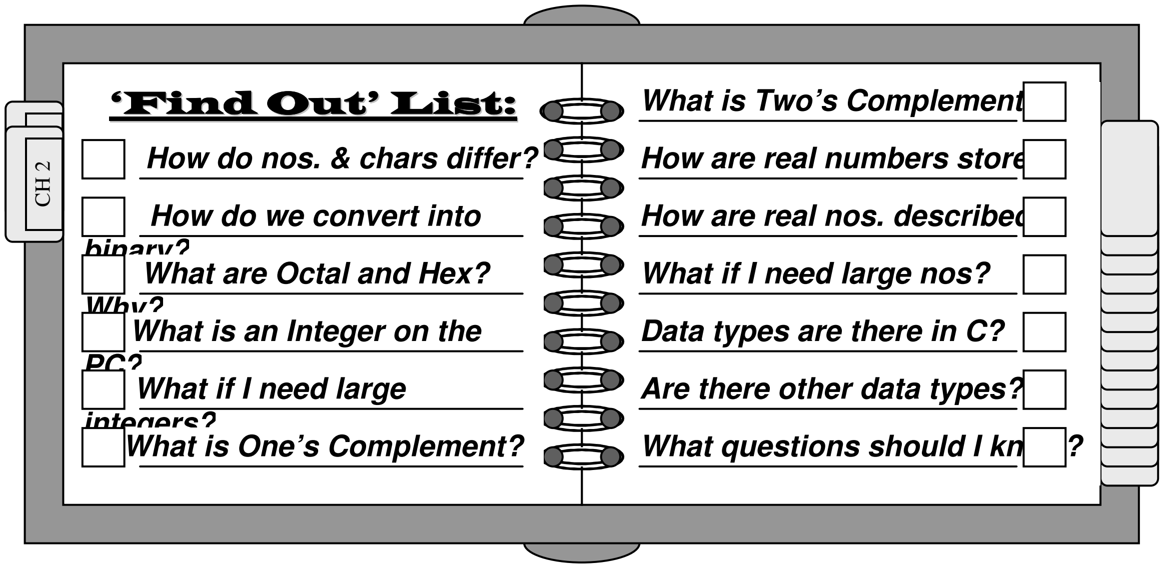
CHAPTER 9: SEARCHING AND SORTING

"Simplicity of life, even the barest, is not a misery,
but the very foundation of refinement"
William Morris (1834–96)


'Find Out' List:
CH 2
How do nos. & chars differ?
How do we convert into binary?
What are Octal and Hex? Why?
What is an Integer on the PC?
What if I need large integers?
What is One's Complement?
What is Two's Complement
How are real numbers store
How are real nos. describe
What if I need large nos?
Data types are there in C?
Are there other data types?
What questions should I kn ?

Introduction

This section starts off with a chapter which introduces no new data structures. It discusses two programming *techniques*. While this might initially seem strange, most textbooks on data structures devote a chapter or two to each of these concepts. Firstly, because they constitute major tasks which most computer programs must deal with, Secondly, they illustrate the major trade-off to be considered when implementing linear versus non-linear data structures, namely, speed of access vs. maintenance.

An understanding of the tradeoffs involved also helps to illustrate why we need to develop additional data structures, especially linked lists and binary trees. These structures help alleviate the problems encountered when trying to quickly access data without extensive maintenance. Additionally, this chapter will serve an introduction to non-contiguous storage of data, and how we can use this approach in an efficient manner.

Sequential Searches

A sequential search is the simplest way of traversing a list in order to find an element on it, and a technique which we have previously seen. Consider a variation on our c program 4.3., in which we searched for a value in an array. In program 9.1, we will get a value to search for from the user, and then attempt to find in the array.

C/C++ Code 9.1

```
#include <stdio.h>
#include <stdlib.h>

void main()
{  int i, num=1, myarray[8] = {6,9,2,4,1,12,3,10};          // initialize the array
   char s[10];                                               // for our gets function
   while (num != 0)                                          // Should we quit?
   {   i = 0;
       printf("Enter a (whole) number between 1 and 12 (0 to quit): ");
       num = atoi(gets(s));                                  // get the number to search
       if (num != 0)                                         // Should we quit?
     {     while((myarray[i] != num) && (i<8)) i++;          // Check the array
           if (i < 8)                                        // Element in the array?
              printf("\nThe number %d was found in position %d\n",num,i);
           else                                              // Element NOT in the array
              printf("\nThe number %d is not in the array\n",num);
     }
   }
}
```

What is happening here ???

Let's take a look at the array set up (from the declaration: *myarray*[8] = {6,9,2,4,1,12,3,10};)

Figure 9.1.

myarray offset:	0	1	2	3	4	5	6	7
myarray value:	6	9	2	4	1	12	3	10

Suppose that the user wishes to see if the value **4** is stored somewhere in the array. We begin by starting with the first value in the array (*myarray*[0] = 6), and see if the value contained at that array position is value 4. If it is, we stop. If it isn't, we continue searching (by incrementing the contents of location offset: i until we find it: *i*++;):

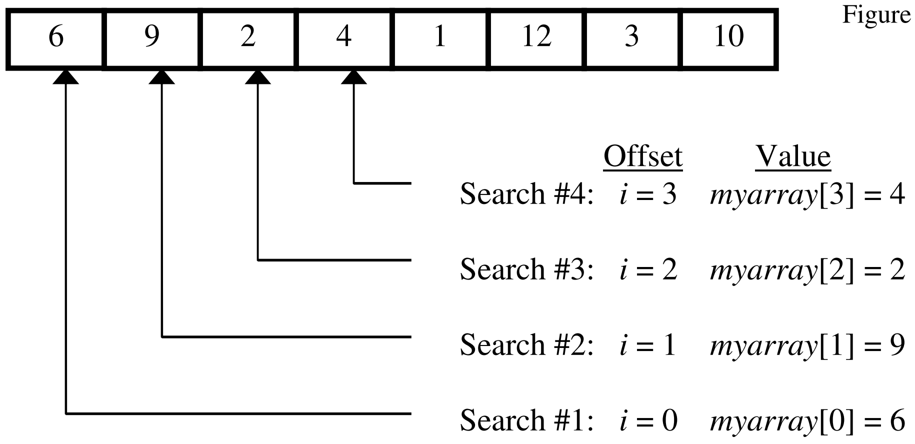

6
9
2
4
1
12
3
10
Offset
Value
Search #4: *i* = 3 *myarray*[3] = 4
Search #3: *i* = 2 *myarray*[2] = 2
Search #2: *i* = 1 *myarray*[1] = 9
Search #1: *i* = 0 *myarray*[0] = 6

Figure 9.2.

How many searches are required???

Obviously, that depends on the value the user enters (and how many elements are in an array). Table 9.1. shows the number of searches required for existing values (i.e., values in the array):

Table 9.1.

Search Value	Searches Required
6	**1**
9	**2**
2	**3**
4	**4**
1	**5**
12	**6**
3	**7**
10	**8**

That means that the minimum number of checks needed is 1 (if the user enters the value 6), and the maximum number of comparisons needed is 8 (if the user enters the value 10). For any value not on the list (e.g., the value 5), we must go though the entire list and that we need 8 searches (or 9, depending on how we define a search). Therefore, the *average* (n) number of comparisons required is:

$$\mathbf{n = \frac{Max.\ Checks + Min.\ Checks + 1}{2} = \frac{8 + 1}{2} = 4.5}$$

where n is the number of elements in an array. In this formula, one (1) is added to the numerator to account for elements not on the list.

