Sorting

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

08/15/2017
7.1 Preliminaries
The STL uses a function template whose parameters represent the start and end markers of a range in a container, and an optional comparator:

```c
void sort( Iterator begin, Iterator end, Comparator cmp );
```
If it is necessary that equal-valued items retain their original order, then `stable_sort` should be used instead of `sort`. The sorting algorithm is generally quicksort.
### 7.2.1 The Algorithm

For pass $p = 1$ through $N - 1$, move the element in position $p$ left until elements 0 through $p$ are correctly ordered.

<table>
<thead>
<tr>
<th>Original</th>
<th>34</th>
<th>8</th>
<th>64</th>
<th>51</th>
<th>32</th>
<th>21</th>
<th>Positions Moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>After $p = 1$</td>
<td>8</td>
<td>34</td>
<td>64</td>
<td>51</td>
<td>32</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>After $p = 2$</td>
<td>8</td>
<td>34</td>
<td>64</td>
<td>51</td>
<td>32</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>After $p = 3$</td>
<td>8</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>32</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>After $p = 4$</td>
<td>8</td>
<td>32</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>After $p = 5$</td>
<td>8</td>
<td>21</td>
<td>32</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 7.1** Insertion sort after each pass
```cpp
/**
 * Simple insertion sort.
 */

template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size(); p++ )
    {
        Comparable tmp = a[ p ];
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

**Figure 7.2** Insertion sort routine
The following figures demonstrate use of the STL interface with iterator array access and a function object for comparison. The generic type of array elements changes from Comparable to Object which need not include operator\(<\). This requires helper routines with Object as an additional template parameter.
Figure 7.3

```cpp
1 template <typename Iterator>
2 void insertionSort( const Iterator & begin, const Iterator & end )
3 {
4     if( begin != end )
5         insertionSortHelp( begin, end, *begin );
6 }
7
8
9 template <typename Iterator, typename Object>
10 void insertionSortHelp( const Iterator & begin, const Iterator & end,
11                        const Object & obj )
12 {
13     insertionSort( begin, end, less<Object>() );
14 }
```

**Figure 7.3** Two-parameter sort invokes three-parameter sort via a helper routine that establishes `Object` as a generic type.
Figure 7.4

```c++
template <typename Iterator, typename Comparator>
void insertionSort(const Iterator & begin, const Iterator & end,
                     Comparator lessThan )
{
    if( begin != end )
        insertionSort( begin, end, lessThan, *begin );
}

template <typename Iterator, typename Comparator, typename Object>
void insertionSort(const Iterator & begin, const Iterator & end,
                    Comparator lessThan, const Object & obj )
{
    Iterator j;

