CLASS NOTES, CS W3137 - SORTING

1 Overview of Sorting

1. There are many different sorting methods available, and which one you choose to use is usually a function of the data you are trying to sort.
2. There are some interesting rules of thumb about sorting that should be kept in mind.
3. First, if you are going to be sorting a small data set, any sort will work fine. You only pay the price of larger sorting complexity for large values of N.
4. The general rule of thumb is that the best sorting time is $O(NLogN)$.

2 Decison Tree Proof of O(N Log N) Lower Bound on Sorting

(see Weiss, p. 303) An intuitive proof of the $O(NLogN)$ lower bound on sorting an arbitrary number of items is this: any sorting algorithm has a choice of $N!$ orderings for N elements. If we place these in a binary comparison tree we see that we will need a tree with $N!$ leaves, where each leaf represents a possible sorting order. The minimum size tree we need is the smallest binary tree with $N!$ leaves. If we think of the path length of a full binary tree as K, then we know there are $2^K = N$ leaves in such a tree. Turning this around, if we have $N!$ leaves, what is the path length, or equivalently, what is the path length K for a tree with $2^K = N!$ leaves. It turns out that such a tree has a path length of $K \sim O(NLogN)$ (see proof, p. 303), and path length (from the root to a leaf) is the number of operations (comparisons) we need to create a sorting order.

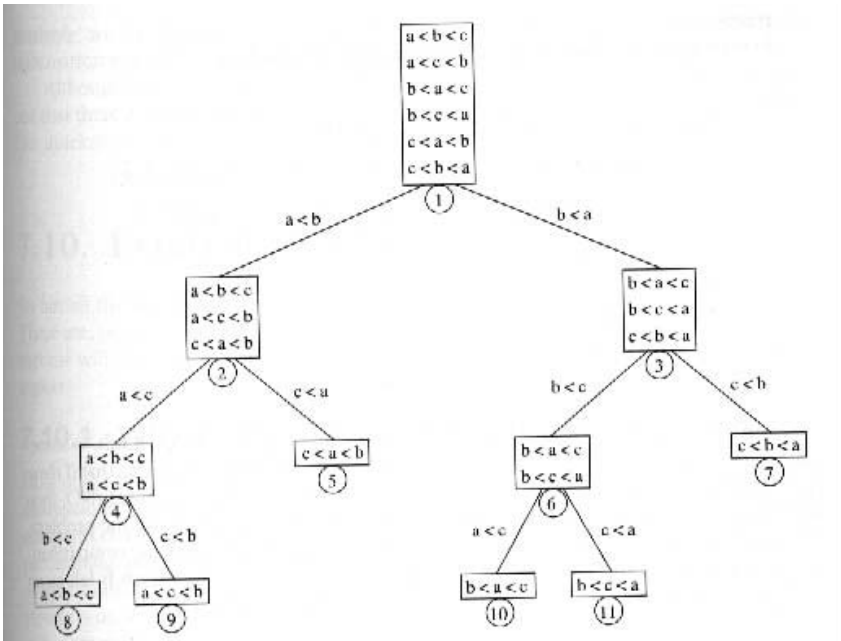


Figure 7.17 A decision tree for three-element insertion sort

3 Bubble Sort

The idea in bubblesort is to have the largest ("heaviest") item sink to the bottom of the array. Once there, we repeat the procedure for an array of one less item, and it only has to sink to the next to the last position (you can also float the smallest ("lightest") element to the top of the array which is where the name comes from). For N elements, you need $N-1$ passes to perform bubblesort. After the $N-1$st pass, all items are in their place but one, and that last item has nowhere to go but its correct position. So $N-1$ passes are sufficient. The cost of bubble sort is $O(N^2)$. This can be seen since the $N-1$ passes each has to perform $N-1, N-2, N-3, ...1$ comparisons. Summing this up we get:

$$\sum_{i=1}^{N-1} i \;=\; \frac{(N-1)\cdot N}{2} \;=\; O(N^2) \tag{1}$$

```
Original list: 29 17 33 39 72  9 42 63 81  2
Pass 0
 17 29 33 39  9 42 63 72  2 81
Pass 1
 17 29 33  9 39 42 63  2 72 81
Pass 2
 17 29  9 33 39 42  2 63 72 81
Pass 3
 17  9 29 33 39  2 42 63 72 81
Pass 4
  9 17 29 33  2 39 42 63 72 81
Pass 5
  9 17 29  2 33 39 42 63 72 81
Pass 6
  9 17  2 29 33 39 42 63 72 81
Pass 7
  9  2 17 29 33 39 42 63 72 81
Pass 8
  2  9 17 29 33 39 42 63 72 81
```

An interesting feature of bubblesort is that the comparison section of the algorithm can be used to speed up the sort if the array becomes sorted during any of the passes. If the array becomes sorted, it will not switch any elements. So we can put a boolean flag in the code to test for this, and exit the sort early if the array becomes sorted.

