CS 3410 – Ch 8 – Sorting

Sections	Pages
8.1-8.8	351-384

8.0 – Overview of Sorting

Name	Best	Average	Worst	Memory	Stable	Method	Class
Insertion sort	n	n^2	n^2	1	Yes	Insertion	CS 1301
Shell sort	n	$n\log^2 n$ or $n^{3/2}$	—	1	No	Insertion	CS 3410
Binary tree sort	—	$n\log n$	$n\log n$	n	Yes	Insertion	CS 3410
Selection sort	n^2	n^2	n^2	1	No	Selection	CS 1301
Heapsort	$n\log n$	$n\log n$	$n\log n$	1	No	Selection	CS 3410
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	CS 1302
Merge sort	$n\log n$	$n\log n$	$n\log n$	Depends	Yes	Merging	CS 1302
Quicksort	$n\log n$	$n\log n$	n^2	$\log n$	Depends	Partitioning	CS 1302

8.0 – Pancake Sort

1. Consider a stack of pancakes, all different sizes, in no particular order. You have a spatula that you can slide underneath a pancake and *flip* the pancakes on top of it. Your goal is to flip the pancakes until you have ordered them so that the largest is on the bottom, next-largest is next-to-last, ..., smallest on top. What strategy for flipping would you use? How many flips would it take? Could you implement a *flip* on a computer?

2. **Pancake Sort** – Pancakes are arranged in a single stack on a counter. Each pancake is a different size than the others. Using a spatula, how would you rearrange the pancakes so that they are ordered, smallest to largest

   Algorithm:

   a. Find the largest pancake
   b. Stick the spatula in beneath it, flip it and those above (largest is on top now).
   c. Stick the spatula under the bottom and flip all the pancakes (largest is now on bottom)

      *Now, we will say that the largest pancake is on a *stack*.

   d. Find the next largest pancake
   e. Stick the spatula in beneath it, flip it and those above (next largest is on top now).
   f. Stick the spatula under the next-to-bottom and flip all the pancakes (next largest is now next-to-bottom, *e.g.* on the stack)

   g. *etc*.

   You can try this out at: http://www.cut-the-knot.org/SimpleGames/Flipper.shtml

3. What is the complexity of this algorithm? If we decide to compute complexity in terms of *flips, i*t can be shown that this algorithm takes at most $2n - 3$ flips, $O(n)$.

4. The fastest pancake flipping algorithms take between $\frac{15}{14}n$ and $\frac{5}{3}n$.

5. How can we have a sorting algorithm that is linear? We said earlier in the semester that in worst case, sorting algorithms, at best are $O(n\log n)$. The answer is that the pancake sorting algorithms are computing complexity based on *flips*. The sorting algorithms we have discussed before are counting *swaps* or *comparisons*.

6. But, this discussion is still useful. It leads us to ask how such a pancake sorting algorithm could be implemented in linear time. To do so, it would have to be able to: (a) find the largest element (pancake) in constant time, (b) *flip* a range of array/linked-list elements.

7. Notice that Pancake sorting is similar to Selection sort:

   a. Find the min
   b. Swap with element in the first position
   c. Repeat (start from the second position and advance each time)

8. The first iteration involves n-1 comparisons and possibly 1 move, the second takes n-2 iterations and possibly one move, *etc*. Thus,

   Thus, $(n-1)+(n-2)+\cdots+2+1$ comparisons are made and up to *n* moves are made.

   Finally, since $\sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$, the complexity is $O(n^2)$.

8.0 – Bogo Sort

1. **Bogo Sort** – (*Bogus* sort). Consider a deck of cards that you want to sort. Assume there is a single unique ordering: 2S, 2H, 2D, 2C, 3S, 3H, 3D, 3C, *etc*. (in poker and bridge, the suites are frequently ordered: Spades, Hearts, Diamonds, Clubs). The algorithm is simple:

```
while not InOrder(deck) do Shuffle(deck);
```

2. Complexity:

   Notice that in worst case, this algorithm will never terminate. It is possible to continue shuffling a deck of cards and never observe that the cards are in order after the shuffle, just the same as it is possible to flip a coin indefinitely, always getting a *head*.