    for( Iterator p = begin+1; p != end; ++p )
    {
        Object tmp = *p;
        for( j = p; j != begin && lessThan( tmp, *( j-1 ) ); --j )
            *j = *(j-1);
        *j = tmp;
    }
}
```

Figure 7.4 Three-parameter sort invokes four-parameter helper routine that establishes Object as a generic type
For each $p = 1$ to $N - 1$ the inner loop is executed at most $p$ times, resulting in an operation count of $O(N^2)$. This upper bound is achieved when the input array is in strictly decreasing order. If, on the other hand, the input is already increasing, the operation count is $O(N)$. Because of this wide variation, it is worth analyzing the average-case behavior.
7.3 A Lower Bound for Simple Sorting Algorithms

Define an *inversion* in array \( a \) as an ordered pair \((i, j)\) with the property that \( i < j \) but \( a[i] > a[j] \). For example, there are nine inversions in the sequence 34, 8, 64, 51, 32, 21. This is exactly the number of (implicit) swaps needed to sort the array. The running time for insertion sort is thus \( O(N + I) \) for \( I \) inversions in the input array.

To define average running time, assume there are no duplicate array elements. Then, since only relative order is important, we may assume that the input is a permutation of integers 1 to \( N \). We also assume that all permutations are equally likely.

**Theorem 7.1**: The average number of inversions in an array of \( N \) distinct elements is \( N(N - 1)/4 \).

**proof**: A list \( L \) and the reverse-ordered list \( L_r \) have the same number of inversions. The number of inversions in \( L \) added to the number of inversions in \( L_r \) is the number of ordered pairs \((i, j)\) with \( i < j \): \( N(N - 1)/2 \). □
Theorem 7.2: Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average.

**proof:** The average number of inversions is initially $N(N-1)/4$ and, because the elements are adjacent, each swap removes only one inversion. □

The theorem applies to an entire class of sorting algorithms, including insertion sort, bubble sort, and selection sort. Most lower-bound proofs are much more complicated than upper-bound proofs. A subquadratic sorting algorithm must eliminate more than one inversion per exchange by operating on elements that are far apart.
Shell sort is a simple algorithm but has a very complex analysis. Its performance is good enough for moderately large input sets. Donald Shell proposed a sequence of $h_k$-sorts using a diminishing increment sequence $h_t, \ldots, h_2, h_1 = 1$, where an $h_k$-sort results in $a[i] \leq a[i + h_k]$ for all $i$.

<table>
<thead>
<tr>
<th>Original</th>
<th>81</th>
<th>94</th>
<th>11</th>
<th>96</th>
<th>12</th>
<th>35</th>
<th>17</th>
<th>95</th>
<th>28</th>
<th>58</th>
<th>41</th>
<th>75</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 5-sort</td>
<td>35</td>
<td>17</td>
<td>11</td>
<td>28</td>
<td>12</td>
<td>41</td>
<td>75</td>
<td>15</td>
<td>96</td>
<td>58</td>
<td>81</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>After 3-sort</td>
<td>28</td>
<td>12</td>
<td>11</td>
<td>35</td>
<td>15</td>
<td>41</td>
<td>58</td>
<td>17</td>
<td>94</td>
<td>75</td>
<td>81</td>
<td>96</td>
<td>95</td>
</tr>
<tr>
<td>After 1-sort</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>28</td>
<td>35</td>
<td>41</td>
<td>58</td>
<td>75</td>
<td>81</td>
<td>94</td>
<td>95</td>
<td>96</td>
</tr>
</tbody>
</table>

**Figure 7.5 Shell sort after each pass**
Figure 7.6

