1. Recursive Binary Search

The idea of the binary search algorithm can be expressed simply:

1. If the first index is greater than the last index, halt and return an impossible index value to indicate that the item searched for was not found.
2. Calculate the middle index between the first and last index
3. If the value at the middle is less than the value of the item sought, set the first index to the middle plus 1 and repeat from step 1.
4. If the value at the middle is greater than the value of the item sought, set the last index to the middle minus 1 and repeat from step 1.
5. If the value at the middle is equal to the value of the item sought, halt and return the middle index.

This easily converts into the iterative version of the binary search from earlier in the text:

```
long bsearch(elementtype array[ ], int listlen, elementtype item) {
    long first = 0, last = listlen-1, mid;
    while (first <= last) {
        mid = (first + last) / 2;
        if (array[mid] < item) first = mid + 1;
        else if (array[mid] > item) last = mid - 1;
        else return mid;
    }
    return -1;
}
```

The binary search algorithm is also easily expressed as a recursive function. To begin with, the function header must have four parameters. Instead of the `listlen` parameter of the iterative version, the recursive version requires the parameters `first` and `last`, which represent the range of the indices to be considered.

```
long bsearch(elementtype array[ ], int listlen, elementtype item) {
    becomes

    long rbsearch(elementtype array[ ], int first, int last, elementtype item) {
```
In the body of the function `rbsearch`, step one requires a decision be made as to when to stop searching and return an impossible index when the item is not found. This is simply an if with a return-if-true. All other steps are performed only if the Boolean expression of this first if fails.

```c
int rbsearch(elementtype array[], int first, int last, elementtype item) {
    if (first > last) return -1;
    :
    :
}
```

Step two is the calculation of the middle index between the values in `first` and `last`. This is much the same as in the iterative version and results in the following:

```c
int rbsearch(elementtype array[], int first, int last, elementtype item) {
    if (first > last) return -1;
    int mid = (first + last) / 2;
    :
    :
}
```

Step three checks to see if the value at the middle location is low when compared to the value in item. If it is, a recursive call is made to `rbsearch` with the value sent to `first` being adjusted to `mid + 1`. This results in the following code:

```c
int rbsearch(elementtype array[], int first, int last, elementtype item) {
    if (first > last) return -1;
    int mid = (first + last) / 2;
    if (array[mid] < item) return rbsearch(array, mid+1, last, item);
    :
    :
}
```

Step four checks to see if the value at the middle location is high when compared to the value in item. If it is, a recursive call is made to `rbsearch` with the value sent to `last` being adjusted to `mid - 1`. This results in the following code:

```c
int rbsearch(elementtype array[], int first, int last, elementtype item) {
    if (first > last) return -1;
    int mid = (first + last) / 2;
    if (array[mid] < item) return rbsearch(array, mid+1, last, item);
    else if (array[mid] > item) return rbsearch(array, first, mid-1, item);
    :
    :
}
```

Notice the inclusion of the else before if (array[mid] > item). Since the recursive call to `rbsearch` in the preceding will be resolved and returned, the use of else insures that the if (array[mid] > item) will not be unnecessarily evaluated.
Step 5 checks to see if the value in item and the value in array[mid] are equal. If they are, the function halts with a return of the middle index. Another else is used for step 5 and no if is needed to check to see if the value of item is equal to the value at array[mid] as that is the only possible option left. This results in the following completed function:

**Complete Recursive Binary Search**

```c
int rbsearch(elementtype array[], int first, int last, elementtype item) {
    if (first > last) return -1;
    int mid = (first + last) / 2;
    if (array[mid] < item) return rbsearch(array, mid+1, last, item);
    else if (array[mid] > item) return rbsearch(array, first, mid-1, item);
    else return mid;
}
```

Of course, this can be written in various ways. An else can be included after the first if, with a compound statement that encompasses all other code. The order of events in the nested if can be rearranged. These are cosmetic changes and do not alter the logic used to create rbsearch.

