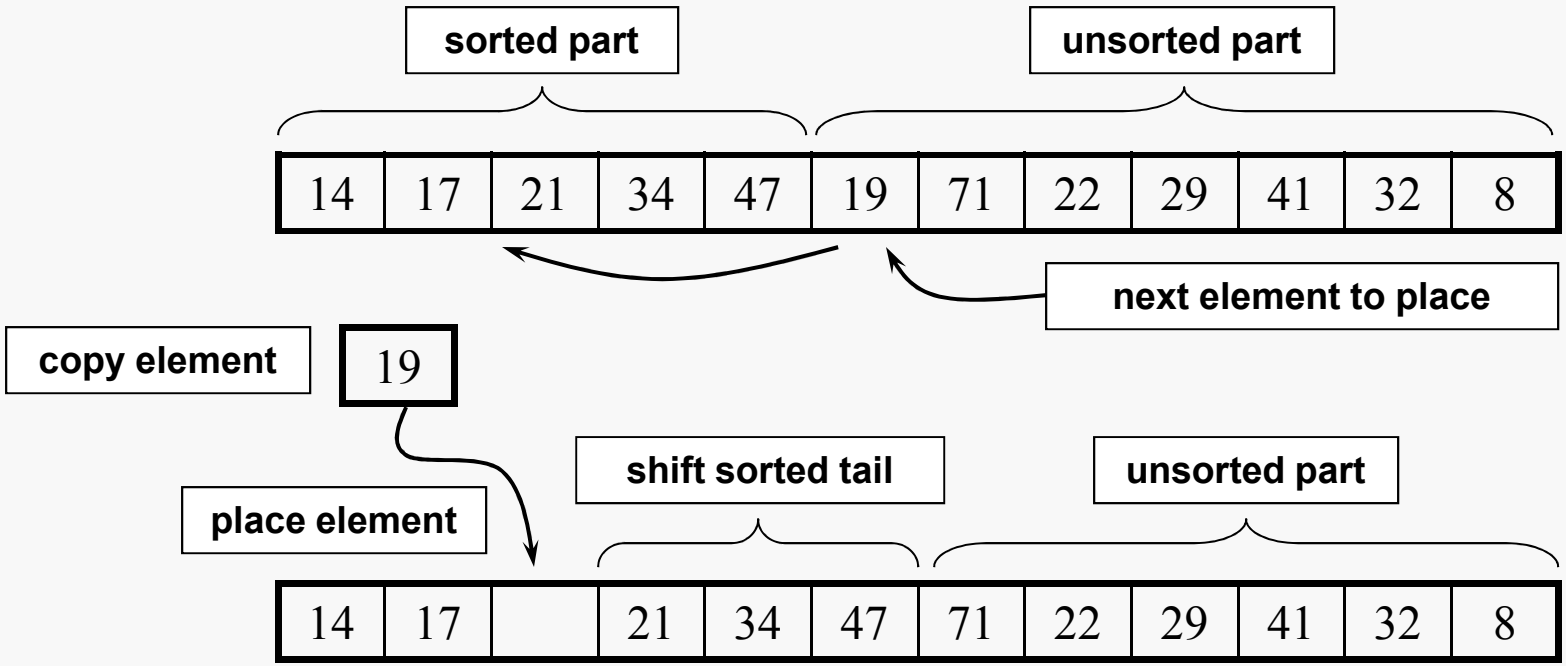
Stable or Not?

A sorting algorithm is stable if it maintains the relative ordering of records that have equal keys:



Insertion Sort

Insertion Sort:



Insertion sort closely resembles the insertion function for a sorted list.

For a contiguous list, the primary costs are the comparisons to determine which part of the sorted portion must be shifted, and the assignments needed to accomplish that shifting of the sorted tail.

Insertion Sort Implementation


```
public void insertionSort(T[] A) {

   int j;
   for (int p = 1; p < A.length; p++) {

      T temp = A[p];

      for (j = p; j > 0 && temp.compareTo(A[j-1]) < 0; j--)
         A[j] = A[j-1];

      A[j] = temp;
   }
}
```

Origin: ancient

Stable? Y

Improving Insertion Sort

Insertion sort is most efficient when the initial position of each list element is fairly close to its final position (take another look at the analysis).