4 Selection Sort

Selection sort works by finding the smallest (or largest) element in an array, and placing that element in the first (or last) position of the array. It then repeats this procedure on the remaining elements - you can think of it doing the same thing on an array that is 1 element smaller.

The running time of selection sort is $O(N^2)$. At each successive pass we have to make $N-1\,, N-2\,, N-3\,, \cdots$ comparisons, which sums up to $\frac{(N-1)\times(N-2)}{2}$ or $O(N^2)$.

```
/* find smallest, put at end of array, repeat  */
void SelectionSort(InputArray A) {
      int  MinPosition, temp, i, j;
      for (i = n - 1; i > 0; --i) {
         MinPosition = i;
         for( j = 0; j < i; ++j) {
            if (A[j] < A[MinPosition]) {
               MinPosition = j;
             }
         }
         temp = A[i];
         A[i] = A[MinPosition];
         A[MinPosition] = temp;
      }
}
```

5 Insertion Sort

Insertion sort is similar to Selection Sort. For pass $P = 1$ through $N - 1$ we make sure that elements in positions 0 through P are already sorted. Page 272 has a sample run of insertion sort.

```
 import java.util.Random;
    public class insertionsort  {
      insertionsort(){}
      public void  insertionSort( Integer [ ] a ) {
      int j;
      for( int p = 1; p < a.length; p++ ) {
         Integer tmp = a[ p ];
         for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- ) {
            a[ j ] = a[ j - 1 ];
            }
          a[ j ] = tmp;
          System.out.println("after pass " + p);
          printArray(a);
     }
    }
    public void printArray(Integer[] a) {
        int i,j;
        for(i=0;i<a.length;i++){
            System.out.print(a[i] + " ");
            }
        System.out.println();
    }
    public static void main( String [ ] args ) {
           int NUM_ITEMS=8;
           Integer [ ] a = new Integer[NUM_ITEMS];
           Random generator= new Random();
           insertionsort Sort = new insertionsort();
           if(args.length==0){
              for( int i = 0; i < a.length; i++ )
                 a[ i ] = (new Integer(generator.nextInt(100)));
           } else for(int i=NUM_ITEMS-1;i>=0;i--)
       a[i]=new Integer(NUM_ITEMS-i);
           System.out.println("original array");
           Sort.printArray(a);
           Sort.insertionSort( a );
   }
}
-------------------------------------------------------
original array
69 28 15 11 80 70 87 46
after pass 1
28 69 15 11 80 70 87 46
after pass 2
15 28 69 11 80 70 87 46
after pass 3
11 15 28 69 80 70 87 46
after pass 4
11 15 28 69 80 70 87 46
after pass 5
11 15 28 69 70 80 87 46
after pass 6
11 15 28 69 70 80 87 46
after pass 7
11 15 28 46 69 70 80 87
```

Insertion sort is a *stable sort*. This means that if two records have equal keys, the original order of the elements is preserved. For example, if we have a sorted list of students by name and year of graduation (sorted by name), and then sort on graduation year, the names are still alphabetical within graduation year.

Insertion sort is somewhat *adaptive* in that it only moves as many elements as it needs to find the right insertion point. It is a good choice if the array is "almost" sorted, as most elements only move a little bit.

Note that if you have records that are VERY large, it doesn't make sense to move the entire record whne you make a positional swap. A better way is to just associate pointers with each record and just swap the pointers so they point to the correct record. This is somewhat language dependent. Also note the at selection sort only does 1 swap at the end of each iteration, so it may be a better choice if large record swaps are required.

6 Heapsort

1. A simple sorting technique that takes advantage of the heap idea is *Heapsort*. Essentially, to sort N items, $A_1 \ldots A_n$, you do the following:

```
for(i=1; i<=N; i++)
   insert item i into Heap

for(i=1; i<=N; i++)
   Deletemin from Heap and readjust Heap
```

2. To analyze the Heapsort algorithm, we create a heap from an unordered array at cost N (see p. 235 of Weiss), and once we create the heap, we need to perform N deletions at cost $Log(N)$ for each deletion. Remember, once you delete an item from the heap (constrant cost) you need to reform the heap at cost $Log(N)$ (you do a percolate down of the last element in the heap) yielding a total cost of $N + N\,Log(N)$ which is $O(N\,Log\,N)$.