   How long does it take to determine if the cards are in order?

   How many times must we do this check? In the best case, once. In the worst case, unbounded. For the average case, we use a probabilistic argument. When we have n distinct elements, there are $n!$ distinct ordered arrangements of those elements. The probability of obtaining the cards in sorted order, after a shuffle is $p = 1/n!$, and is independent and constant between iterations. Thus, using the geometric probability function, the expected number of shuffles to obtain a sorted set is $1/p = n!$ with $O(n)$ work required each time to establish if order exists. Thus, $O(n \cdot n!) = O(n!)$.

Best	Average	Worst
$O(n)$	$O(n!)$	unbounded

8.3 – Insertion Sort

1. **Insertion Sort** – The basic idea is simple: partition the items into sorted and unsorted. At each iteration, move the next unsorted item into the sorted partition. Initially, the first item is sorted and you take the second item and determine whether it belongs in front of the first. Next, you take the third item and determine where it goes among the first two sorted items, *etc*.

2. **Algorithm**

```
insertSort( a )
   Loop over unsorted items
      cur = current unsorted item
      Loop over sorted positions, decreasing
         if cur < curSorted
            shuffle curSorted backwards
         else
            break
      put cur in current sorted position
```

```
insertSort( a )
   for( i=1; i<a.length; i++ )
      temp = a[i];
      for( j=i; j>=0; j--)
         if( temp < a[j-1] )
            a[j] = a[j-1];
         else
            break
      a[j] = temp
```

figure 8.2

Insertion sort implementation

```
1     /**
2      * Simple insertion sort
3      */
4     public static <AnyType extends Comparable<? super AnyType>>
5     void insertionSort( AnyType [ ] a )
6     {
7         for( int p = 1; p < a.length; p++ )
8         {
9             AnyType tmp = a[ p ];
10            int j = p;
11
12            for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
13                a[ j ] = a[ j - 1 ];
14            a[ j ] = tmp;
15        }
16    }
```

3. Complexity. Since both nested loops can take *n* iterations, the algorithm is $O(n^2)$ in worst case. If the data is presorted, the running time is $O(n)$ as the inner loop's comparison fails every time. If the data is *almost sorted* (defined later), then the running time will be quick.

4. An *inversion* is a pair of elements that are *out of order* in an array. In other words, it is any ordered pair (i, j) such that $i < j$ and $A_i > A_j$. For example, the sequence {8,5,9,2,6,3} has 10 inversions:

   (8,5), (8,2), (8,6), (8,3), (5,2), (5,3), (9,2), (9,6), (9,3), (6,3)

   Consider an array of size n:

	Number of Inversions	**Complexity**	**Comment**
Best	0	$O(n)$	Sorted
Average	$n(n-1)/4$	$O(n^2)$	
Worst	$n(n-1)/2$	$O(n^2)$	Reverse-sorted

5. Note the way insertion sort works. The number of inversions in the input array is the number of times line 13 is executed, *e.g.* the number of *swaps*. Swapping two elements that are out of place removes exactly one inversion. This process continues until there are no more inversions.

6. Thus, another way to think about the complexity is to say that if there are I inversions initially, then I swaps will occur. In addition, $O(n)$ other work must be done for the algorithm. Thus, the total complexity is $O(I + n)$. Thus, if the number of inversions is $O(n)$, then the total complexity is linear as well. However, on average, it is quadratic.

7. Insertion sort falls into a class of algorithms (including Selection sort and Bubble sort) that operate by exchanging adjacent elements. It can be shown that any such algorithm can never be better than $O(n^2)$ on average.