```cpp
/**
 * Shellsort, using Shell's (poor) increments.
 */
template <typename Comparable>
void shellsort( vector<Comparable> & a )
{
    for( int gap = a.size() / 2; gap > 0; gap /= 2 )
        for( int i = gap; i < a.size(); i++ )
            {
            Comparable tmp = a[ i ];
            int j = i;
            for( ; j >= gap && tmp < a[ j - gap ]; j -= gap )
                a[ j ] = a[ j - gap ];
            a[ j ] = tmp;
        }
}
```

**Figure 7.6** Shellsort routine using Shell's increments (better increments are possible)
7.4.1 Worst-Case Analysis of Shellsort

**Theorem 7.3:** The worst-case running time of Shellsort, using Shell’s increments, is $\Theta(N^2)$.

**proof:** Suppose $N$ is a power of 2 and the $N/2$ smallest elements are in the odd positions (with position 1 first). Then before the beginning of the last pass the $i^{th}$ smallest element is in position $2i - 1$ and must be moved $i - 1$ positions for $i \leq N/2$. The number of operations is therefore at least $\sum_{i=1}^{N/2} i - 1 = \Omega(N^2)$. A pass with increment $h_k$ consists of $h_k$ insertion sorts of about $N/h_k$ elements for a total cost of $O(h_k(N/h_k)^2) = O(N^2/h_k)$. Summing over all passes gives an upper bound of $O(\sum_{i=1}^{t} N^2/h_i) = O(N^2)$ since $\sum_{i=1}^{t} 1/h_i < 2$. \qed

<table>
<thead>
<tr>
<th>Start</th>
<th>1</th>
<th>9</th>
<th>2</th>
<th>10</th>
<th>3</th>
<th>11</th>
<th>4</th>
<th>12</th>
<th>5</th>
<th>13</th>
<th>6</th>
<th>14</th>
<th>7</th>
<th>15</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 8-sort</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>14</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>After 4-sort</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>14</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>After 2-sort</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>14</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>After 1-sort</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

**Figure 7.7** Bad case for Shellsort with Shell’s increments
Better results are obtained if consecutive increments have no common factors. Hibbard suggested the sequence 1, 3, 7, \ldots, 2^k - 1. The following theorem can be proven.

**Theorem 7.4**: The worst-case running time of Shellsort using Hibbard’s increments is $\Theta(N^{3/2})$.

The $\Theta(N^{3/2})$ upper bound applies to a wide range of increment sequences. The average-case running time with Hibbard’s increments is thought to be $\Theta(N^{5/4})$ based on simulations. There are several sequences that give a worst-case $O(N^{4/3})$ running time. The average running time for these sequences is conjectured to be $O(N^{7/6})$. The best of these is \{1, 5, 19, 41, 109, \ldots\}, in which the terms are either $9 \cdot 4^i - 9 \cdot 2^i + 1$ or $4^i - 3 \cdot 2^i + 1$. 
Recall from Chapter 6 that we can build a binary heap in time \(O(N)\), and then sort the elements by performing a sequence of \(N\) deleteMin operations, each of which takes \(O(\log N)\) time, resulting in an \(O(N \log N)\) algorithm. Since each deleteMin leaves an empty space at the end of the array, we can overwrite the array with the sorted elements in nonincreasing order. If we require nondecreasing order, we can use a max-heap. For example, the sequence 31, 41, 59, 26, 53, 58, 97 results in the heap shown in Figure 7.8.

### 7.5.1 Analysis of Heapsort

**Theorem 7.5:** The average number of comparisons used to heapsort a random permutation of \(N\) distinct items is 
\[2N \log N - O(N \log \log N).\]

It can also be shown that heapsort always uses at least 
\(N \log N - O(N)\) comparisons, and that there are inputs that can achieve this bound.
Figures 7.8 and 7.9

**Figure 7.8** (Max) heap after buildHeap phase

**Figure 7.9** Heap after first deleteMax
```cpp
/**
 * Standard heapsort.
 */

template <typename Comparable>
void heapSort( vector<Comparable> & a )
{
    for( int i = a.size() / 2; i >= 0; i-- ) /* buildMax */
        perDown( a, i, a.size() );
    for( int j = a.size() - 1; j > 0; j-- )
    {
        swap( a[0], a[j] ); /* deleteMax */
        perDown( a, 0, j );
    }
}

/**
 * Internal method for heapSort.
 * i is the index of an item in the heap.
 * Returns the index of the left child.
 */
inline int leftChild( int i )
{
    return 2 * i + 1;
}

/**
 * Internal method for heapSort that is used in deleteMax and buildMax.
 * i is the position from which to percolate down.
 * n is the logical size of the binary heap.
 */
template <typename Comparable>
void perDown( vector<Comparable> & a, int i, int n )
{
    int child;
    Comparable tmp;
    for( tmp = a[i]; leftChild( i ) < n; i = child )
    {
        child = leftChild( i );
        if( child < n - 1 && a[child] < a[child + 1] )
            child++;
        if( tmp < a[child] )
            a[i] = a[child];
        else
            break;
    }
    a[i] = tmp;
}
```

Mergesort is a recursive $O(N \log N)$ divide-and-conquer algorithm: recursively mergesort the left and right halves of the array, and then merge the two sorted lists in a single pass using three pointers. For example, the sequence 24, 13, 26, 1, 2, 27, 38, 15 has sorted left half 1, 13, 24, 26 and right half 2, 15 27, 38.
```c++
/**
 * Mergesort algorithm (driver).
 */

template<typename Comparable>
void mergeSort( vector<Comparable> & a )
{
    vector<Comparable> tmpArray( a.size() );
    mergeSort( a, tmpArray, 0, a.size() - 1 );
}