3. Another speedup we can do is that we can use the array holding the heap to actually store the result instead of having to use a seondary array to hold the sorted list. We simply put each deletemin() item at the newly vacated last position of the array after we do the reheap operation. So the first item in the heap array (the minimum) will actually be the last item in the array when we are done. This performs a decreasing order sort. If we desire an increasing order sort, we simply have to build a max-heap, and we will perform successive delete-max() operations which will put the largest element last in the array, and create an increasing order sort.

7 Linear Time Sorting?

However, if we know something about the data, we can actually sort in *linear time*. The program below sorts 4 digit positive integers by simply incrementing an array slot indexed by the number if the number is present in the sequence. It then prints out the array elements that are marked from lowest to highest. If the increment is more than 1, then duplicates were in the sequence which are easily handled. Cost of the algorithm is $O(M + N)$ where M is number of elements to be sorted and N is the largest value of the integers to be sorted

```
//  Linear time sort of a file containing 4 digit integers:
import java.io.*;
public class LinearSort {
   public static  void main(String[] args) {
      int i,j,code;
      int[] A= new int[10000]; /* holds all 4 digit ints */
      BufferedReader reader;
      String line;
      try{
         reader= new BufferedReader(new InputStreamReader(System.in));
         for(i=0;i<10000;i++) A[i]=0; // initialize array to 0
         while((line = reader.readLine())!=null) {
            int number = Integer.parseInt(line);
            A[number]++;
         }
      }
      catch(IOException e)
      {
         System.out.println("exception " + e);
      }
      for(i=0;i<10000;i++) //print out sorted numbers
         if(A[i]>0) System.out.println(i);
  }
}
```

8 Radix Sort

When sorting data that is organized by digits, we can do a multipass sort that "buckets" each record according to successive digits of the key, beginning at the Least Significant Digit (LSD -rightmost digit) and continuing to the Most Significant Digit (MSD -leftmost digit). This works like a multiway tree - at each step you can do a partition into 1 of 10 sets (assuming decimal digits), and keep partitioning for each successive digit, but using the results of the previous partitioning as you go along. Running time is O(MN), where N is the number of items and M is the number of digits in each number. This can be extended to lexical sorts easily by using the ascii characters as the successive keys.

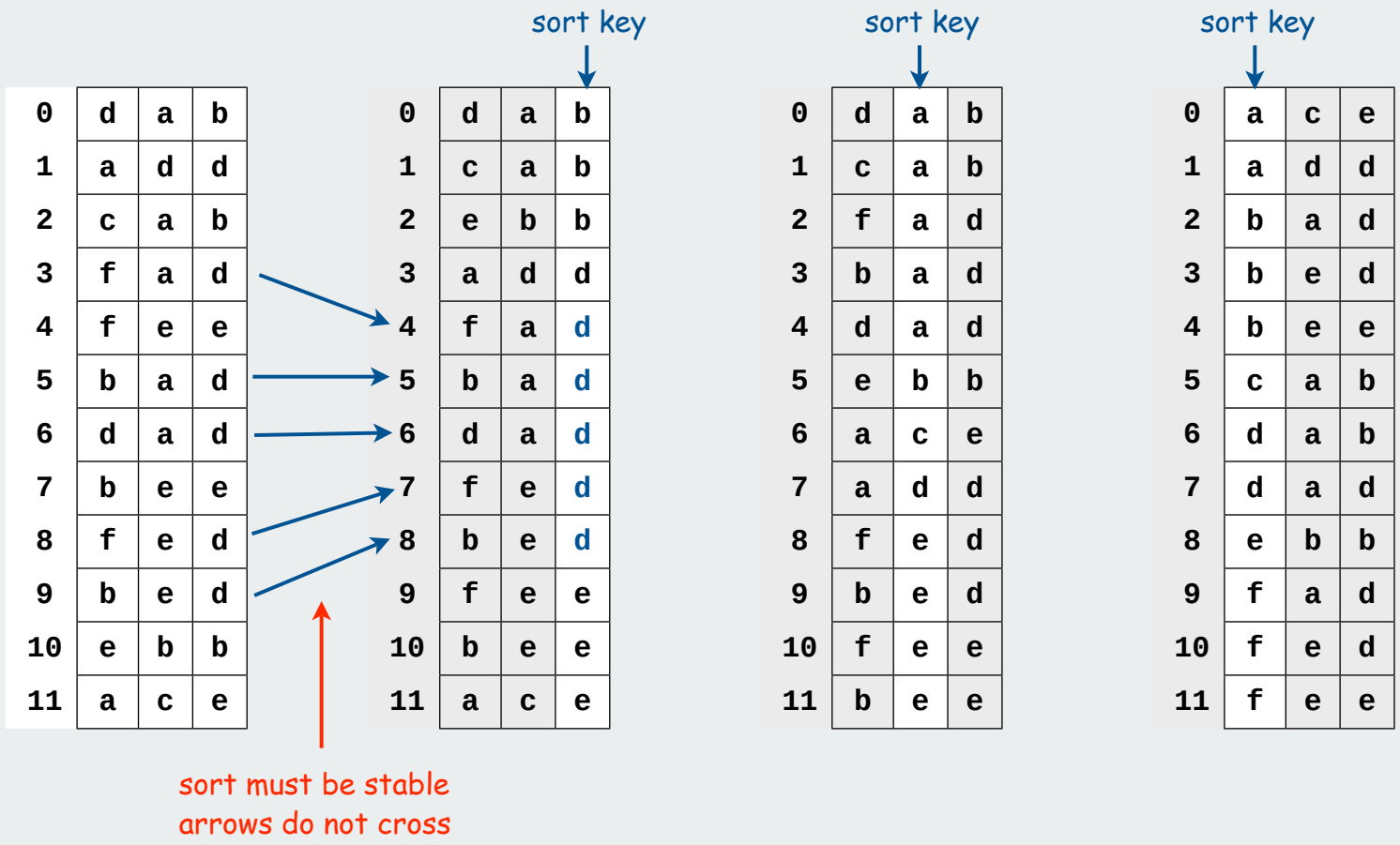
```
Initial Sequence: 970 72 879 43 485 183 984 992 311 217 363 754

Sort on first digit
List[0]: 970
List[1]: 311
List[2]: 72 992
List[3]: 43 183 363
List[4]: 984 754
List[5]: 485
List[6]:
List[7]: 217
List[8]:
List[9]: 879

Sort on second digit
List[0]:
List[1]: 311 217
List[2]:
List[3]:
List[4]: 43
List[5]: 754
List[6]: 363
List[7]: 970 72 879
List[8]: 183 984 485
List[9]: 992

Sort on third digit
List[0]: 43 72
List[1]: 183
List[2]: 217
List[3]: 311 363
List[4]: 485
List[5]:
List[6]:
List[7]: 754
List[8]: 879
List[9]: 970 984 992
Final Sequence: 43 72 183 217 311 363 485 754 879 970 984 992
```