Regardless of list length, the average (as well as maximum) number of checks required is completely dependent upon the number of elements in the list. Consider the different list lengths given in Table 9.2.:

Table 9.2.

Length	Comparisons Required	Length	Comparisons Required
10	(n+1)/2 = 11/2 = 5.5	5,000	(n+1)/2 = 5,001/2 = 2,500.5
25	(n+1)/2 = 25/2 = 12.5	10,000	(n+1)/2 = 1,0001/2 = 5,000.5
100	(n+1)/2 = 101/2 = 55.5	100,000	(n+1)/2 = 100,001/2 = 50,000.5
500	(n+1)/2 = 501/2 = 255.5	500,000	(n+1)/2 = 500,001/2 = 250,000.5
1,000	(n+1)/2 = 1,001/2 = 500.5	1,000,000	(n+1)/2 = 1,00,001/2 = 500,000.5

? ***Is there anyway to reduce the number of comparisons required???***

If the list is ***not*** sorted, no. We could just as easily search from the end of the list (continuing to the beginning), or apply some other search algorithm, but since the value being sought could be anywhere on the list (or not on it at all), these strategies would be futile.

Now, if the list were sorted or ordered in some known fashion ….

Binary Searches

If we know that a list is ordered (whether in ascending or descending order), we do **not** have to search every element on that list. Let's take a look at the same array, but this time as a sorted (ordered) list. Suppose we had made our declaration (in C code 9.1) as:

int *myarray*[8] = {1, 2, 3. 4, 6, 9, 10, 12};

which is in order. The array would appear as:

myarray offset:	0	1	2	3	4	5	6	7
myarray value:	1	2	3	4	6	9	10	12

Let's assume that we were once again looking for the value **4** in the list. We could arbitrarily select a position (based on array offset) in the list and compare the value found there with the value we are looking for. If, for example, we decide to look at offset 2 (*myarray*[2]), we would obviously find the value **3** stored there.

? ***What does that tell us???***

Because we know that the array is sorted (in ascending order), we therefore know that ***if*** the value 4 does exist in the list, can't be at offsets 0 (zero) or 1 (one); that saves us two comparisons. Because we already checked offset 2, the value must be in either offsets 3 .. 7, or not on the list at all.

Since we are arbitrarily checking locations, let's chose offset 5 (*myarray*[5]), which we already know contains the value 9. This time, because the value we are looking for (4) is smaller than the value 9, we can eliminate offsets 6 and 7 (as well as offset 5, which we just checked) from consideration. **If** the value 4 is on the list, it must be in positions 3 or 4.

Notice that after just two comparisons, we have eliminated six elements from consideration, and reduced our list of candidates to just two.

? ***Which position do we check next???***

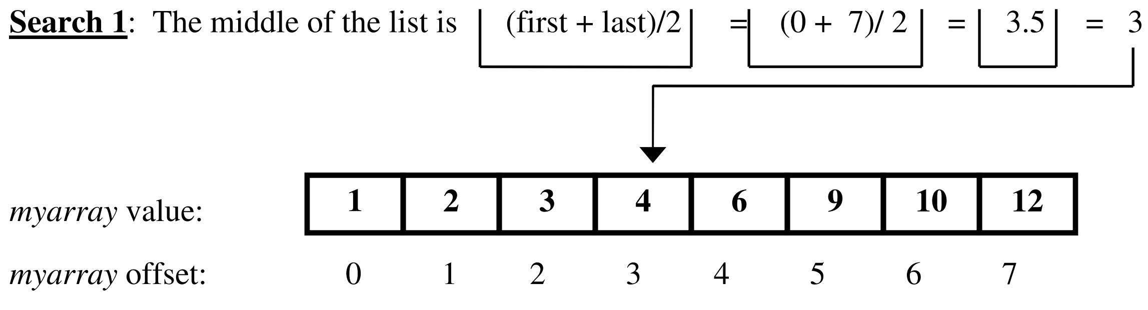
The way that we are searching (arbitrarily), it doesn't matter. If we decide to check offset 3, we will find it; if we don't, we will find it on the next try.

? ***Do we always arbitrarily choose an offset position???***

No, arbitrarily selecting a position doesn't make sense. We could, arbitrarily, select the first element in the list (*myarry*[0]) or the last element in the list (myarray[7]). In either of these cases, we are only eliminating one element from consideration (either the first or the last). We would like to eliminate as many elements from consideration as possible, or, in other words, reduce our list of potential candidates as quickly as possible. This implies that we should split our list in half each time, or perform a **binary search.**

This time, let's look for the value **6** (which we know is in position *myarray*[4]). The list contains 8 elements, with offsets ranging from 0 to 7, so let's split it in two and begin searching from there:

Figure 9.3.



Search 1: The middle of the list is $\lfloor (\text{first} + \text{last})/2 \rfloor = \lfloor (0 + 7)/2 \rfloor = \lfloor 3.5 \rfloor = 3$

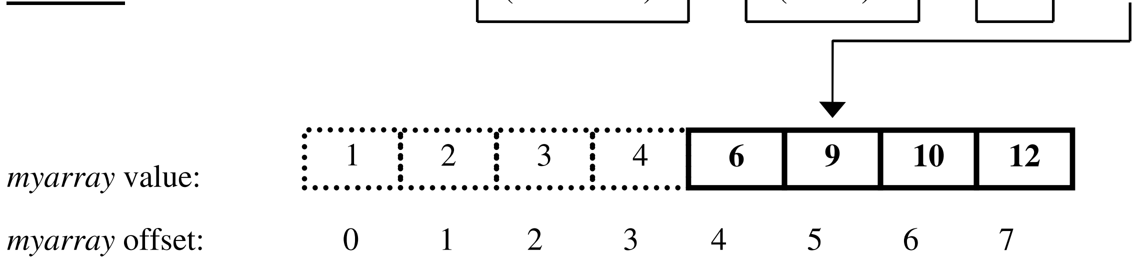
myarray value:	**1**	**2**	**3**	**4**	**6**	**9**	**10**	**12**
myarray offset:	0	1	2	3	4	5	6	7

Since 4 (the value contained at location *myarray*[3]) is *less than* 6 (our search value), we can eliminate offsets 0 through 3 from consideration (leaving offsets 4 through 7 to be checked).

At this point in time, we know that the element we are looking for can not be in positions (offsets) 1, 2, or three. The first place it could be at is offset 4. Therefore:

Figure 9.4.

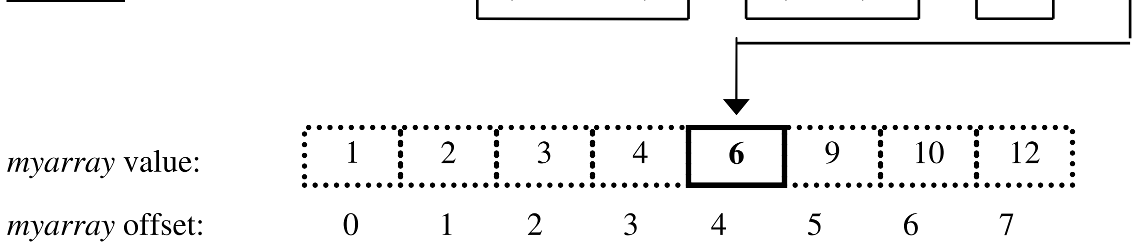
Search 2: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (4 + 7)/ 2 \rfloor = \lfloor 5.5 \rfloor = 5$



Since 9 (the value contained at location *myarray*[3]) is *greater than* 6 (our search value), we can eliminate offsets 5 through 7 from consideration (leaving only offset 4 to be checked. Therefore:

Figure 9.5.