/**
 * Internal method that makes recursive calls.
 * a is an array of Comparable items.
 * tmpArray is an array to place the merged result.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 */

template<typename Comparable>
void mergeSort( vector<Comparable> & a, 
                vector<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 );
    }
}
```

**Figure 7.11 Mergesort routines**
/**
 * Internal method that merges two sorted halves of a subarray.
 * a is an array of Comparable items.
 * tmpArray is an array to place the merged result.
 * leftPos is the left-most index of the subarray.
 * rightPos is the index of the start of the second half.
 * rightEnd is the right-most index of the subarray.
 */

template<typename Comparable>
void merge(vector<Comparable> & a, vector<Comparable> & tmpArray,
           int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ] <= a[ rightPos ] )
            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++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}
We simplify the analysis by assuming that \( N \) is a power of 2. The running time is the time required for two mergesorts of size \( N/2 \) plus the merge time which is linear, resulting in the recurrence relation

\[
T(1) = 1 \\
T(N) = 2T(N/2) + N
\]

**Solution of recurrence relation**

\[
T(N/2) = 2T(N/4) + N/2 \Rightarrow T(N) = 4T(N/4) + 2N \\
T(N/2) = 4T(N/8) + 2N/2 \Rightarrow T(N) = 8T(N/8) + 3N \\
T(N/2) = 8T(N/16) + 3N/2 \Rightarrow T(N) = 16T(N/16) + 4N
\]

Using \( k = \log N \), we obtain

\[
T(N) = 2^k T(N/2^k) + k \cdot N = NT(1) + N \log N = \mathcal{O}(N \log N).
\]
Mergesort uses the lowest number of comparisons of all the popular sorting algorithms but has the disadvantage that it uses additional memory and incurs the cost of copying to the temporary array and back. This can be expensive for large objects in a generic sort. The running time compared to other $O(N \log N)$ sorting algorithms thus depends on the relative costs of comparing and moving elements, and those costs are language dependent. In Java element comparisons are expensive but moving elements is cheap, and mergesort is therefore the algorithm used in the standard Java library for sorting.

In C++ on the other hand, copying can be expensive while compares are often relatively cheap. Consequently, C++ libraries commonly use Quicksort.
7.7 Quicksort

Quicksort, due to Tony Hoare in 1959, is the fastest known generic sorting algorithm with average running time $O(N \log N)$. Like mergesort, it’s a divide-and-conquer recursive algorithm, but it operates in place with no additional storage required. This efficiency more than compensates for the lack of equal-size recursive calls. The algorithm is simple to understand but difficult to implement correctly. An array $S$ is sorted by the following four steps.

1. If the number of elements in $S$ is 0 or 1, then return.
2. Choose a *pivot* element $v$ in $S$.
3. **Partition** $S - \{v\}$ into disjoint groups
   
   $S_1 = \{x \in S - \{v\} | x \leq v\}$ and $S_2 = \{x \in S - \{v\} | x \geq v\}$.
4. Return the sequence quicksort($S_1$), $v$, quicksort($S_2$).

Note the ambiguity associated with elements equal to the pivot element. It is essential to handle this efficiently.
Figure 7.13 The steps of quicksort illustrated by example
Choosing the first element as the pivot is acceptable if the input is random, but produces a poor partition (at every recursive call) if the input (or a large portion of the input) is presorted or in reverse order, as often occurs in practice. The result is quadratic running time on a data set which can be sorted in linear time by a simple algorithm.

Choosing a random pivot is safe but expensive. A better strategy is median-of-three partitioning: choose the median of the three elements in the first, middle, and last positions.
The first step in partitioning a sequence is to swap the pivot element with the last element. Denote by $i$ and $j$ the indices of the first and next-to-last elements, respectively; e.g., 8 and 7 in the sequence 8, 1, 4, 9, 0, 3, 5, 2, 7, 6 for which 6 is the pivot element. We want to end up with elements smaller than 6 in the left portion of the array, and larger elements in the right portion. While $i < j$, we move $i$ to the right until we encounter a large element, and move $j$ to the left until we encounter a small element, and then swap those elements if $i$ is still less than $j$. The final step, after $i$ and $j$ have crossed, is to swap the pivot element with the element pointed to by $i$, resulting in the sequence 2, 1, 4, 5, 0, 3, 6, 8, 7, 9.
How do we treat duplicate elements? In an array with $10^6$ elements it is not unlikely that there are 50,000 duplicates, and they will be end up in contiguous locations. It is therefore imperative that we efficiently handle the case that all elements are identical. The correct strategy is to stop moving both $i$ and $j$, and to perform the useless swaps in the case of equality. Then $i$ and $j$ cross in the middle resulting in equal-length subarrays and running time $O(N \log N)$ by the mergesort analysis. The alternative results in $O(N^2)$ time to do nothing.

7.7.3 Small Arrays
Since quicksort is recursive, small subarrays occur frequently, and these are more efficiently handled by insertion sort. A cutoff size of $N = 10$ can save 15% of the running time. The cutoff also avoids degenerate cases such as taking the median of three elements with $N < 3$. 
7.7.4 Actual Quicksort Routines

The pivot selection code, in addition to returning the median of three element, actually sorts the elements, placing the smallest in a[left], the largest in a[right], and the pivot in a[right−1]. This allows us to initialize i and j to left+1 and right−2. In addition to the efficiency advantage, a[left] and a[right−1] serve as sentinels for j and i, respectively.

Note that the swap should be compiled in-line for efficiency. The code in Figure 7.17 produces an infinite loop if a[i] = a[j] = pivot.

1    /**
2     * Quicksort algorithm (driver).
3     */
4    template<typename Comparable>
5    void quicksort( vector<Comparable> & a )
6    {
7        quicksort( a, 0, a.size() - 1 );
8    }

**Figure 7.14** Driver for quicksort
Figure 7.15