You can use bases other than decimal as well. If you are trying to sort 1 million 64 bit integers, you can treat each 64 bit integer as 4 16 bit digits. Then you can use $2^{16} = 65,536$ buckets for your radix sort, and do the sort in 4 passes over the data, one for each 16 bit digit.

Least-significant-digit-first radix sort

LSD radix sort.

- Consider characters `d` from right to left
- Stably sort using `d`th character as the key via key-indexed counting.



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	d	a	b
1	c	a	b
2	e	b	b
3	a	d	d
4	f	a	d
5	b	a	d
6	d	a	d
7	f	e	d
8	b	e	d
9	f	e	e
10	b	e	e
11	a	c	e

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

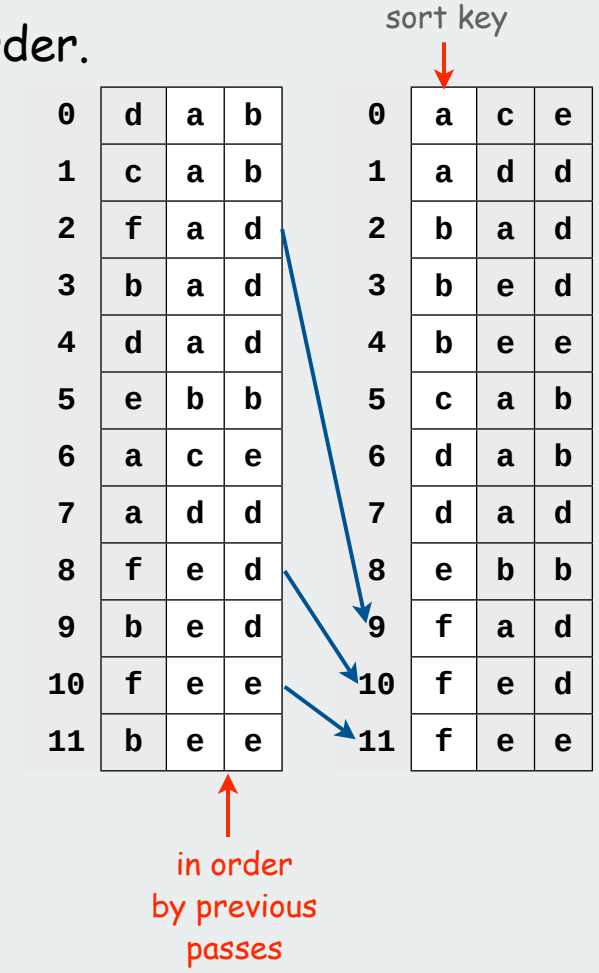
LSD radix sort: Why does it work?