Consider:

10	8	6	20	4	3	22	1	0	15	16

Pick a step size (5 here) and logically break the list into parts.

10					3					16
	8					22				
		6					1			
			20					0		
				4					15	

Sort the elements in each part. Insertion Sort is acceptable since it's efficient on short lists.

3					10					16
	8					22				
		1					6			
			0					20		
				4					15	

Improving Insertion Sort

This gives us:

3	8	1	0	4	10	22	6	20	15	16

Now we pick a smaller increment, 3 here, and repeat the process:

Partition list:

3			0			22			15	
	8			4			6			16
		1			10			20		

Sort the parts:

0			3			15			22	
	4			6			8			16
		1			10			20		

Giving:

0	4	1	3	6	10	15	8	20	22	16

Finally repeat the process with step size 1:

0	1	3	4	6	8	10	15	16	20	22

Why Is This Better?

Well…

- Until the last pass, the sublists that are being sorted are much shorter than the entire list — sorting two lists of length 1000 is faster than sorting one list of length 2000 .
- Since the sublists exhibit mixing as we change the step size, the effect of the early passes is to move each element closer to its final position in the fully sorted list.
- In the last pass, most of the elements are probably not too far from their final positions, and that's one situation where Insertion Sort is quite efficient.

QTP: Suppose than a sorting algorithm is, on average, $\Theta(N^2)$. Using that face, how would the expected time to sort a list of length 50 compare to the time required to sort a list of length 100?

Shell Sort

The process just shown is known as Shell Sort (after its originator: Donald Shell, 1959), and may be summed up as follows:

- partition the list into discontiguous sublists whose elements are some step size, *h*, apart.
- sort each sublist by applying Insertion Sort (or …)
- decrease the value of *h* and repeat the first two steps, stopping after a pass in which *h* is 1.

The end result will be a sorted list because the final pass (with *h* being 1) is simply an application of Insertion Sort to the entire list.

What is an optimal sequence of values for the step size *h*? No one knows…

Origin: 1959

Stable? N

Shell Sort Implementation


```
public void shellSort(T[] A) {

   int increment = A.length / 2;
   while (increment > 0) {

      for (int i = 0; i < A.length; i++) {
         int j = i;
         T temp = A[i];
         while ((j >= increment) &&
                (A[j-increment].compareTo(temp) > 0)) {
            A[j] = A[j - increment];
            j = j - increment;
         }
         A[j] = temp;
      }

      if (increment == 2)
         increment = 1;
      else
         increment = increment * 5 / 11;
   }
}
```

Divide and Conquer Sorting

Shell Sort represents a "divide-and-conquer" approach to the problem.

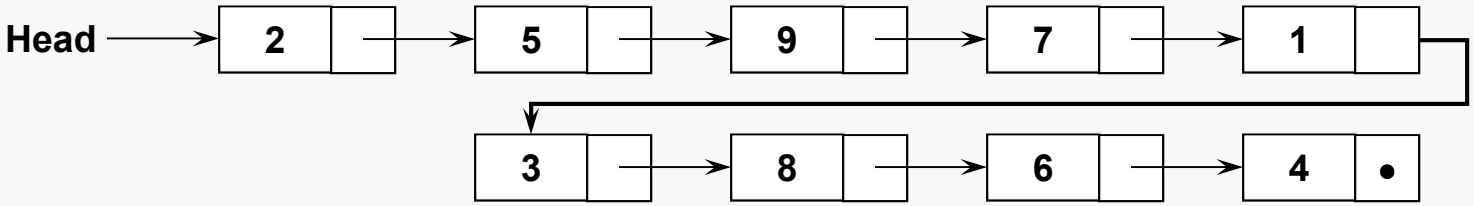
That is, we break a large problem into smaller parts (which are presumably more manageable), handle each part, and then somehow recombine the separate results to achieve a final solution.

In Shell Sort, the recombination is achieved by decreasing the step size to 1, and physically keeping the sublists within the original list structure. Note that Shell Sort is better suited to a contiguous list than a linked list.