8.4 – Shell Sort

1. The first algorithm to substantially improve on Insertion sort was Shell sort which has a sub-quadratic run-time. Though not the fastest in this class, it has the simplest code of the faster algorithms. The idea of Shell sort is to take the 1-d input array and put the values into a 2-d array with a certain number of columns. Sort the columns using insertion sortrearrange it so that it is in is:

   1. Represent input array to 2-d array

   2. While num columns > 0
      a. For each column
         i. insert sort column
      b. Reduce number of columns

2. Example: Initial array: 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2

First iteration, use 7 columns:

Initial Arrangement

3	7	9	0	5	1	6
8	4	2	0	6	1	5
7	3	4	9	8	2	

→

Columns Sorted

3	3	2	0	5	1	5
7	4	4	0	6	1	6
8	7	9	9	8	2	

Second Iteration, use 3 columns:

Initial Arrangement

3	3	2
0	5	1
5	7	4
4	0	6
1	6	8
7	9	9
8	2	

→

Columns Sorted

0	0	1
1	2	2
3	3	4
4	5	6
5	6	8
7	7	9
8	9	

Third Iteration, use 1 column (shown transposed). You can see that most of the data is in order.

0	0	1	1	2	2	3	3	4	4	5	6	5	6	8	7	7	9	8	9

3. We do not actually need a 2-d array to implement this algorithm, we can use a properly indexed 1-d array. For instance, when we are conceptualizing that there are 7 columns in the 2-d array, to iterate over a "column", simply access every 7th item. Thus, Shell sort compares elements that are far apart, and then less far apart, shrinking until it arrives at the basic insertion sort. In the example above:

<u>7 Columns:</u>

Do insertion sort on every 7th value:

3	7	9	0	5	1	6	8	4	2	0	6	1	5	7	3	4	9	8	2

Result:

3	3	2	0	5	1	5	7	4	4	0	6	1	6	8	7	9	9	8	2

<u>3 Columns:</u>

Do insertion sort on every 3rd value:

3	3	2	0	5	1	5	7	4	4	0	6	1	6	8	7	9	9	8	2

Result:

0	0	1	1	2	2	3	3	4	4	5	6	5	6	8	7	7	9	8	9

<u>1 Column:</u>

Do insertion sort:

0	0	1	1	2	2	3	3	4	4	5	6	5	6	8	7	7	9	8	9

Result:

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9

4. To proceed, we need to be a bit more formal about the specification of Shell's algorithm. Shell sort defines an *increment* for each iteration which corresponds to the number of columns (or every n^{th} element) in that iteration. Shell sort uses an increment sequence $h_1, h_2, \ldots, h_t$, where h_t is the number of columns for the first iteration, h_{t-1} is the number of columns for the second iteration, ..., and $h_1 = 1$ for the last iterations. For instance, in the example above, the sequence was 1, 3, 7. In the example below, the sequence is 1, 3, 5. Any sequence can be used as long as $h_1 = 1$, but some choices of sequences are better than others. After an iteration where the increment is h_k, all elements spaced h_k apart are sorted. We say that such an array is h_k-sorted.

figure 8.5

Shellsort after each pass if the increment sequence is {1, 3, 5}

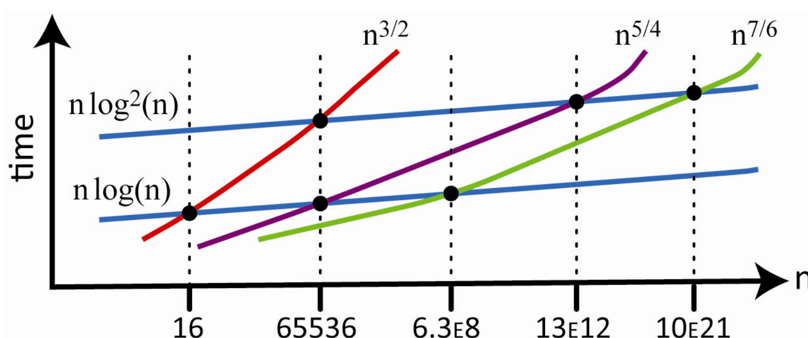
Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