Pf 1. [thinking about the past]

- If two strings differ on first character, key-indexed sort puts them in proper relative order.
- If two strings agree on first character, stability keeps them in proper relative order.

Pf 2. [thinking about the future]

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, stability ensures later pass won't affect order.



now	ace	ace	ace
for	ago	ago	ago
tip	and	and	and
ilk	bet	bet	bet
dim	cab	cab	cab
tag	caw	caw	caw
jot	cue	cue	cue
sob	dim	dim	dim
nob	dug	dug	dug
sky	egg	egg	egg
hut	for	few	fee
ace	fee	fee	few
bet	few	for	for
men	gig	gig	gig
egg	hut	hut	hut
few	ilk	ilk	ilk
jay	jam	jay	jam
owl	jay	jam	jay
joy	jot	jot	jot
rap	joy	joy	joy
gig	men	men	men
wee	now	now	nob
was	nob	nob	now
cab	owl	owl	owl
wad	rap	rap	rap
caw	sob	sky	sky
cue	sky	sob	sob
fee	tip	tag	tag
tap	tag	tap	tap
ago	tap	tar	tar
tar	tar	tip	tip
jam	wee	wad	wad
dug	was	was	was
and	wad	wee	wee

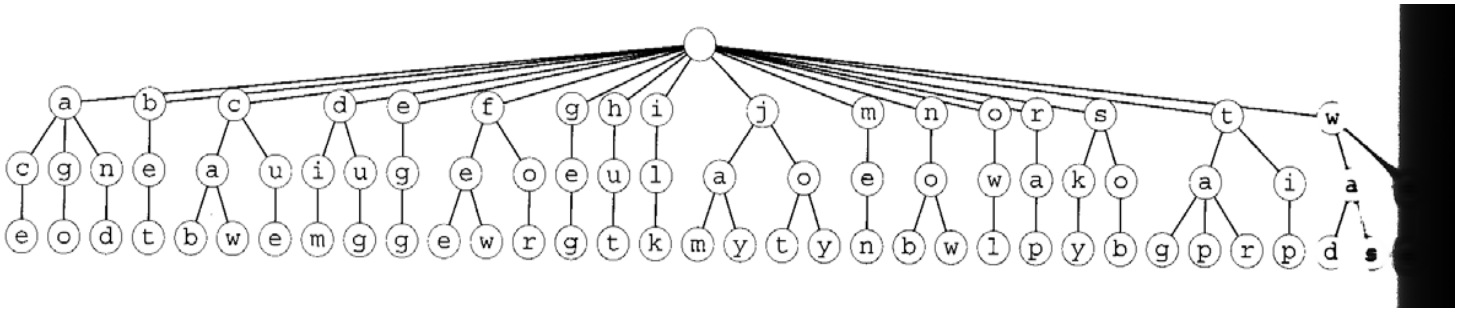
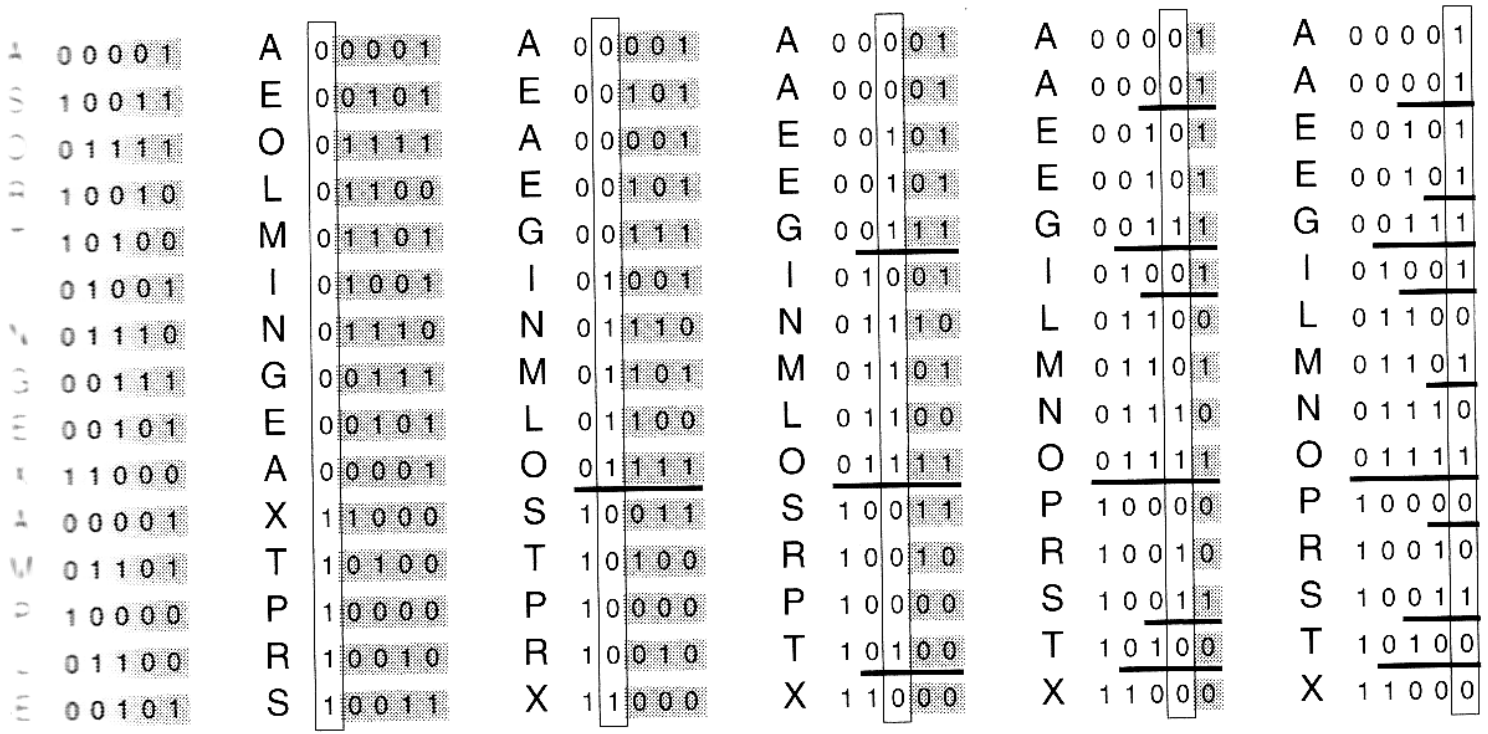