```cpp
/**
 * Return median of left, center, and right.
 * Order these and hide the pivot.
 */

template <typename Comparable>
const Comparable & median3( vector<Comparable> & a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ] < a[ left ] )
        swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
    swap( a[ center ], a[ right - 1 ] );
    return a[ right - 1 ];
}
```

Figure 7.15  Code to perform medium-of-three partitioning
template<typename Comparable>
void quicksort(vector<Comparable> & a, int left, int right)
{
    if (left + 10 <= right)
    {
        Comparable pivot = median3(a, left, right);
        // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; )
        {
            while( a[ ++i ] < pivot ) { }
            while( pivot < a[ --j ] ) { }
            if( i < j )
                swap( a[ i ], a[ j ]);
            else
                break;
        }
        swap( a[ i ], a[ 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);
}
7.7.5 Analysis of Quicksort

We assume a randomly selected pivot and no cutoff for small arrays. The running time for quicksort is the running time for the two recursive calls plus the linear time for partitioning:

\[ T(N) = T(i) + T(N - i - 1) + cN, \]

where \( i = |S_1| \) is the number of elements in \( S_1 \).

**Worst-case Analysis**

Suppose the pivot is always the smallest element, so that \( i = 0 \). Then, ignoring the insignificant term \( T(0) = 1 \), the recurrence is

\[ T(N) = T(N - 1) + cN, \]

so that \( T(2) = T(1) + 2c, T(3) = T(2) + 3c = T(1) + 5c, T(4) = T(3) + 4c = T(1) + 9c, \ldots, \) and

\[ T(N) = T(1) + c \sum_{i=2}^{N} i = O(N^2). \]
In the best case, the pivot is in the middle, and the running time can be modeled by $T(N) = 2T(N/2) + cN$ so that $T(N) = N \log N$ by the mergesort analysis. For the average case, assume that each size for $S_1$ is equally likely so that the average value of $T(i)$ and, hence of $T(N - 1 - i)$, is $(1/N) \sum_{j=0}^{N-1} T(j)$, resulting in

$$T(N) = \frac{2}{N} \left[ \sum_{j=0}^{N-1} T(j) \right] + cN.$$ 

Hence

$$NT(N) = 2 \left[ \sum_{j=0}^{N-1} T(j) \right] + cN^2.$$ 

We eliminate the sum by substituting $N - 1$ for $N$ to obtain a second equation, and then subtract the second equation from the first.
Average-Case Analysis of Quicksort continued

\[(N - 1)T(N - 1) = 2 \left[ \sum_{j=0}^{N-2} T(j) \right] + c(N - 1)^2,\]

\[NT(N) - (N - 1)T(N - 1) = 2T(N - 1) + 2cN - c.\]

We drop the insignificant term \(c\) to obtain

\[NT(N) = (N + 1)T(N - 1) + 2cN.\]

Divide both sides by \(N(N + 1)\) and then telescope:

\[
\frac{T(N)}{N + 1} = \frac{T(N - 1)}{N} + \frac{2c}{N + 1}
\]

\[
\frac{T(N - 1)}{N} = \frac{T(N - 2)}{N - 1} + \frac{2c}{N}
\]

\[
\frac{T(N - 2)}{N - 1} = \frac{T(N - 3)}{N - 2} + \frac{2c}{N - 1}
\]

\[
\ldots
\]

\[
T(2)/3 = T(1)/2 + 2c/3
\]
Adding the above equations, we obtain

\[
\frac{T(N)}{N + 1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i}.
\]

The sum is approximately equal to \(\ln(N + 1) + \gamma - 3/2\), where \(\gamma\) is Euler's constant, so that

\[
\frac{T(N)}{N + 1} = O(\log N),
\]

and

\[
T(N) = O(N \log N).
\]