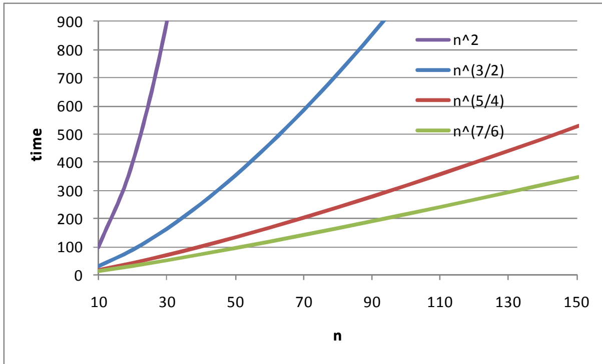
5. The complexity of Shell sort depends on the increment sequence. Analyzing the complexity for a particular sequence is challenging. Several strategies are shown below for choosing the increment sequence and what is known about their complexities.

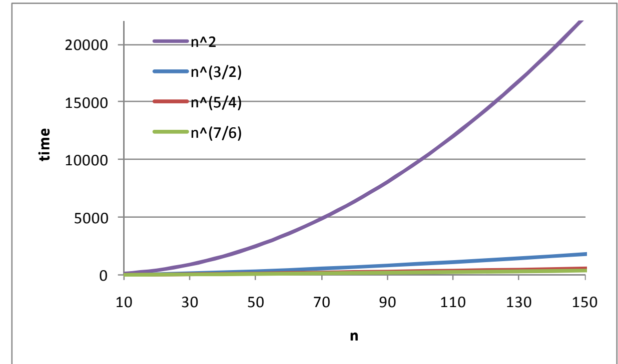
	Complexity	
Increment Strategy	**Average**	**Worst**
Shell: Start at $n/2$ and halve until 1 is reached	$O(n^{3/2})$	$O(n^2)$
Odd-only: Start at $n/2$ and halve, if even add 1, until 1 is reached	$O(n^{5/4})$*	$O(n^{3/2})$
Gonnet: Divide by 2.2 until 1 is reached	$O(n^{7/6})$*	
Pratt		$O(n \log^2 n)$

*Not proven, but empirical evidence suggests.

6. It can be proven that the best sorting algorithms can do no better than $O(n \log n)$. So, how do these complexities from the table above compare:







7. Suppose that we start with an array of size 1000. Here are the increments:

Shell's	Odd Only	Divide by 2.2
500	501	454
250	251	227
125	125	113
62	63	56
31	31	28
15	15	14
7	7	6
3	3	3
1	1	1

8. Run-time example:

		Shellsort		
N	**Insertion Sort**	**Shell's Increments**	**Odd Gaps Only**	**Dividing by 2.2**
10,000	575	10	11	9
20,000	2,489	23	23	20
40,000	10,635	51	49	41
80,000	42,818	114	105	86
160,000	174,333	270	233	194
320,000	NA	665	530	451
640,000	NA	1,593	1,161	939

figure 8.6

Running time of the insertion sort and Shellsort for various increment sequences

9. Shell-sort implementation:

```
1       /**
2        * Shellsort, using a sequence suggested by Gonnet.
3        */
4       public static <AnyType extends Comparable<? super AnyType>>
5       void shellsort( AnyType [ ] a )
6       {
7           for( int gap = a.length / 2; gap > 0;
8                        gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) )
9               for( int i = gap; i < a.length; i++ )
10              {
11                  AnyType tmp = a[ i ];
12                  int j = i;
13
14                  for( ; j >= gap && tmp.compareTo( a[j-gap] ) < 0; j -= gap )
15                      a[ j ] = a[ j - gap ];
16                  a[ j ] = tmp;
17              }
18      }
```