Figure 10.7
MSD radix sort example

We divide the words into 26 bins according to the first letter. Then, we sort all the bins by the same method, starting at the second letter.

MSD Radix Sort Using Radix 2 (sort by MSD bits)



9 Mergesort

1. Mergesort is an interesting algorithm that reflects a typical Computer Science strategy: *divide and conquer*.
2. The idea is very simple: To sort N items do the following: Split the array in half, forming two arrays of $\frac{N}{2}$ items each. Sort these arrays separately, and then merge the two arrays back together again.
3. The algorithm is naturally recursive, as we can keep splitting and merging the arrays of size N, $\frac{N}{2}$, $\frac{N}{4}$, , ...
4. To sort N items, we need an additional array to store the merged data, so there is an additional memory cost for mergersort.
5. Check out an animation of mergesort at:
   `http://www-2.cs.cmu.edu/~jaharris/applets/sorting/merge/mergesort.htm`

```
public static void mergeSort( Comparable [ ] a )
{
   Comparable [ ] tmpArray = new Comparable[ a.length ];
   mergesort( a, tmpArray, 0, a.length - 1 );
}


public static void mergesort( Comparable [ ] a, Comparable [ ] tmpArray,
                   int left, int right )
{
    if( left < right ) {
       int center = ( left + right ) / 2;
       mergesort( a, tmpArray, left, center );
       mergesort( a, tmpArray, center + 1, right );
       merge( a, tmpArray, left, center + 1, right );
    }
}

public static void merge( Comparable [ ] a, Comparable [ ] tmpArray,
               int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while( leftPos <= leftEnd && rightPos <= rightEnd )
       if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
          tmpArray[ tmpPos++ ] = a[ leftPos++ ];
       else
          tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    while( leftPos <= leftEnd )    // Copy rest of first half
       tmpArray[ tmpPos++ ] = a[ leftPos++ ];

    while( rightPos <= rightEnd )  // Copy rest of right half
       tmpArray[ tmpPos++ ] = a[ rightPos++ ];
    for( int i = 0; i < numElements; i++, rightEnd-- )
       a[ rightEnd ] = tmpArray[ rightEnd ]; // Copy tmpArray back
}
```

6. The time to mergesort N items is the time to do 2 recursive mergesorts of size $\frac{N}{2}$, plus the merge step (which is linear): $T(N) = 2T(\frac{N}{2}) + N$, which is $O(N \log N)$

10 Shellsort

1. We saw the problem with sorts such as insertion sort that each item can only move one position at a time to find its place. Quicksort improves on this by swapping elements across large distances relative to the choice of pivot. Shell sort is another sort that tries to allow data to move a large distance in the array. It is called a diminishing increment sort since it uses successively smaller increments to K-sort the array. A K-sorted array has all elements which are K positions apart in the correct ordering. As long as the increment K eventually reaches 1, any increment sequence is OK - although some are obviously better than others (a 1-Sorted array is fully sorted - all elements 1 position away are in the correct order).