Search 3: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (4 + 4)/ 2 \rfloor = \lfloor 4 \rfloor = 4$



And we have found the value we were looking for in three comparisons (as opposed to 5 if we were performing a sequential search).

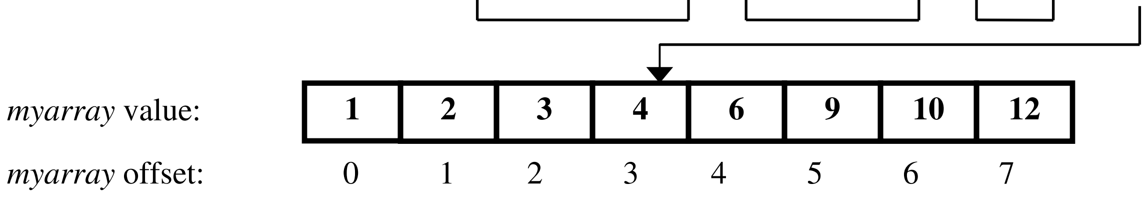
Why did we have to make the last search? We only had one element left in our list, so it had to be there!!!

Not necessarily.

Remember, we could have been searching for an element which was ***not*** on the list. Suppose we were looking for the value 5 (not on the list), we would follow the exact same procedure (try following the steps above) only to find that in search 3, the value stored in *myarray*[4] was not 5. Suppose we were looking for the value 11 (also *not* on the list). Our search steps would be:

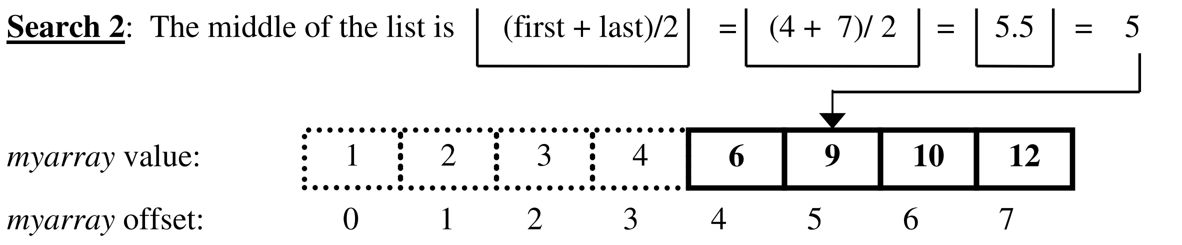
Figure 9.6.

Search 1: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (0 + 7)/ 2 \rfloor = \lfloor 3.5 \rfloor = 3$



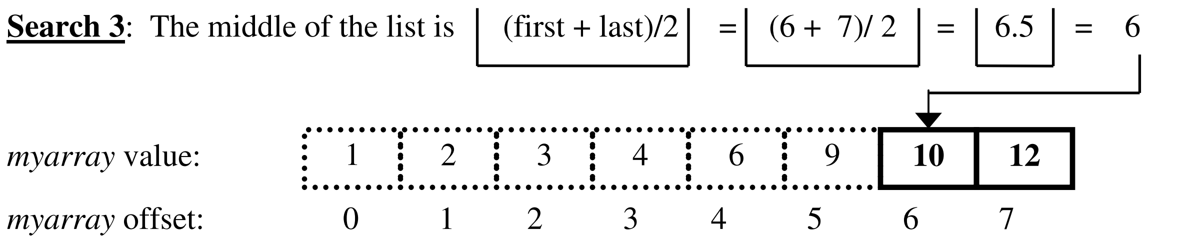
Since 4 (the value contained at location *myarray*[3]) is *less than* 11 (our search value), we can eliminate offsets 0 through 3 from consideration (leaving offsets 4 through 7 to be checked).At this point in time, we know that the element we are looking for can not be in positions (offsets) 1, 2, or three. The first place it could be at is offset 4. Therefore:

Figure 9.7.


Search 2: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (4 + 7)/2 \rfloor = \lfloor 5.5 \rfloor = 5$
myarray value: 1 2 3 4 **6** **9** **10** **12**
myarray offset: 0 1 2 3 4 5 6 7

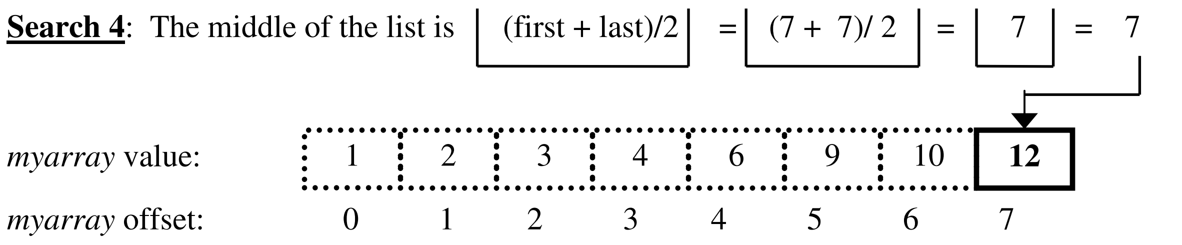
Since 9 (the value contained at location *myarray*[5]) is *less than* 11 (our search value), we can eliminate offsets 4 and 5 from consideration (leaving offsets 6 and 7 to be checked). Therefore:

Figure 9.8.


Search 3: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (6 + 7)/2 \rfloor = \lfloor 6.5 \rfloor = 6$
myarray value: 1 2 3 4 6 9 **10** **12**
myarray offset: 0 1 2 3 4 5 6 7

Since 10 (the value contained at location *myarray*[6]) is *less than* 11 (our search value), we can eliminate offset 6 from consideration (leaving only offset 7 to be checked). Therefore:

Figure 9.9.


Search 4: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (7 + 7)/2 \rfloor = \lfloor 7 \rfloor = 7$
myarray value: 1 2 3 4 6 9 10 **12**
myarray offset: 0 1 2 3 4 5 6 7

? ***Now what???***

According to our algorithm, ***IF*** the value 11 exists on the list, it *must* be at some offset less than 7.

? ***But, that can't be !!!***

That is true, but we don't necessarily know that. However, given that we know that if the value exists on the list the *first* available offset is 7, and (as we determined above), and the *last* available offset where is 6, then we know that the element can't be on the list since the *first* offset is now greater than the *last* offset.

? ***What about the minimum, maximum, and average number of comparisons required using a binary search???***

The minimum number of comparisons is (of course) only 1 (we could always get lucky and find it on the first try). The maximum number of comparisons (for n > 30)[1] needed in a binary search is (we will avoid the derivations):

$$\lceil \log_2 n \rceil \qquad \text{Formula 9.1.}$$

and, on average, the number of comparisons needed:

$$(\log_2 n) - 1 \quad \text{(for } n > 30) \qquad \text{Formula 9.2.}$$

Where n = list length

? ***When is a binary search better than a sequential search???***

That is not as easy to answer as it might appear. The code for a Binary search is considerably more complex than the code for a sequential search, so it takes longer to execute. A general rule of thumb, however, is somewhere around 30 to 50 elements.

? ***How do the two approaches compare, aside from the programming considerations???***

Consider the following comparisons (Table 9.3.):

Table 9.3.