The selection problem of finding the $k^{th}$ smallest or $k^{th}$ largest element is solved (for all values of $k$) by sorting. For a single value of $k$, the problem is solved efficiently by a small modification of quicksort involving one recursive call instead of two calls. If $k \leq |S_1|$, then the $k^{th}$ smallest element must be in $S_1$ and there is no need to sort $S_2$. Similarly, if the $k^{th}$ smallest element is in $S_2$, then it is not necessary to sort $S_1$.

Recall that a priority queue can be used to solve the selection problem in running time $O(N + k \log N)$. For finding the median, the time is $\Theta(N \log N)$. The worst case running time for quickselect is the same as that for quicksort: $O(N^2)$. This occurs when $S_1$ or $S_2$ is empty and both algorithms use only one recursive call. The average running time for quickselect is $O(N)$. 

R. J. Renka

Sorting
Figure 7.18

```c
/**
 * Internal selection method that makes recursive calls.
 * Uses median-of-three partitioning and a cutoff of 10.
 * Places the kth smallest item in a[k-1].
 * a is an array of Comparable items.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 * k is the desired rank (1 is minimum) in the entire array.
 */

template <typename Comparable>
void quickSelect( vector<Comparable> & a, int left, int right, int k )
{
    if( left + 10 <= right )
    {
        Comparable pivot = median3( a, left, right );

        // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; )
        {
            while( a[ ++i ] < pivot ) {}  // While sorting
            while( pivot < a[ --j ] ) {}  // While sorting
            if( i < j )
                swap( a[ i ], a[ j ] );
            else
                break;
        }

        swap( a[ i ], a[ right - 1 ] ); // Restore pivot

        // Recurse; only this part changes
        if( k <= i )
            quickSelect( a, left, i - 1, k );
        else if( k > i + 1 )
            quickSelect( a, i + 1, right, k );
    }
    else // Do an insertion sort on the subarray
        insertionSort( a, left, right );
}
```

Figure 7.18 Main quickselect routine
7.8 Indirect Sorting

If the Comparable objects being sorted are large, it is more efficient to create an array of pointers, sort the pointers (by the dereferenced values), and then use the pointers to move the objects rather than repeatedly moving objects in the sort. A simple algorithm uses a second array of $N$ Comparable objects, but that could exceed our memory limits since both $N$ and the size of Comparable are assumed to be large. It also requires $2N$ copies. That’s an improvement over the original algorithm, but we can do better by using an **in-situ permutation** which uses $L+1$ copies for each cycle of length $L > 1$ (and no copies for cycles of length 1). Let $C_L$ denote the number of cycles of length $L$. Then $N = \sum_{L=1}^{N} L \cdot C_L$, and the number of copies is $\sum_{L=2}^{N} (L + 1) C_L = N - C_1 + \sum_{L=2}^{N} C_L$. The minimum, maximum, and mean values of this expression are 0, $3N/2$, and $N - 2 + H_N$ for the harmonic sum $H_N = \sum_{i=1}^{N} 1/i$. 


R. J. Renka  Sorting
The following example has a cycle of length 2 and a cycle of length 3. The sequence of copies associated with the latter cycle, beginning with $p[2]$, are as follows.