2. For an array of N items, form sequences of length K taking every Kth item, starting from successive locations in the array. This forms N/K subsequences, that we sort using insertion sort. This will allow numbers way out of place to move large distances in the array.

3. For example, in a $N = 1000$ element array, take sequences of $K = 20$ elements (element indices: 1,21,41,61,...,981; 2,22,42,62,...,982; and so on). Sort each of these chains using insertion sort. Now, take a new set of sequences of length less than K and do the same. Eventually the length of the sequences becomes 1, and the last pass is simply insertion sort; however, the array is hopefully almost sorted at this point, reducing the effort for the entire sort. Most items have already moved large distances to be "close" to where they should be, so the final insertion sort doesn't have to do much switching of elements.

4. Its hard to prove results on Shell sort, save it can be $O(N^2)$ worst case. Its performance relies heavily on the sequencing length choice, and there is no known optimal sequence for random input. But empirically, it is a very good choice for a sort. Finally, note the simplicity of the code. This alone makes it a nice choice for a fast sorting method.

```
void Shellsort( ElementType A[ ], int N )
{int i, j, Increment;
 ElementType Tmp;
   for( Increment = N / 2; Increment > 0; Increment /= 2 )
      for( i = Increment; i < N; i++ )  {
        Tmp = A[ i ];
        for( j = i; j >= Increment; j -= Increment )
           if( Tmp < A[ j - Increment ] )
             A[ j ] = A[ j - Increment ];
           else   break;
        A[ j ] = Tmp;
      }
}
```

```
         Index:  0  1  2  3  4  5  6  7  8  9
                 ----------------------------
Unsorted items: 46 68 15 22 59 44 54 39 12 88
Increment (K) = 5
5-Sorted items: 44 54 15 12 59 46 68 39 22 88
Increment (K) = 2
2-Sorted items: 15 12 22 39 44 46 59 54 68 88
Increment (K) = 1
1-Sorted items: 12 15 22 39 44 46 54 59 68 88
```

11 Quicksort

1. Quicksort does a divide and conquer routine similar to mergesort by partioning the array at a chosen pivot element.
2. The array is quickly broken down into 2 smaller arrays with items less than the pivot element on one side, and items greater than the pivot on the other. These smaller arrays are then sorted recursively by again calling quicksort.
3. If we compare quicksort with mergesort, we can see similarities in the divide and conquer strategy. Where quicksort wins is that the array can be sorted *in place* because the smaller arrays contain the correct elements that will eventually fit in these slots. We do not need the merge step as in merge sort, which requires an additional temporary array.
4. Quicksort is only as good as its choice of pivot element. If we continually choose a pivot element that is one of the end elements of the final sorted array, we only reduce the amount of work by 1 element each time.
5. This can lead to a worst case performance of $O(N^2)$ for quicksort. However, choosing a good pivot will yield performance on the order of $O(NlogN)$.
6. To choose the pivot, a good strategy is to generate a random number and choose an element at random.
7. Another good strategy is to examine the two end elements and the center element of the array and choose the median element (a very quick procedure with 3 elements). This guarantees that you will not get an end element as a pivot (although it may be only 1 away from the end!).
8. Using the median of three partitioning, and actually sorting the three elements used in choosing the median, can help speed up the sort. You can use this "pre-sort" to reduce the number of items left to sort. If we are partioning array elements A[LEFT] to A[RIGHT], we can put the smallest of the three elements in A[LEFT] (obviously it will be less than the pivot, and to its left), the largest can be placed in A[RIGHT] (obviously larger than the pivot, and to its right), and we can start our exchanges at LEFT+1 and RIGHT-1 since the LEFT and RIGHT items are in the correct parts of the array relative to the pivot.
9. To prevent quicksort from making lots of recursive calls of small subarrays, a good strategy is to quicksort an array into partitions of a few elements (say less than 10) and then sort these items in-place using a simple sort technique like insertion sort. This cuts the overhead of many recursive subcalls.

```
sort 10 items by Quicksort - choose pivot randomly each time

Orig. Array: 65 48 76 15 88  3 50 20 44 77
Call Qsort of items 0-9,  Pivot=50
65 48 76 15 88  3 77 20 44 50
Swap 65 and 44
44 48 76 15 88  3 77 20 65 50
Swap 76 and 20
44 48 20 15 88  3 77 76 65 50
Swap 88 and 3
44 48 20 15  3 88 77 76 65 50
Restore Pivot
44 48 20 15  3 50 77 76 65 88
Call Qsort of items 0-4, Pivot=20
44 48  3 15 20 50 77 76 65 88
Swap 44 and 15
15 48  3 44 20 50 77 76 65 88
Swap 48 and 3
15  3 48 44 20 50 77 76 65 88
Restore Pivot
15  3 20 44 48 50 77 76 65 88
Call Qsort of items 0-1, Pivot=15
 3 15 20 44 48 50 77 76 65 88
Restore Pivot
 3 15 20 44 48 50 77 76 65 88
Call Qsort of items 3-4, Pivot=44
 3 15 20 48 44 50 77 76 65 88
Restore Pivot
 3 15 20 44 48 50 77 76 65 88
Call Qsort of items 6-9, Pivot=65
 3 15 20 44 48 50 77 76 88 65
Restore Pivot
 3 15 20 44 48 50 65 76 88 77
Call Qsort of items 7-9, Pivot=77
 3 15 20 44 48 50 65 76 88 77
Restore Pivot
 3 15 20 44 48 50 65 76 77 88
Sorted Array:  3 15 20 44 48 50 65 76 77 88
```