Number of Elements	**Maximum Sequential n+1**	**Average Sequential (n + 1)/2**	**Maximum Binary $\lceil \log_2 n \rceil$**	**Average Binary $\log_2 n$ - 1**
10	11	5.5	4	2.9
50	51	25.5	6	4.6
100	101	55.5	7	5.8
1,000	1,001	500.5	10	9.0
10,000	10,000	5,000.5	14	12.3
100,000	100,001	50,000.5	17	15.6
1,000,000	1,000,001	500,000.5	20	18.9
10,000,000	10,000,001	50,000,000.5	24	22.3

[1] Discrpencies with the table below are due the small number of elements (8) in the array

Notice that as the number of elements to be searched increases, the number of comparisons necessary in a binary search is exponentially less than the number of comparisons necessary for a sequential search.

? ***What about the c code necessary to perform a binary search???***

Let's perform an interactive binary search on our (sorted) array where the user gets to choose a numeric value to look for:

C/C++ Code 9.2

```
#include <stdio.h>                                                   // Std. I/O
#include <stdlib.h>                                                  // For atoi
 void main()
{  int value = 1, myarray[8] = {1,2,3,4,6,9,10,12},                  // The sorted array
       first, last, mid, found;                                      // first, last, middle
  char s[10];                                                        // For our gets
  while (value != 0)                                                 // Continue until Quit
  {   printf("\nEnter a (whole) number between 1 and 12 (0 to quit): ");  // Prompt the user
      value = atoi(gets(s));                                         // Get search Value
      if (value != 0)                                                // If not Quit
      {    first = 0;                                                // first is initially 0
           last = 8 - 1;                                             // last is initially 7
          mid = (first + last)/2;                                    // middle will be 3
           found = 1;                                                // value is not yet found
          while(found != 0)
         {    printf("value = %d, middle = %d, last = %d, first = %d\n",  // Show starting values
                      value, mid, last, first);
              if (myarray[mid] == value)                             // Was the value found?
             {   printf("The number %d was found in position %d\n",  // The print the info
                          value, mid);
                 found = 0;  }                                       // set the found flag
             else                                                    // if not found
                 if (first  >= last)                                 // NOT in the array??
                 {  printf("The value %d is not in the array\n", value);  // Then indicate
                    found = 0; }                                     // and set the flag
                else
                {    if (value > myarray[mid])                       // Is it in the top half?
                         first = ++mid;                              // set the first value
                     else                                            // Is it in the bottom half?
                         last = --mid;                               // reset the last value
                       mid  = (first + last)/2;                      // set new middle value
                 } }}}}
```

Let's follow the variable values first if the user enters the numeric value 3 (a legal value) and then 7 (a value not on the list). Note that whatever integer the user enters, it will be stored in location (variable) ***value***.

value = 3

Table 9.10.

Pass	*first*	*last*	*mid*	*myarray* [*mid*]	*value* == *myarray* [*mid*]?	*value* > *myarray* [*mid*]?	*value* < *myarray* [*mid*]?	*last* >= *first* ?
0	0	7	3	4	NO	NO	YES: *last* = 2	NO
1	0	2	1	2	NO	YES: *first* = 2	NO	NO
2	2	2	2	3	YES: *found* = 1			

Now, if the user were to enter the value 7:

value = 7

Table 9.11.

Pass	*first*	*last*	*mid*	*myarray* [*mid*]	*value* == *myarray* [*mid*]?	*value* > *myarray* [*mid*]?	*value* < *myarray* [*mid*]?	*last* >= *first* ?
0	0	7	3	4	NO	YES: *first* = 5	NO	NO
1	5	7	6	10	NO	NO	YES: *last* = 5	NO
2	5	5	5	6	NO	YES: *first* = 7	NO	YES

The search ends because the contents of location *last* are less than the contents of location *first* (i.e., 5 < 7).

In terms of what the program would print out if we entered various values, we would find:

Program 9.2. Output

```
Enter a (whole) number between 1 and 12 (0 to quit):      6
value = 6, middle = 3, last = 7, first = 0
value = 6, middle = 5, last = 7, first = 4
value = 6, middle = 4, last = 4, first = 4
The number 6 was found in position 4

Enter a (whole) number between 1 and 12 (0 to quit):      5
value = 5, middle = 3, last = 7, first = 0
value = 5, middle = 5, last = 7, first = 4
value = 5, middle = 4, last = 4, first = 4
The value 5 is not in the array

Enter a (whole) number between 1 and 12 (0 to quit):      11
value = 11, middle = 3, last = 7, first = 0
value = 11, middle = 5, last = 7, first = 4
value = 11, middle = 6, last = 7, first = 6
value = 11, middle = 7, last = 7, first = 7
The value 11 is not in the array


Enter a (whole) number between 1 and 12 (0 to quit):      0
```

If we were to enter all the values between 1 and 13, we would find:

Table 9.12.

Search Value	No. Comparisons	Search Value	No.
1	3	8	3
2	2	9	2
3	3	10	3
4	1	11	3
5	3	12	3
6	3	13	3
7	3		

Notice that for any value greater than 12, the number of comparisons necessary is 3. This corresponds to our formula (9.1.)

Maximum Number Comparisons $= \lceil \log_2 n \rceil = \lceil \log_2 8 \rceil = \lceil 3.00 \rceil = 3$

Does that mean that we must Always sort a list if we want to perform a binary search???

Basically, yes. However, there are some alternatives. One which we can discuss here is the concept of an **index** (much like the index of a textbook). The original data is left unsorted, but we construct an alternative list (the index) which is ordered and points to the unsorted list. We can then use a binary search on the index to find elements in the unsorted list.

Let's consider a new example. Consider the following list of names:

Table 9.13.

Offset	Name
0	Lincoln, A.
1	Washington, G.
2	Fillmore, M.
3	Nixon, R.
4	Van Buren, M.
5	Kennedy, J.F.
6	Adams, J.
7	Eisenhower, D.D.
8	Reagan, R.
9	Johnson, L.B.

This list is obviously not in any order (alphabetically or chronologically). Let's assume that we had initialized the array as:

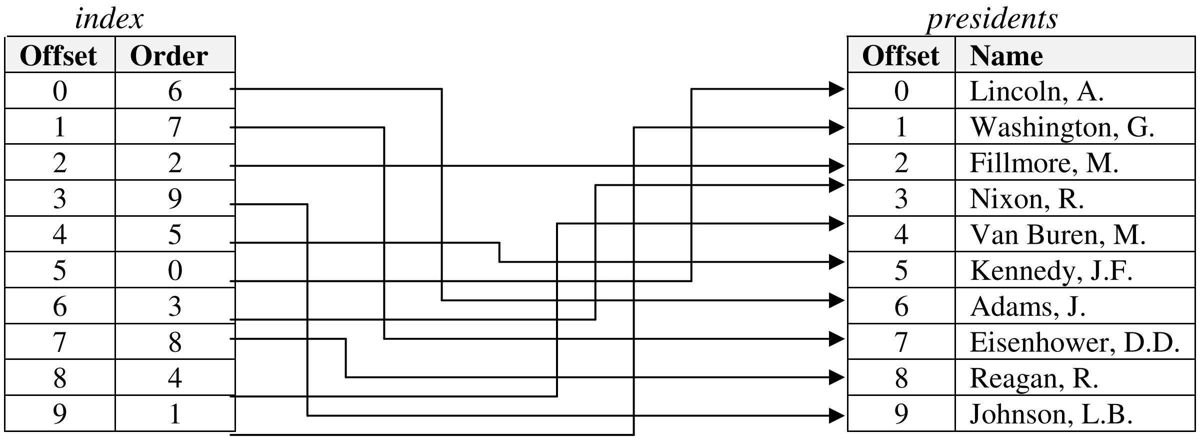
char *presidents*[10][20] = {…};

Remember: this array consists of 10 elements with each element containing 20 characters (a two dimensional array).