```c
    tmp = a[2];
    a[2] = a[4];
    a[4] = a[3];
```

**Figure 7.19** Using an array of pointers to sort
template<typename Comparable>

class Pointer
{
   public:
      Pointer( Comparable *rhs = NULL ) : pointee( rhs ) {}

      bool operator<( const Pointer & rhs ) const
      { return *pointee < *rhs.pointee; }

      Comparable *() const
      { return pointee; }

   private:
      Comparable *pointee;
};

Figure 7.20 Class that stores a pointer to a Comparable
template <typename Comparable>
void largeObjectSort( vector<Comparable> & a )
{
    vector<Pointer<Comparable> > p( a.size( ) );
    int i, j, nextj;

    for( i = 0; i < a.size( ); i++ )
        p[ i ] = &a[ i ];

    quicksort( p );

    // Shuffle items in place
    for( i = 0; i < a.size( ); i++ )
        if( p[ i ] != &a[ i ] )
            { 
                Comparable tmp = a[ i ];
                for( j = i; p[ j ] != &a[ i ]; j = nextj )
                    { 
                        nextj = p[ j ] - &a[ 0 ];
                        a[ j ] = *p[ j ];
                        p[ j ] = &a[ j ];
                    }
                a[ j ] = tmp;
                p[ j ] = &a[ j ];
            }
With regard to the code in Figures 7.20 and 7.21, note the following.

- We cannot declare `p` as a `vector<Comparable*>` because we would have to change the template argument for `quicksort` to `Comparable*`, and the `<` operator for pointers is not what we want and cannot be overridden. We need a new class with an appropriate `<` operator as in the `Employee` class of Figure 1.21.

- Pointer is a **smart pointer class**. Unlike a plain pointer, it automatically initializes itself to `NULL` if no initial value is provided.

- The method at line 10 defines a type conversion from `Pointer<Comparable>` to `Comparable*` which simplifies the code. Along with the omission of `explicit` on the constructor, this disables strong typing and enables implicit type conversion.
On line 22 the constructor is used to create a temporary `Pointer<Comparable>` for assignment to `p[i]`.

Line 28 uses implicit conversion of `p[i]` to a temporary variable of type `Comparable*` for which `operator!` is defined. If we had defined `operator!` in class `Pointer` there would be an ambiguity due to the dual-direction implicit conversions, and line 28 would not compile.

Line 33 implicitly converts `p[j]` to `Comparable*` and computes the difference between the two primitive pointers. Since they point to elements in the same array, the difference is the signed displacement (number of elements) between the array elements.

Line 34 dereferences a `Pointer<Comparable>` for which there is no dereference operator, but the type conversion at line 10 creates a temporary `Comparable*` from `p[j]`. 
Any sorting algorithm that uses only comparisons has worst-case time complexity $\Omega(N \log N)$, so that mergesort and heapsort are optimal to within a constant factor. The average-case complexity is also $\Omega(N \log N)$, so that quicksort is optimal within a constant.

### 7.9.1 Decision Trees

A binary decision tree can be used to prove information-theoretic lower bounds. Any algorithm that uses binary decisions to distinguish among $P$ possible cases will require $\lceil \log P \rceil$ decisions for some input. In the following tree we associate each node with a set of possible orderings, and each edge with a comparison. We assume that all $N$ elements are distinct. Every algorithm that sorts by using only comparisons can be represented by a decision tree. The average number of comparisons is the average depth of the leaves.