figure 8.7

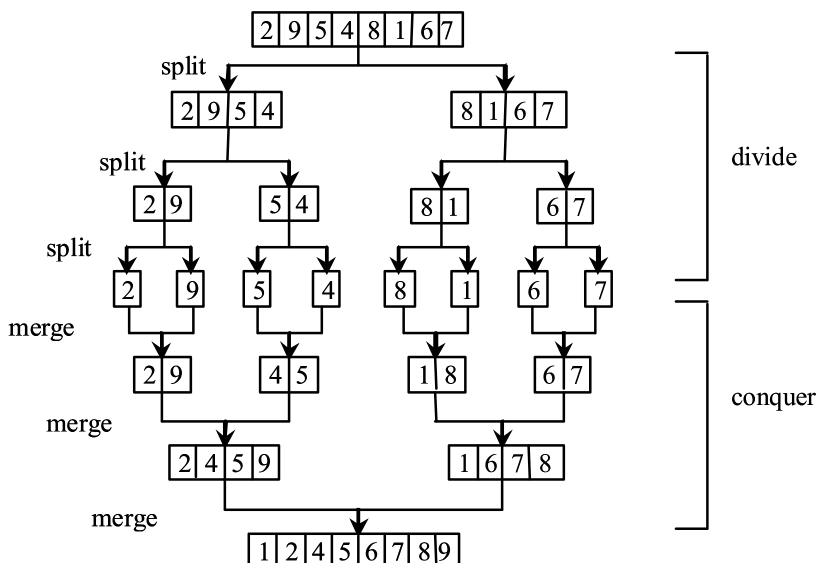
Shellsort implementation

8.5 – Merge Sort

1. Merge Sort is a recursive strategy that uses divide-and-conquer so that two ½ sized problems are solved with $O(n)$ overhead. The basic idea is:

    1. If the number of items to sort is 0 or 1, return
    2. Recursively sort the first and second halves separately
    3. Merge the two sorted halves into a sorted group

2. Example:



3. Thus, an algorithm for MergeSort:

```
mergeSort( list )
   if list.size = 0 or 1 then
      return
   else
      mergeSort( firstHalfList )
      mergeSort( secondHalfList )
      merge( firstHalfList, secondHalfList )
   return
```

4. Note, we do not need two separate arrays to hold the first and second halves of the list. We can use indices *left* and *center* for the first list and *center+1* and *right* for the second list. We will need an additional array to hold the result of the *merge* operation, temporarily, before it is copied back to the original list. The implementation of merge sort is shown below.

```
1     /**
2      * Mergesort algorithm.
3      * @param a an array of Comparable items.
4      */
5     public static <AnyType extends Comparable<? super AnyType>>
6     void mergeSort( AnyType [ ] a )
7     {
8         AnyType [ ] tmpArray = (AnyType []) new Comparable[ a.length ];
9         mergeSort( a, tmpArray, 0, a.length - 1 );
10    }
11
12    /**
13     * Internal method that makes recursive calls.
14     * @param a an array of Comparable items.
15     * @param tmpArray an array to place the merged result.
16     * @param left the left-most index of the subarray.
17     * @param right the right-most index of the subarray.
18     */
19    private static <AnyType extends Comparable<? super AnyType>>
20    void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray,
21                    int left, int right )
22    {
23        if( left < right )
24        {
25            int center = ( left + right ) / 2;
26            mergeSort( a, tmpArray, left, center );
27            mergeSort( a, tmpArray, center + 1, right );
28            merge( a, tmpArray, left, center + 1, right );
29        }
30    }
```

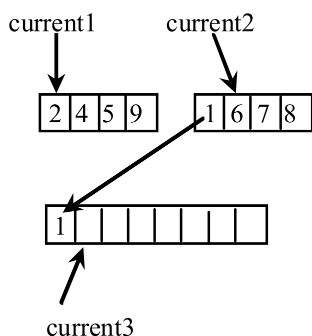
figure 8.8

Basic `mergeSort` routines

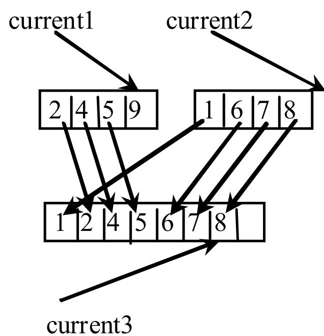
5. The *merge* operation takes two sorted lists and puts them together. An algorithm for *merge:*

```
Merge( list1, list2 )
   list = create new list
   While still elements in list1 and list2
      m = min(next element in list1, next element in list2)
      Put m in list
   Copy any leftovers from list1 to list
   Copy any leftovers from list2 to list
```