If we wished to print the array alphabetically, we would first have to print out the string found at offset 6 ("Adams, J."), then the string found at offset 7 ("Eisenhower, D.D."), then at offset 2 ("Fillmore, M."), and so forth, until we print out the last name on the list ("Washington, G.") which is found at offset 1.

If we were to construct another array, say: **int** *index*[10]; we could store the offsets of ordered list of presidents in it. The relationship between the two lists might appear as:

Figure 9.10.



index

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

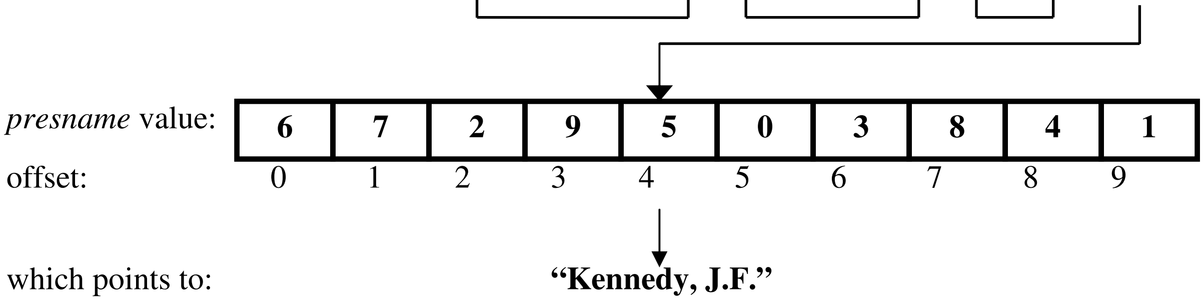
presidents

Offset	Name
0	Lincoln, A.
1	Washington, G.
2	Fillmore, M.
3	Nixon, R.
4	Van Buren, M.
5	Kennedy, J.F.
6	Adams, J.
7	Eisenhower, D.D.
8	Reagan, R.
9	Johnson, L.B.

Notice that while we have not changed the order of our *presidents* array, we could quickly find any name on the list by performing a binary search on our *index* array. Let's suppose we wished to find the name "Reagan, R." on our list. We could essentially proceed as before:

Figure 9.11.

Search 1: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (0 + 9)/ 2 \rfloor = \lfloor 4.5 \rfloor = 4$

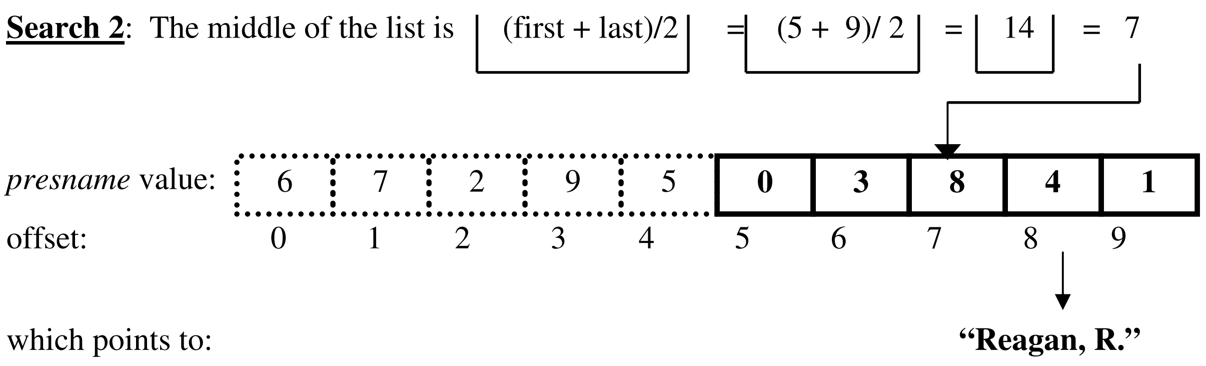


presname value:	**6**	**7**	**2**	**9**	**5**	**0**	**3**	**8**	**4**	**1**
offset:	0	1	2	3	4	5	6	7	8	9

which points to: **"Kennedy, J.F."**

Since "Kennedy, J.F." is alphabetically less than "Reagan, R.", we know that if "Reagan, R." exists on the list, it must be in positions 5, 6, 7, 8, or 9 (in array *index*) which point to names greater than "Kennedy, J.F." in array *presidents*. Since the first place it could be is in position 5, we reset our first position to 5. The next search would appear as (Figure 9.12):

Figure 9.12.


Search 2: The middle of the list is ⌊(first + last)/2⌋ = ⌊(5 + 9)/ 2⌋ = ⌊14⌋ = 7
presname value:
6
7
2
9
5
0
3
8
4
1
offset:
0
1
2
3
4
5
6
7
8
9
which points to:
"Reagan, R."

Notice that we are done with our search after only two comparisons (instead of the 8 it would have taken with a sequential search).

Using an index does <u>not</u> seem life it is much of an improvement over simply sorting the list of names!!!

In this case, it might not be. But we will review this approach a little later, and we will find that can offer significant benefits.

What about the c code necessary to perform a binary Search using an index???

The code is very similar to the code we saw for a simple binary search. Consider the code in program 9.3. Basically, the only differences are:

1. Our variables first, last, and mid are applied to array index (instead of being directly applied to array presidents).
2. When we compare the value we are looking for (in this program, variable search) with the name string in the array (presidents), instead of using the subscript (offset) mid, we use the offset contained in the array prenames. That is, instead of issuing the command[2]:

   if (strcmp(*search*, *presidents*[*mid*]) == 0)

   we *MUST* issue the command:

   if (strcmp(*search*, *presidents*[*index*[*mid*]]) == 0)

where we check the offset of the *presidents* array via the contents of our *index* array.

[2] Notice that since we are dealing with strings, we must use the strcmp function

Let's follow our variable values if we were to look for the string "Washington, G." on the list (Table 9.14):

Table 9.14.

Pass	***first***	***last***	***mid***	***index[mid]***	***president [index[mid]]***	***search*[3] == *president [index[mid]]*?**	***search > president [index[mid]]*?**	***search < president [index[mid]]*?**
0	0	9	4	5	"Kennedy, J.F.	NO	YES: *first* = 5	NO
1	5	9	7	8	"Reagan, R."	NO	YES: *first* = 8	NO
2	8	9	8	4	"Van Buren, M."	NO	YES: *first* = 9	NO
1	5	9	7	8	"Reagan, R."	NO	YES: *first* = 8	NO
3	9	9	9	1	"Washington, G."	YES: *found* = 1		

We leave it to you to determine what would happen if we were searching for a name that was not on the list.