Figure 7.22 A decision tree for three-element sort
Lemma 7.1: A binary tree of depth $d$ has at most $L = 2^d$ leaves.
proof: The result is true for $d = 0$. For $d > 0$ the root cannot be a leaf, and the left and right subtrees each have depth at most $d - 1$. By the inductive hypothesis, they have at most $2^{d-1}$ leaves, giving a total of $2^d$ leaves. □

Lemma 7.2: A binary tree with $L$ leaves has depth at least $\lceil \log L \rceil$.
proof: $L \leq 2^d \Rightarrow d \geq \lceil \log L \rceil$. □

Theorem 7.6: Any sorting algorithm that uses only comparisons between elements requires at least $\lceil \log L \rceil$ comparisons in the worst case.
proof: A decision tree to sort $N$ elements must have $N!$ leaves (one for each permutation). The theorem follows from Lemma 7.2. □
**Theorem 7.7**: Any sorting algorithm that uses only comparisons between elements requires $\Omega(N \log N)$ comparisons.

**proof**: From the previous theorem, $\log(N!)$ comparisons are required.

$$
\log(N!) = \log(N(N-1)(N-2) \cdots (2)(1)) \\
= \log(N) + \log(N-1) + \log(N-2) + \cdots + \log(1) \\
\geq \log(N) + \log(N-1) + \log(N-2) + \cdots \log(N/2) \\
\geq (N/2) \log(N/2) \\
\geq (N/2) \log(N) - (N/2) \\
= \Omega(N \log N). \quad \square
$$
If additional information (beyond ordering information) about the input is available, it may be possible improve the efficiency of a sorting algorithm. For example, the input often consists of small integers which can be sorted in linear time by a bucket sort. For positive integers $A_1, A_2, \ldots, A_N$ bounded above by $M$, we keep an array $\text{count}$ of length $M$ initialized to zeros (empty buckets). For $i = 1$ to $N$, increment $\text{count}[A_i]$ by 1. Then create the sorted list by scanning the $\text{count}$ array. The complexity is $O(M + N)$ or $O(N)$ if $M = O(N)$. 

R. J. Renka

Sorting
Summary

Figure 7.24 displays running times on a slow computer for sorting random permutations of integers. The unoptimized quicksort uses a simple pivoting strategy and no cutoff. The fastest method is the textbook version of quicksort. It could be further optimized by a nonrecursive implementation, coding the median-of-three routine in-line, and using assembly language.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Insertion Sort $O(N^2)$</th>
<th>Shellsort $O(N^{7/6})$</th>
<th>Heapsort $O(N \log N)$</th>
<th>Quicksort $O(N \log N)$</th>
<th>Quicksort (opt.) $O(N \log N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000001</td>
<td>0.000002</td>
<td>0.000003</td>
<td>0.000002</td>
<td>0.000002</td>
</tr>
<tr>
<td>100</td>
<td>0.000106</td>
<td>0.000039</td>
<td>0.000052</td>
<td>0.000025</td>
<td>0.000023</td>
</tr>
<tr>
<td>1000</td>
<td>0.011240</td>
<td>0.000678</td>
<td>0.000750</td>
<td>0.000365</td>
<td>0.000316</td>
</tr>
<tr>
<td>10000</td>
<td>1.047</td>
<td>0.009782</td>
<td>0.010215</td>
<td>0.004612</td>
<td>0.004129</td>
</tr>
<tr>
<td>100000</td>
<td>110.492</td>
<td>0.13438</td>
<td>0.139542</td>
<td>0.058481</td>
<td>0.052790</td>
</tr>
<tr>
<td>1000000</td>
<td>NA</td>
<td>1.6777</td>
<td>1.7967</td>
<td>0.6842</td>
<td>0.6154</td>
</tr>
</tbody>
</table>

Figure 7.24 Comparison of different sorting algorithms (times in seconds)