12 Improved Quicksort

This version of quicksort is optimized in 2 ways. First, the median of 3 sorting is used to reduce the computation by putting the smallest of the 3 in position LEFT and then starting the scan of items at LEFT + 1 since the LEFT item is in place correctly. The RIGHT item is placed at RIGHT-1 (Pivot is placed in RIGHT), and the scan starts now at RIGHT-2. The sceond optimization is to use an insertion sort when the indices of the quicksort differ by less than 3 (CUTOFF=3).

```
public static void quicksort( Comparable [ ] a, int left, int right )
{
   if( left + CUTOFF <= right ) {
        Comparable pivot = median3( a, left, right );
        int i = left, j = right - 1;
        for( ; ; ) {
           while( a[ ++i ].compareTo( pivot ) < 0 ) { }
           while( a[ --j ].compareTo( pivot ) > 0 ) { }
           if( i < j )
              swapReferences( a, i, j );
           else   break;
        }
        swapReferences( a, i, right - 1 );   // Restore pivot
        quicksort( a, left, i - 1 );    // Sort small elements
        quicksort( a, i + 1, right );   // Sort large elements
  }   // Do an insertion sort on the subarray
  else   insertionSort( a, left, right );
}
private static Comparable median3( Comparable [ ] a, int left, int right )
{
        int center = ( left + right ) / 2;
        if( a[ center ].compareTo( a[ left ] ) < 0 )
             swapReferences( a, left, center );
        if( a[ right ].compareTo( a[ left ] ) < 0 )
             swapReferences( a, left, right );
        if( a[ right ].compareTo( a[ center ] ) < 0 )
             swapReferences( a, center, right );
// Place pivot at position right - 1
        swapReferences( a, center, right - 1 );
        return a[ right - 1 ];
        }
```

```
Sort 10 numbers by optimized quicksort...
original array
8 1 4 9 6 3 5 2 7 0
Call Quicksort of items 0 TO 9
Pivot is 6
       before swap..
       0 1 4 9 7 3 5 2 6 8
       after swap..
       0 1 4 2 7 3 5 9 6 8
       before swap..
       0 1 4 2 7 3 5 9 6 8
       after swap..
       0 1 4 2 5 3 7 9 6 8
    before pivot replace
    0 1 4 2 5 3 7 9 6 8
    after pivot replace
    0 1 4 2 5 3 6 9 7 8
Call Quicksort of items 0 TO 5
Pivot is 3
       before swap..
       0 1 5 2 3 4 6 9 7 8
       after swap..
       0 1 2 5 3 4 6 9 7 8
    before pivot replace
    0 1 2 5 3 4 6 9 7 8
    after pivot replace
    0 1 2 3 5 4 6 9 7 8
Call Quicksort of items 0 TO 2
    before INSERTION SORT..
    0 1 2 3 5 4 6 9 7 8
    after INSERTION SORT
    0 1 2 3 5 4 6 9 7 8
Call Quicksort of items 4 TO 5
    before INSERTION SORT..
    0 1 2 3 5 4 6 9 7 8
    after INSERTION SORT
    0 1 2 3 4 5 6 9 7 8
Call Quicksort of items 7 TO 9
    before INSERTION SORT..
    0 1 2 3 4 5 6 9 7 8
    after INSERTION SORT
    0 1 2 3 4 5 6 7 8 9
```