C program 9.3

```
#include <stdio.h>                                              // Std. I/O
#include <string.h>                                             // for strcmp, strlen
 void main()
{  int  first, last, mid, found,                                // our offset pionters
        index[10] = {6, 7, 2, 9, 5, 0, 3, 8, 4, 1};             // our index
   char presidents[10][20] = {"Lincoln, A.", "Washington, G.", "Fillmore, M.",
                              "Nixon, R.", "Van Buren, M.", Kennedy, J.F.", "Adams, J.",
                              "Eisenhower, D.D.", "Reagan, R.", Johnson, L.B."},
        search[10] = "XXX";                                     // Search Name
   while (strlen(search) > 0)                                   // Continue until Quit
   {   printf("\nEnter a name (CR to quit): ");                 // Prompt the user
       if (strlen(search) > 0)                                  // entering CR => len = 0
       {   first = 0; last = 9; mid = (first + last)/2; found = 1;  // set staring values
           while(found != 0)
           {   printf("name = %s, middle = %d, last = %d, first = %d\n", search, mid, last, first);
               if (strcmp(search, presidents[index[mid]]) == 0)     // Was the name found?
              {   printf("The name %s was found in position %d\n", search, index[mid]);
                  found = 0;   }                                // set the found flag
               else                                             // if not found
                  if (first  >= last)                           // NOT in the array??
                  {  printf("The name %s is not in the array\n", search);  // Then indicate
                     found = 0; }                               // and set the flag
                  else
                  {    if (strcmp(search, presidents[index[mid]]) > 0)  // Is it in the top half?
                           first = ++mid;                       // set the first value
                       else                                     // Is it in the bottom half?
                         last = --mid;                          // reset the last value
                           mid  = (first + last)/2;  }          // set new middle value
                  }  }  }  }
```

Tradeoffs Between Sorted and Unsorted Lists

As we implied, there are trade-offs between sorted and unsorted lists. If we don't really care about finding an individual element in a list, or about displaying elements in a specified order, then unsorted lists might be preferred. Certainly, for many transaction processing systems, which are processed in order of receipt and in batch, such an approach is preferred. Updating and maintenance are simple and not programatically complex or expensive.

If being able to locate specific elements, or records in a database, and being able to locate them quickly, is a concern, then more advanced searching procedures must be applied. The approach that we discussed previously, a binary search, provides the speed necessary (by the way, it is also the fastest search method). However, as we have seen, if we wish to perform a binary search, then we *must* first sort the list. Even if we apply the index approach discussed above, we *must* first sort the index.

While sorted list provide us with searching advantages, there are some associated problems:

- **Initial Sorting of the elements can be time consuming** (as we shall see in the next section). This is especially true if we are trying to sort extremely large data sets (which would necessarily involve external sorting (mentioned in the following section) procedures).

- **Maintaining the sorted list requires some effort**. Each time a new element is added, it must be inserted in the correct position and the other elements repositioned accordingly. Each time an element is deleted from the list, the remaining elements must be rearranged.

- **The programs needed to find an element are more complex and require more execution time than do sequential search programs**. Compare the code needed for a binary search versus a sequential search.

- **A true binary search requires all the elements (or index of the elements) to be stored in as an array in contiguous locations in memory**. For large arrays, this may be not be feasible (e.g., assume an array of structs containing 100,000 elements, where each struct requires 10,000 bytes of storage. We would need 1,000,000,000 (1 billion) bytes of storage).

This last constraint, namely that we use contiguous storage in RAM is one which we shall loosen in the subsequent chapters. However, even when we do, the other concerns will remain, and in many cases, will be intensified.

Sorting

Sorting algorithms are a topic of great concern to computer scientists, who are keenly interested in maximizing efficiency. We will discuss them very briefly, and consider only two of the most common approaches.

There are two general classes of sorts: Internal and External. External sorts are those which are performed outside of RAM on the disk. External sorts are much slower than internal sorts, primarily because the can require extensive reading and writing to and from the external storage device, which is the slowest operation a computer performs. However, in cases where we are dealing with large data sets (too large to fit into RAM), we have no choice. In COBOL, a language designed to deal with large data sets, there are built-in external sorting procedures.

Internal sorts are those where the entire list is placed in memory and sorted. In fact, all sorting (whether internal or external) is performed by the CPU, and thus data must first be placed in RAM. However, with external sorts, only a portion of data will be placed in memory at any one time. We will consider only internal sorts in this section.

There are three major type of internal sorts:

1. **Exchange Sorts** (e.g., bubble sort). In these techniques, a single (usually; multiple lists may be employed), we scan the list for elements which are not in the correct order, exchanging their positions as found.

2. **Selection Sorts.** In this approach, there are generally two (or more) lists (Selection sorts with exchange use only one list). The basic methodology is to **select** the largest and smallest elements in each pass and move them into their correct position.

3. **Insertion Sorts.** This approach also can employ one or two lists, although two are more common. Each item from original list is **inserted** into the correct position in the new list as they are examined.

Once again, since our intent is essentially to illustrate some of the issues involved in sorting, we will consider only the two most common exchange sorts: **Bubble Sorts**, perhaps the most common and certainly the simplest off all sorts, and the **Quick Sort**, generally the fastest sort, especially with longer lists. The quick sort will be shown only in graphical format at this time since it involves a programming technique (recursion) which will be cover in a later chapter.

Bubble Sorts

Bubble sorts are perhaps the simplest of all sorting (internal/exchange) techniques, and for (relatively) small lists, are more than adequate. The basic concept behind bubble sorts is to continue examining the list of data and letting the largest (or smallest) elements bubble to the top (or bottom) of the list with each pass.

Consider our previous (unsorted) list, *myarray*:

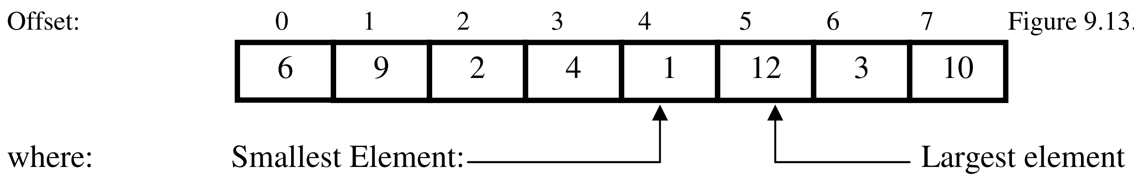


Figure 9.13.

It is obvious that this element (12), presently located in position *myarray*[5], should be in the position (*myarray*[7]). Further, the second largest element (10 in *myarray*[7]) should be moved to *myarray*[6], and so on. The smallest element (1), presently located in position myarray[4], should be in position *myarray*[0]. In fact, we know that the sorted array shouls appear as:

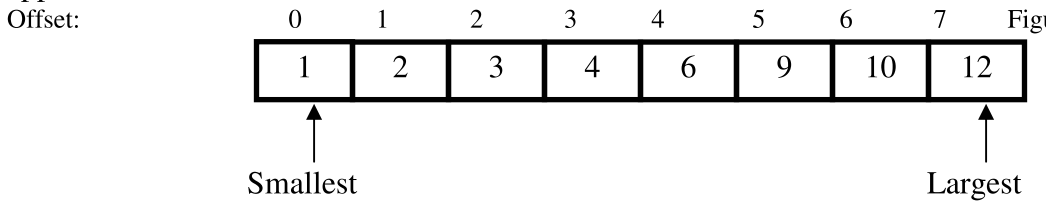


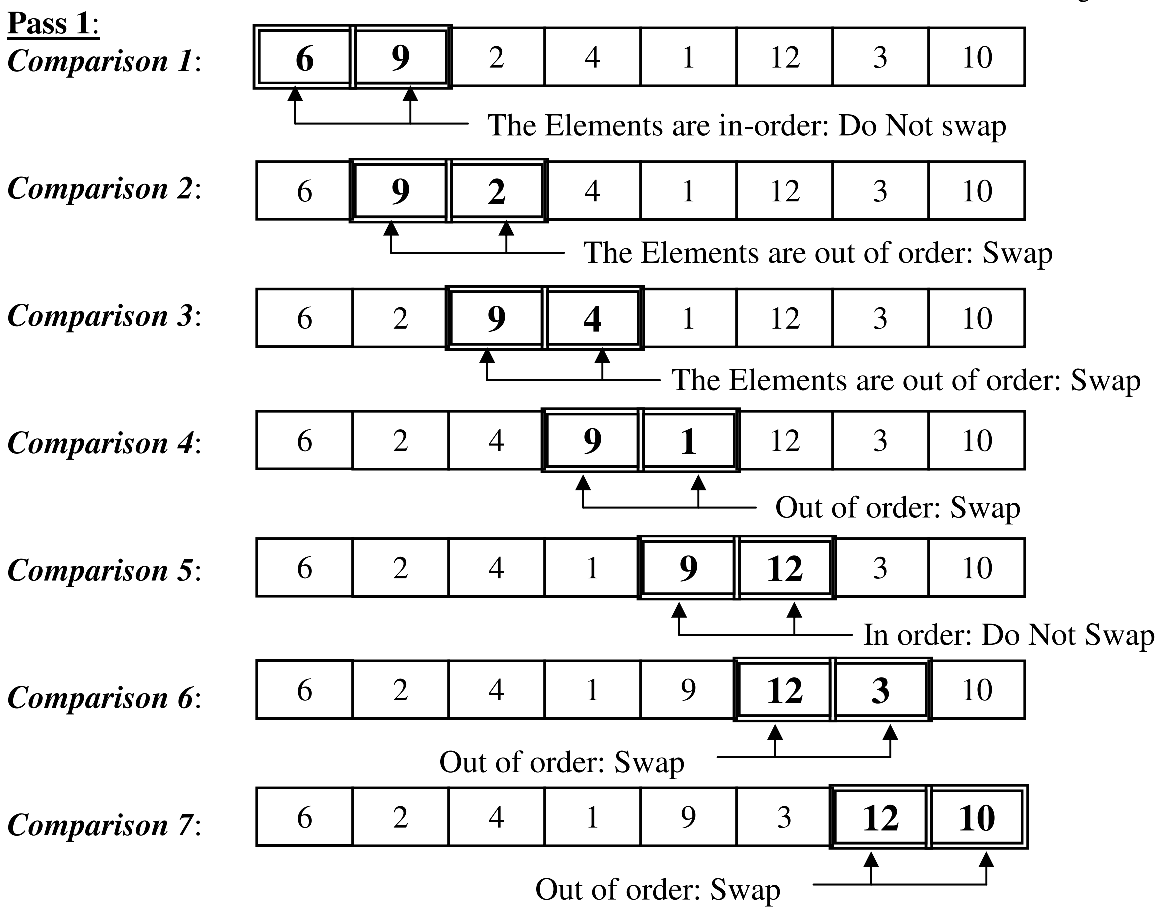
Figure 9.14.

This logic forms the logic for our algorithm (we will modify it a little later on):

1. Starting from the bottom (or top), compare each element in the array with the adjacent element in the array.
2. If the elements are in correct order, swap them.
3. Continue comparing the elements in the array until the largest element is at the top (or the smallest element is at the bottom).
4. Reduce the length of the list to be considered by 1 (either ignore the largest element in the list or the smallest element in the list).
5. If the length of the list is 1 (one), stop. Otherwise, go to step 1 and repeat the procedure.

Let's see how the algorithm actually works (see Figure 9.15.). Starting at the bottom of the list, we compare the 1st element with 2nd, the 2nd with the 3rd, an so forth, until we have compared the 7th with the 8th. IF any comparison shows that the elements are out of order, we swap them. IF they are in order, we continue.

Figure 9.15



At the end of the 1st pass, our list would appear as:

6	2	4	1	9	3	10	12

? ***What do we know???***

1. The largest element on the list has ‘bubbled-up’ to the top of the list.
2. We had to make a total of 7 comparisons, given that there are 8 elements on the list.

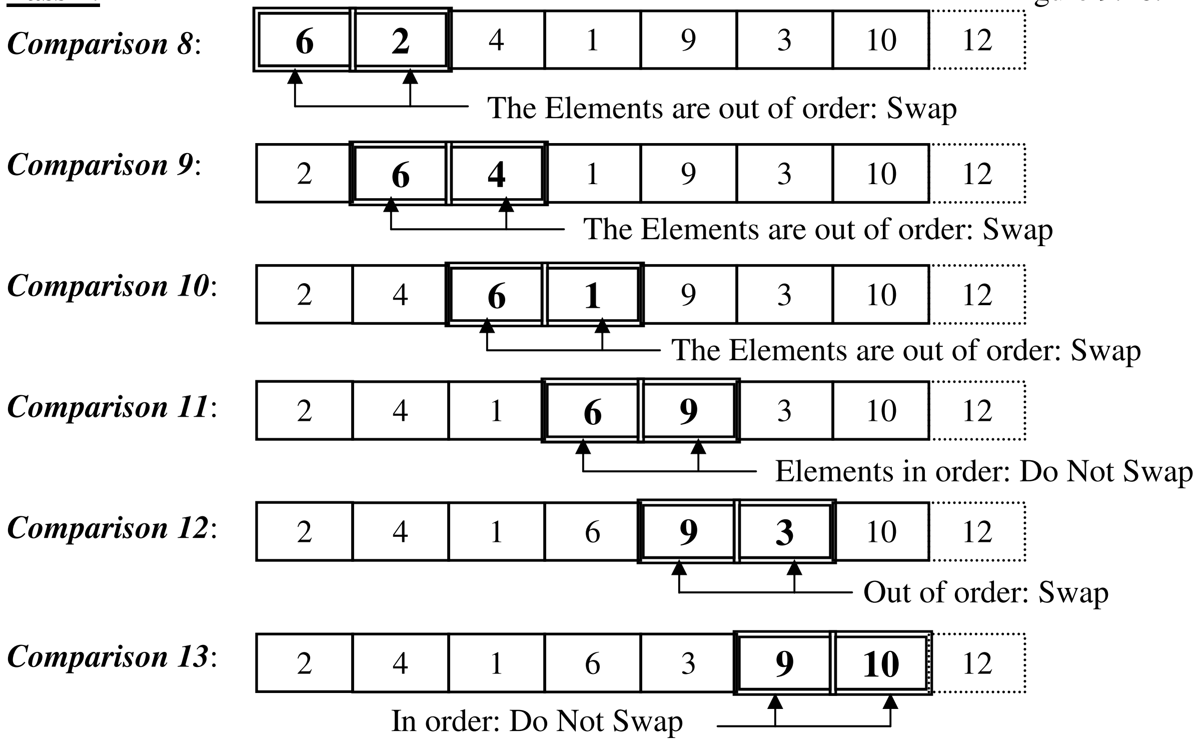
   In fact, when we make pair-wise comparisons, for a list of n elements (in this case, 8), we need to make n-1 total comparisons.

3. Some of the elements have moved closer to their ‘true’ positions (although we can’t always be sure of this).

? ***Are we done???***

Obviously not. BUT, since we know that the largest element is at the end of the list, for our next pass, we need not compare it with the element beneath it. In other words, we need compare the *only* the elements in offsets 0 through 6 (7 elements).