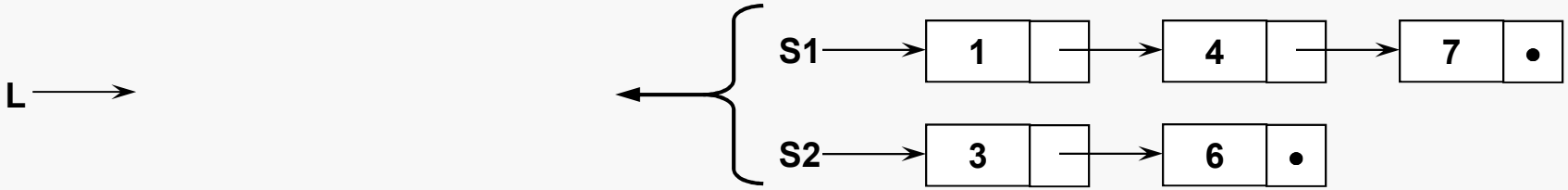
We will now consider a somewhat similar algorithm that retains the divide and conquer strategy, but which is better suited to linked lists.

Merge Sort

In Merge Sort, we chop the list into two (or more) sublists which are as nearly equal in size as we can achieve, sort each sublist separately, and then carefully merge the resulting sorted sublists to achieve a final sorting of the original list.



Merging two sorted lists into a single sorted list is relatively trivial:



Origin: J von Neumann 1945 (?)

Stable? Y

Heap Sort

A list can be sorted by first building it into a heap, and then iteratively deleting the root node from the heap until the heap is empty. If the deleted roots are stored in reverse order in an array they will be sorted in ascending order (if a max heap is used).

```
public static void heapSort(Integer[] List, int Sz) {

   BinaryHeap<Integer> toSort = new BinaryHeap<Integer>(List, Sz);

   int Idx = Sz - 1;
   while ( !toSort.isEmpty() ) {
      List[Idx] = toSort.deleteMax();
      Idx--;
   }
}
```

Origin: JWJ Williams, 1964

Stable? Y

QuickSort

QuickSort is conceptually similar to MergeSort in that it is a divide-and-conquer approach, involving successive partitioning of the list to be sorted.

The differences lie in the manner in which the partitioning is done, and that the sublists are maintained in proper relative order so no merging is required.

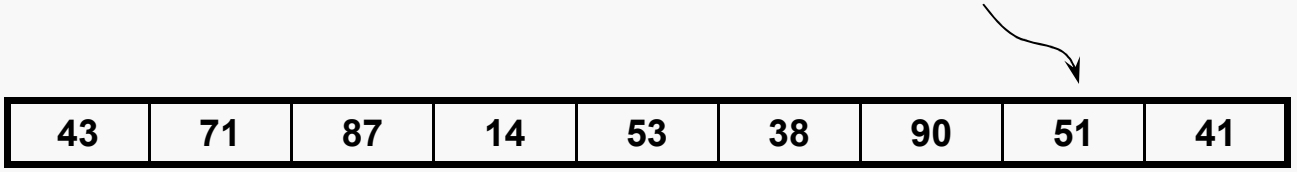
QuickSort is a naturally recursive algorithm, each step of which involves:

- pick a pivot value for the current sublist
- partition the sublist into two sublists, the left containing only values less than the pivot value and the right containing only values greater than or equal to the pivot value
- recursively process each of the resulting sublists

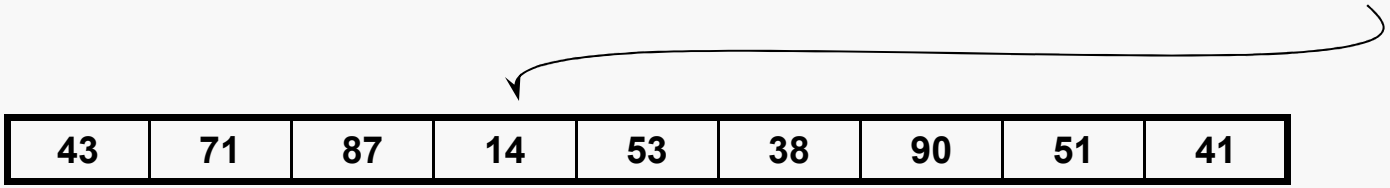
As they say, the devil is in the details…

Importance of the Pivot Value

The choice of the pivot value is crucial to the performance of QuickSort. Ideally, the partitioning step produces two equal sublists, as here:



In the worst case, the partitioning step produces one empty sublist, as here:



Theoretically, the ideal pivot value is the median of the values in the sublist; unfortunately, finding the median is too expensive to be practical here.