A typical use of the binary search algorithm is to search a list of structs stored in an array. This means that the item searched for must be checked against the field or fields that are used as the key in the struct. This too involves minor changes to the basic function. For example:

**Complete Recursive Binary Search on Key Field**

```c
int rbsearch(elementtype array[], int first, int last, keytype item) {
    if (first > last) return -1;
    int mid = (first + last) / 2;
    if (array[mid].key < item) return rbsearch(array, mid+1, last, item);
    else if (array[mid].key > item)
        return rbsearch(array, first, mid-1, item);
    else return mid;
}
```
A more substantive change to the recursive binary search involves calling it. In a class, it is often not desirable to allow the programmer using the class too much access to the data structures inside of the class. This is often enforced to insure that data structures are manipulated only in the ways provided in the public access section. In the case of \texttt{rbsearch}, the call requires knowledge of the actual indices being used, which may not be available to the programmer. To get around this, \texttt{rbsearch} and similar algorithms are often implemented as private functions, with public functions of reduced parameter needs created to call the private functions. For example:

```cpp
class example {
public:
    :
    :
    long search(keytype item);
    :
    :
private:
    :
    :
    long rbsearch(arraytype array[], int first, int last, keytype item);
    arraytype * array;
    int listlen;
    :
    :
};

long search(keytype item) {
    return rbsearch(array, 0, listlen, item);
}
```

**Exercise 38.1**
Using pencil and paper, create a trace of the \texttt{rbsearch}'s use of memory (see chapter 37 for an example of such a trace) as it searches for various elements that can be found (or not found) in the following list.

<table>
<thead>
<tr>
<th>value</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>11</th>
<th>18</th>
<th>21</th>
<th>22</th>
<th>25</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</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>
</tr>
</tbody>
</table>

**Programming Exercise 38.1**
Alter programming assignment 32.1 so that a recursive binary search is used.
2. **Divide and Conquer**

The binary search algorithms (recursive and iterative) were the first examples of *divide and conquer* algorithms we have studied. Divide and conquer refers to the practice of dividing the elements of a data structure into two halves with each decision made. One half is the group to be examined, the other is to be ignored at least for the time being. Divide and conquer strategies produce very efficient algorithms because the need to examine every element in a data structure over and over is eliminated and, instead, only a small subset of elements need be examined or if the entire body of elements need to be examined, the number of repetitions is greatly reduced.

3. **Modified Quick Sort**

*Modified quick sort* is a divide and conquer algorithm that offers a large improvement over the brute force sorts ([O(n^2)] growth in time) that have been studied thus far. In fact, modified quick sort offers no disadvantage in terms of growth in space [O(1) as it does not change the size of the data structure] and can sort with a growth in time of O(n log₂n), which is dramatically better than any previously studied sort. The disadvantage of modified quick sort is that it is complex. Most students do not find it intuitive and often are reduced to route memorization to learn the algorithm.

Here is an outline of the modified quick sort:

0. if the list exists, do 1 through 8
   1. select a mid point (generally best - middle element value)
   2. starting from the left and moving to the right, find the first element that has a value greater than the value of mid
   3. starting from the right and moving to the left, find the first element that has a value less than the value at the mid point
   4. if the left index is less than the right index, swap the values found in steps 2 and 3
   5. repeat steps 2, 3 and 4 until the indices of 2 and 3 cross
   6. recursively call (go to step 0) the function, passing the indices of the left most element and step 3
   7. recursively call (go to step 0) the function, passing the indices of step 2 and the rightmost element