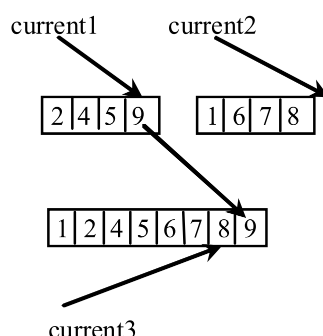
6. To implement merge, we use three counters, one for list1, one for list2, and one for the new list.



(a) After moving 1 to temp



(b) After moving all the elements in list2 to temp



(c) After moving 9 to temp

7. It should be clear that *merge* is an $O(n)$ operation as at each step of the loop, we move one element to merged list.

8. The implementation of *merge* is shown below.

```
1   /**
2    * Internal method that merges two sorted halves of a subarray.
3    * @param a an array of Comparable items.
4    * @param tmpArray an array to place the merged result.
5    * @param leftPos the left-most index of the subarray.
6    * @param rightPos the index of the start of the second half.
7    * @param rightEnd the right-most index of the subarray.
8    */
9   private static <AnyType extends Comparable<? super AnyType>>
10  void merge( AnyType [ ] a, AnyType [ ] tmpArray,
11              int leftPos, int rightPos, int rightEnd )
12  {
13      int leftEnd = rightPos - 1;
14      int tmpPos = leftPos;
15      int numElements = rightEnd - leftPos + 1;
16
17      // Main loop
18      while( leftPos <= leftEnd && rightPos <= rightEnd )
19          if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
20              tmpArray[ tmpPos++ ] = a[ leftPos++ ];
21          else
22              tmpArray[ tmpPos++ ] = a[ rightPos++ ];
23
24      while( leftPos <= leftEnd )    // Copy rest of first half
25          tmpArray[ tmpPos++ ] = a[ leftPos++ ];
26
27      while( rightPos <= rightEnd )  // Copy rest of right half
28          tmpArray[ tmpPos++ ] = a[ rightPos++ ];
29
30      // Copy tmpArray back
31      for( int i = 0; i < numElements; i++, rightEnd-- )
32          a[ rightEnd ] = tmpArray[ rightEnd ];
33  }
```

figure 8.9

The merge routine

9. Now, let's consider the complexity of merge sort. The only work being done is by *merge*, which is $O(n)$ and this occurs $O(\log n)$ times. Thus, merge sort runs in $O(n \log n)$ for best, average, and worst cases.

10. MergeSort uses linear extra memory (*tmpArray* in the code above). Copying to and from this memory slows the algorithm down, though it doesn't affect its complexity.

11. In Java, comparing objects is expensive because of the use of functions objects. Moving objects is not expensive because only references change.

12. MergeSort uses the fewest number of comparisons of all the popular sorting algorithms. Thus, it is a good general purpose algorithm for sorting objects in Java. MergeSort is used in the java.util.Arrays.sort method. These comparisons do not carry over to other languages and to primitive types.

8.6 – Quick Sort

1. Quick Sort is another divide and conquer algorithm. It is used in the java.util.Arrays.sort method to sort primitive data types. C++ uses quick sort for all sorting. It is based on the following steps:

2. Quick Sort – recursively partition a list based on a *pivot*.

3. Example:

Initial List:

7	9	3	8	6	1	5

For now, we will arbitrarily choose the first number, 7 as pivot and partition so that values to the left of pivot are less than or equal to pivot and values on right side are greater than pivot.