Choosing the Pivot Value

Commonly used alternatives to finding the median are:

- take the value at some fixed position (first, middle-most, last, etc.)
- take the median value among the first, last and middle-most values
- find three distinct values and take the median of those

The third does not guarantee good performance, but is the best of the listed strategies since it is the only one that guarantees two nonempty sublists. (Of course, if you can't find three distinct values, this doesn't work, but in that case the current sublist doesn't require any fancy sorting — a quick swap will finish it off efficiently.)

QTP: under what conditions would choosing the first value produce consistently terrible partitions?

Each of the given strategies for finding the pivot is $\Theta(1)$ in comparisons.

Partitioning a Sublist Efficiently

Since each iteration of QuickSort requires partitioning a sublist, this must be done efficiently. Fortunately, there is a simple algorithm for partitioning a sublist of N elements that is Θ(N) in comparisons and assignments:

```
template <typename T>
unsigned int Partition(T List[], unsigned int Lo, unsigned int Hi ) {

   T Pivot;
   unsigned int Idx,
                LastPreceder;
   Swap(List[Lo], List[(Lo + Hi)/2]);   // take middle-most element as Pivot
   Pivot = List[Lo];                    // move it to the front of the list
   LastPreceder = Lo;

   for (Idx = Lo + 1; Idx <= Hi; Idx++) {
      if (List[Idx] < Pivot) {
         LastPreceder++;
         Swap(List[LastPreceder], List[Idx]);
      }
   }
   Swap(List[Lo], List[LastPreceder]);

   return LastPreceder;
}
```

QuickSort Implementation

Assuming the pivot and partition function just described, the main QuickSort function is quite trivial:

```
template <typename T>
void QuickSort(T List[], unsigned int Lo, unsigned int Hi) {

   QuickSortHelper(List, Lo, Hi);
}

template <typename T>
void QuickSortHelper(T List[], unsigned int Lo, unsigned int Hi) {

   unsigned int PivotIndex;
   if (Lo < Hi) {
      PivotIndex = Partition(List, Lo, Hi);
      QuickSortHelper(List, Lo, PivotIndex - 1); // recurse on lower part,
      QuickSortHelper(List, PivotIndex + 1, Hi); //    then on higher part
   }
}
```

QuickSort Improvements

QuickSort can be improved by

- switching to a nonrecursive sort algorithm on sublists that are relatively short. The common rule of thumb is to switch to insertion sort if the sublist contains fewer than 10 elements.
- eliminating the recursion altogether. However, the resulting implementation is considerably more difficult to understand (and therefore to verify) than the recursive version.
- making a more careful choice of the pivot value than was done in the given implementation.
- improving the partition algorithm to reduce the number of swaps. It is possible to reduce the number of swaps during partitioning to about 1/3 of those used by the given implementation.

Bin Sort

Assume we need to sort a list of integers in the range 0-99:

34	16	83	76	40	72	38	80	89	87

Given an integer array of dimension 100 (the bins) for storage, make a pass through the list of integers, and place each into the bin that matches its value:

```
Bin[Souce[k]] = Source[k]
```

This will take one pass, requiring $\Theta(N)$ work, much better than N log(N).

Limitations of Bin Sort:

- the number of bins is determined by the range of the values
- only suitable for integer-like values

LSD Radix Sort

Assume we need to sort a list of integers in the range 0-99:

34	16	83	76	40	72	38	80	89	87

Given an array of 10 linked lists (bins) for storage, make a pass through the list of integers, and place each into the bin that matches its 1's digit.

Then, make a second pass, taking each bin in order, and append each integer to the bin that matches its 10's digit.

Bin		
0:	40	80
1:		
2:	72	
3:	83	
4:	34	
5:		
6:	16	76
7:	87	
8:	38	
9:	89	

Bin				
0:				
1:	16			
2:				
3:	34	38		
4:	40			
5:				
6:				
7:	72	76		
8:	80	83	87	89
9:				

Origin: Hollerith 1887 (?)

Stable? N

References

Williams JWJ. *Algorithm 232 - Heapsort*, 1964, Comm. ACM 7(6): 347–348.

Hoare, C. A. R. "Partition: Algorithm 63," "Quicksort: Algorithm 64," and "Find: Algorithm 65." Comm. ACM 4(7), 321-322, 1961