The basic idea of modified quick sort is to divide the elements of a list into two groups: Those that are greater than a middle value and those that are less. This process is repeated with increasingly smaller sub-lists until the final sub-list size is one. At this point, each sub-list is sorted by default and the entire list is now in sorted order.
Here is the modified quick sort. Note that the function *swap* represents swapping two array elements.

```c
void qsort(arraytype a[ ], int lo, int hi) {
    if (lo < hi) {
        int mid = (lo + hi) / 2;
        arraytype midItem = a[mid];
        int left = lo - 1;
        int right = hi + 1;
        arraytype temp;
        do {
            do {
                left++;
            while (a[left] < midItem);
            do {
                right--;
            } while (midItem < a[right]);
            if (left < right) {
                temp = a[left];
                a[left] = a[right];
                a[right] = temp;
            }
        } while (left <= right);
        qsort(a, lo, right);
        qsort(a, left, hi);
    }
}
```

**Exercise 38.2**

Using pencil and paper, create a trace of the function *qsort*’s use of memory (see chapter 37 for an example) as it sorts the elements in the following list.

<table>
<thead>
<tr>
<th>value</th>
<th>18</th>
<th>5</th>
<th>22</th>
<th>30</th>
<th>11</th>
<th>25</th>
<th>3</th>
<th>6</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</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>
</tr>
</tbody>
</table>

**Programming Exercise 38.2**

Test the performance in real time of the modified quick sort algorithm against a brute force sorting algorithm with list lengths of 1,000, 10,000, 100,000 and 1,000,000 integers.
4. **Recursive Merge Sort**

The *recursive merge sort* algorithm is another divide and conquer sort. Because it is simpler in concept than the modified quick sort, merge sort is often easier for students to learn. Unfortunately, merge sort requires twice the memory that quick sort sorts requires. While this does not make any difference to *a priori* analysis (it still comes out as $O(1)$ in terms of space), doubling the amount of memory required for an array can make a huge difference in terms of the size of list that can be sorted on a real world system.

The basic idea of the recursive merge sort is as follows:

1. start rmerge
2. if the low index of the list is less than the high index of the list, do as follows:
   3. calculate the mid index
   4. recursively call rmerge with a sub-list left of and including the mid point
   5. recursively call rmerge with a sub-list right of the mid point
   6. call merge to merge the two sublists

Recursive merge sort operates on the idea that repeated recursive calls to itself will bring the length of the sub lists down to 1. A list of length 1 is by definition sorted. The call to merge will combine the two sorted lists and produce a sorted list of length 2. With each duel fall back from recursion, merge will be called to merge two sorted lists, producing another, longer, sorted list. Eventually, all of the sub-lists are merged and the complete list is sorted.
Here is the recursive merge sort and the merge algorithm:

```c
void rmerge(arraytype a[ ], int lo, int hi) {
    if (lo < hi) {
        int mid = (lo + hi) / 2;
        rmerge(a, lo, mid);
        rmerge(a, mid+1, hi);
        merge(a, lo, mid, hi);
    }
}

void merge(arraytype a[ ], int left1, int left2, int right2) {
    int right1 = left2+1;
    int start = left1;
    int t = left1;
    arraytype *temp;
    temp = new arraytype(right2);
    while (left1 <= left2 && right1 <= right2) { // standard merge algorithm
        if (a[left1] < a[right1]) {
            temp[t] = a[left1];
            t++;
            left1++;
        } else {
            temp[t] = a[right1];
            t++;
            right1++;
        }
    }
    if (left1 > left2) { // copy remaining to temp
        for (; right1<= right2; right1++) {
            temp[t] = a[right1];
            t++;
        }
    } else {
        for (; left1<= left2; left1++) {
            temp[t] = a[left1];
            t++;
        }
    }
    for (int x=start; x<t; x++)
        a[x] = temp[x]; // copy from temp array back
                               // to original array
    delete []temp;
}
```
Exercise 38.3
Using pencil and paper, create a trace of the \textit{rmerge} and \textit{merge}'s use of memory (see chapter 37 for an example) as it sorts the elements in the following list.