Left

3	6	1	5

7

Right

9	8

Repeat on Left list. Choose first number, 3 as pivot and partition.

Left

1

3

Right

6	5

Repeat on Left list. Only 1 item, so sorted.
Repeat on Right list. Choose first number, 6 as pivot and partition.

Left

5

6

Right

Repeat on Left list. Only 1 item, so sorted.
Repeat on Right list. No items, so sorted.

Repeat on Right list. Choose first number, 9 as pivot and partition

Left

8

9

Right

Repeat on Left list. Only 1 item, so sorted.
Repeat on Right list. No items, so sorted.

4. An algorithm

```
QuickSort( list )
If the number of items is 0 or 1, return
Pick any item, p as the pivot
Partition the list into a Left and Right sublists
   Left = items less than p
   Right = items greater than p
return QuickSort( Left ) + p + QuickSort( Right )
```

5. In the quick sort algorithm, there is $O(n)$ overhead at each recursive step to do the partitioning. In the worst case, the largest (smallest) value is chosen as the pivot leaving a left (right) sublist of size $n-1$, which requires $n-1$ steps to partition. If this process continues, choosing the largest (smallest) value to pivot on, we can see that the total work will be $1+2+\cdots+n=\frac{n(n+1)}{2}=O(n^2)$.

   For the best case, we choose a partition each step that equally divides the list into left and right sub-lists. In this case, similar to merge sort, this recursion occurs $\log n$ times. Thus, the best case $O(n \log n)$. It turns out that the average case is also $O(n \log n)$.

6. How do we choose the partition? You should never use the first element, as in the example above. If the data is sorted in some way, using the first element will lead to poor performance. A safe choice is to use the middle element. A better strategy is to use the *median-of-three* approach. There, we use the median of the elements in the first, middle, and last positions.

7. We can do quick sort without requiring any additional memory. In other words, we can do the partitioning by swapping elements in the original array. Consider this algorithm:

   Start from the left. Find the first element that is greater than pivot.
   Start from right. Find the first element, working backwards that is less than or equal to the pivot.
   Swap these two values.
   Repeat until no more swaps.
   Move pivot into proper position.

8. Example:

figure 8.18
Original array

figure 8.11
Partitioning algorithm: Pivot element 6 is placed at the end.

figure 8.12
Partitioning algorithm: `i` stops at large element 8; `j` stops at small element 2.

figure 8.13
Partitioning algorithm: The out-of-order elements 8 and 2 are swapped.

figure 8.14
Partitioning algorithm: `i` stops at large element 9; `j` stops at small element 5.

figure 8.15

Partitioning algorithm: The out-of-order elements 9 and 5 are swapped.

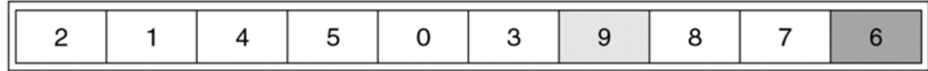


figure 8.16

Partitioning algorithm: `i` stops at large element 9; `j` stops at small element 3.

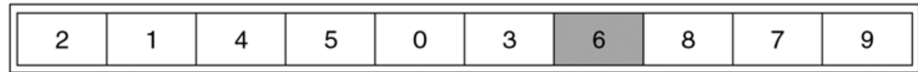


figure 8.17

Partitioning algorithm: Swap pivot and element in position `i`.

9. Partitioning using the median-of-three approach is actually simpler and more efficient. Start by sorting the three values used for the median and put the smallest in the first position, the median in the middle, and the largest in the last position.

10. The recursive nature of quick sort tells us that we will generate many recursive calls that only have small subsets. Thus, the author proposes testing the size of the subsets, and when they are smaller than a cutoff value, say 10, we apply insertion sort. Adopting this approach guarantees an $O(n \log n)$ worst case.

11. The implementation of quick sort:

figure 8.21

Quicksort with median-of-three partitioning and cutoff for small arrays

```
1       /**
2        * Quicksort algorithm (driver)
3        */
4       public static <AnyType extends Comparable<? super AnyType>>
5       void quicksort( AnyType [ ] a )
6       {
7           quicksort( a, 0, a.length - 1 );
8       }
9
10      /**
11       * Internal quicksort method that makes recursive calls.
12       * Uses median-of-three partitioning and a cutoff.
13       */
14      private static <AnyType extends Comparable<? super AnyType>>
15      void quicksort( AnyType [ ] a, int low, int high )
16      {
17          if( low + CUTOFF > high )
18              insertionSort( a, low, high );
19          else
20          {       // Sort low, middle, high
21              int middle = ( low + high ) / 2;
22              if( a[ middle ].compareTo( a[ low ] ) < 0 )
23                  swapReferences( a, low, middle );
24              if( a[ high ].compareTo( a[ low ] ) < 0 )
25                  swapReferences( a, low, high );
26              if( a[ high ].compareTo( a[ middle ] ) < 0 )
27                  swapReferences( a, middle, high );
28
29                  // Place pivot at position high - 1
30              swapReferences( a, middle, high - 1 );
31              AnyType pivot = a[ high - 1 ];
32
33                  // Begin partitioning
34              int i, j;
35              for( i = low, j = high - 1; ; )
36              {
37                  while( a[ ++i ].compareTo( pivot ) < 0 )
38                      ;
39                  while( pivot.compareTo( a[ --j ] ) < 0 )
40                      ;
41                  if( i >= j )
42                      break;
43                  swapReferences( a, i, j );
44              }
45                  // Restore pivot
46              swapReferences( a, i, high - 1 );
47
48              quicksort( a, low, i - 1 );    // Sort small elements
49              quicksort( a, i + 1, high );   // Sort large elements
50          }
51      }
```

8.7 – Quick Select

1. Suppose we want only the kth smallest item in a list of size n. Obviously, we could sort the list to obtain the desired item resulting in an average cost of $O(n \log n)$. However, Quick sort can be modified to return the kth smallest item and runs in $O(n)$ time, on average.

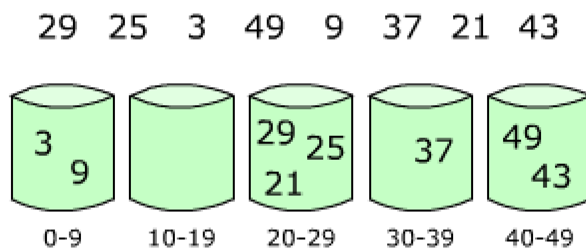
8.8 – A Lower Bound for Sorting

1. It can be proved (and our text does!) that any sorting algorithm that uses comparisons must use at least $\mathrm{n} \log \mathrm{n}$ comparisons for some input sequence.

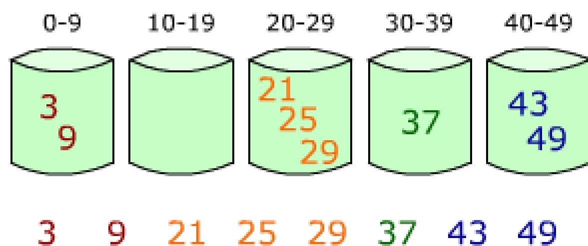
2. Can we do better than this? Yes, if we abandon comparisons. Consider the bucket sort:

    1. Set up an array of initially empty "buckets."
    2. **Scatter**: Go over the original array, putting each object in its bucket.
    3. Sort each non-empty bucket.
    4. **Gather**: Visit the buckets in order and put all elements back into the original array.

    Elements are scattered among the buckets:



    Then, elements are sorted within each bin:



3. In a more general situation where we have n objects to sort and there are N distinct keys, then we can setup N buckets, one for each key. Then, as we go through the items, we simply place them in the correct bucket. At the conclusion, we iterate over the buckets and the extract the items which will be in order. This has complexity $O(n + N)$.