<table>
<thead>
<tr>
<th>value</th>
<th>18</th>
<th>5</th>
<th>22</th>
<th>30</th>
<th>11</th>
<th>25</th>
<th>3</th>
<th>6</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</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>
</tr>
</tbody>
</table>

Programming Exercise 38.3
Test the performance in real time of the merge sort algorithm against the modified quick sort algorithm with list lengths of 1,000, 10,000, 100,000 and 1,000,000 integers.

5. Heap Sort
The \textit{heap sort} algorithm is another divide and conquer sort often used to sort lists in arrays, although it is seldom implemented recursively. Heap sort is based on the idea of an array representing a tree that has the property of being a heap.

First, here is an illustration of the idea of an array as tree. All elements in the array are called \textit{nodes} of the tree. The node at \textit{a[0]} is known as the \textit{root} of the tree. Each node has two children, known as the left child and the right child. The children of node \textit{a[0]} are \textit{a[1]} (left child) and \textit{a[2]} (right child). The children of the node \textit{a[1]} are \textit{a[3]} and \textit{a[4]}. The children of \textit{a[2]} are \textit{a[5]} and \textit{a[6]}. Obviously there is a pattern that determines what the index is of a child of a node. If the index of the node is represented by \textit{i}, the left child of the node is \textit{2i+1} and the right child of the node is \textit{2i+2}. Any node that has no children is called a \textit{leaf}.

Example of a Tree in an Array of 11 Elements

\begin{center}
\begin{tikzpicture}[level distance=1.5cm, sibling distance=1.5cm]
  \node {a[0]}
    child {node {a[1]}
      child {node {a[3]}
        child {node {a[7]}}
        child {node {a[8]}}}
      child {node {a[4]}
        child {node {a[9]}}
        child {node {a[10]}}}
    }
    child {node {a[2]}
      child {node {a[5]}}
      child {node {a[6]}}
    }
\end{tikzpicture}
\end{center}

A list in an array is called a \textit{heap} if the value of each node is greater than or equal to the values of its children. This would make \textit{a[0]} the largest value in the array.
Heap sort is based on the idea that if $a[i+1..(n-1)]$ is a heap, then $a[i..(n-1)]$ can be easily made into a heap. Here is a function that takes in a heap located at $a[i+1]$ to $a[n-1]$ and returns a heap located at $a[i]$ to $a[n-1]$.

```c
void makeHeap(arraytype a[], int i, int n) {
    arraytype r = a[i];     // preserve root value
    int child;
    while (i < n/2) {     // if not leaf, continue
        child = 2*i+1;     // get index for left child
        if (child < n-1 && a[child] < a[child+1])  // if right child bigger, get
            child++;    // index for right child
        if (r >= a[child]) break;   // if root larger than child, stop
        a[i] = a[child];     // copy larger child to root
        i = child;     // set root index to child index
    }
    a[i] = r;      // copy old root value to child
}
```

By repeatedly calling `makeHeap` from a large $i$ to the smallest possible $i$ value, the array can be sorted.

```c
void heapSort(arraytype a[], int n) {
    int i;
    arraytype temp;
    for (i=(n-1)/2; i>=0; i--) makeHeap(a, i, n);  // repeated calls to makeHeap
    for (i=n-1; i>0; i--) {
        temp = a[i];     // swap order
        a[i] = a[0];
        a[0] = temp;
        makeHeap(a, 0, i-1);    // restore heap
    }
}
```

**Exercise 38.4**
Using pencil and paper, create a trace of the array elements as `heapSort`'s sorts the list.

<table>
<thead>
<tr>
<th>value</th>
<th>18</th>
<th>5</th>
<th>22</th>
<th>30</th>
<th>11</th>
<th>25</th>
<th>3</th>
<th>6</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</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>
</tr>
</tbody>
</table>

**Programming Exercise 38.4**
Test the performance in real time of the recursive heap sort algorithm against the modified quick sort and recursive merge sort algorithms with list lengths of 1,000, 10,000, 100,000 and 1,000,000 integers.