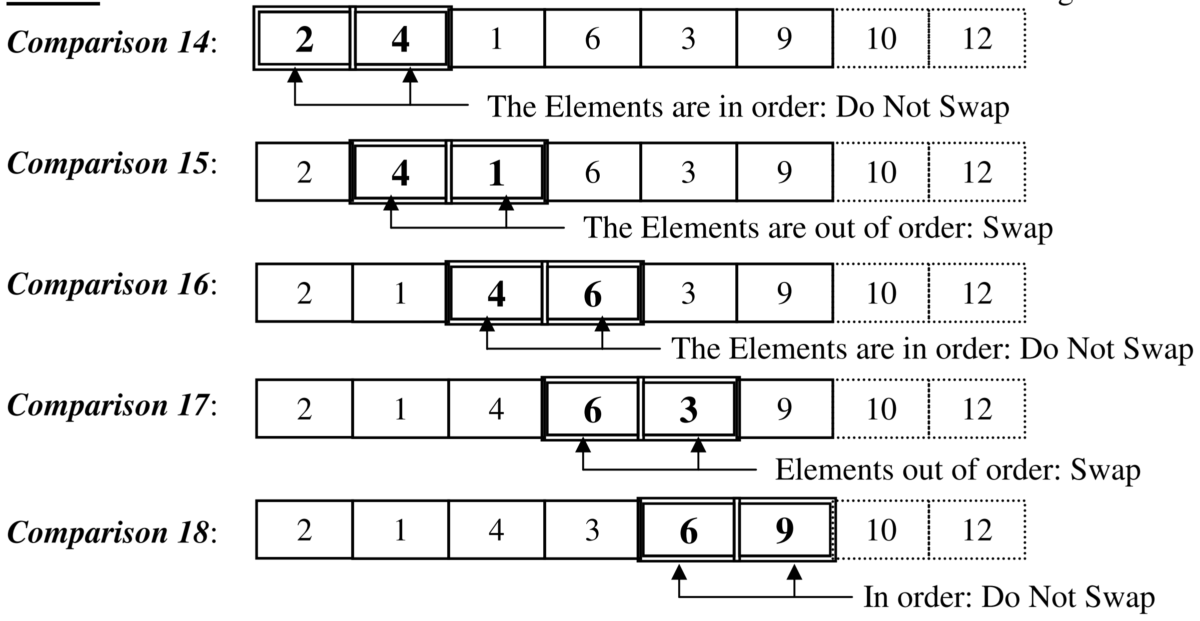
<u>Pass 2</u>: Figure 9.16.



Comparison 8: 6 2 4 1 9 3 10 12
The Elements are out of order: Swap
Comparison 9: 2 6 4 1 9 3 10 12
The Elements are out of order: Swap
Comparison 10: 2 4 6 1 9 3 10 12
The Elements are out of order: Swap
Comparison 11: 2 4 1 6 9 3 10 12
Elements in order: Do Not Swap
Comparison 12: 2 4 1 6 9 3 10 12
Out of order: Swap
Comparison 13: 2 4 1 6 3 9 10 12
In order: Do Not Swap

Notice that this time, we only made 6 comparisons. Also, we know with certainty that the *two* largest elements occupy the last two positions, so we need not check them in the next pass. Continuing:

<u>Pass 3</u>: Figure 9.17.



Comparison 14: 2 4 1 6 3 9 10 12
The Elements are in order: Do Not Swap
Comparison 15: 2 4 1 6 3 9 10 12
The Elements are out of order: Swap
Comparison 16: 2 1 4 6 3 9 10 12
The Elements are in order: Do Not Swap
Comparison 17: 2 1 4 6 3 9 10 12
Elements out of order: Swap
Comparison 18: 2 1 4 3 6 9 10 12
In order: Do Not Swap

And we know for sure that the top three elements are in place.

Pass 4: Figure 9.18.

Comparison 19:

2	**1**	4	3	6	9	10	12

The Elements are out of order: Swap

Comparison 20:

1	**2**	**4**	3	6	9	10	12

The Elements are in order: Do Not Swap

Comparison 21:

1	2	**4**	**3**	6	9	10	12

The Elements are out of order: Swap

Comparison 22:

1	2	3	**4**	**6**	9	10	12

Elements in order: Do Not Swap

And the list (after Comparison 22) is in order.

? ***Do we know that a list of 8 elements will be sorted after 22 comparisons???***

No. Not at all. Consider the worst case scenario (when the list is in ***reverse*** order):

Figure 9.19.

Comp: Pass 1:

1	**12**	**10**	9	6	4	3	2	1
2	10	**12**	**9**	6	4	3	2	1
3	10	9	**12**	**6**	4	3	2	1
4	10	9	6	**12**	**4**	3	2	1
5	10	9	6	4	**12**	**3**	2	1
6	10	9	6	4	3	**12**	**2**	1
7	10	9	6	4	3	2	**12**	**1**

Pass 2:

8	**10**	**9**	6	4	3	2	1	12
9	9	**10**	**6**	4	3	2	1	12
10	9	6	**10**	**4**	3	2	1	12
11	9	6	4	**10**	**3**	2	1	12
12	9	6	4	3	**10**	**2**	1	12
13	9	6	4	3	2	**10**	**1**	12

And the list will be sorted after we swap 2 & 1 in pass 7, comparison 28

Comp:

Pass 3:

14	**9**	**6**	4	3	2	1	10	12
15	6	**9**	**4**	3	2	1	10	12
16	6	4	**9**	**3**	2	1	10	12
17	6	4	3	**9**	**2**	1	10	12
18	6	4	3	2	**9**	**1**	10	12

Pass 4:

19	**6**	**4**	3	2	1	9	10	12
20	4	**6**	**3**	2	1	9	10	12
21	4	3	**6**	**2**	1	9	10	12
22	4	3	2	**6**	**1**	9	10	12

Pass 5:

23	**4**	**3**	2	1	6	9	10	12
24	3	**4**	**2**	1	6	9	10	12
25	3	2	**4**	**1**	6	9	10	12

Pass 6:

26	**3**	**2**	1	4	6	9	10	12
27	2	**3**	**1**	4	6	9	10	12

Pass 7:

28	**2**	**1**	3	4	6	9	10	12

? *What do we know???*

- For an array of size n, we need n-1 passes (i.e., for an array of 8 elements, we need 7 passes) to be completely sure that we have sorted the list
- For the first pass, we need n-1 comparisons; each subsequent pass needs 1 less comparison than the previous one. In our list, we needed:

Table 9.15.

Pass	Comparisons	Pass	Comparisons
1	7	5	3
2	6	6	2
3	5	7	1
4	4		

In other words, we need a total of :

$\Sigma[(n-1)+(n-2)+...+1]$

$= 7 + 6 + 5 + 4 + 3 + 2 + 1 =$ **<u>28</u>** Comparisons

Or: $(n^2 - n)/2$ $= (8^2 - 8)/2 = (64 - 8)/2 = 56/2 =$ **<u>28</u>**

? *Is that a lot???*

It can be. Consider the following list lengths:

Table 9.16.

Elements	Max. Passes (n – 1)	Max Compares $(n^2 - n)/2$	Elements	Max. Passes (n – 1)	Max Compares $(n^2 - n)/2$
10	9	45	100,000	99,999	4,999,950,000
100	99	4,950	250,000	249,999	31,249,875,000
1,000	999	999,500	500,000	499,999	124,999,750,000
5,000	4,999	12,497,500	1,000,000	999,999	499,999,500,000
10,000	9,999	49,995,000	5,000,000	4,999.999	12,499,997,500,000

For a company like American Express, with 300 million customers, the (maximum) number of comparisons is $4.9999985 * 10^{16}$ (= 44,999,999,850,000,000)

? **But, in our original example, the list was sorted after 4 passes and 22 Comparisons (see figure 9.18) . Couldn't we stop there*???***

Yes and no. The list is indeed sorted after pass 4, but we don't know it. We know that a list is sorted *IF* we make the maximum (n-1) passes *OR* we make a pass *AND* we make <u>no</u> swaps. If we put a flag in our program which checked to see if we made a swap, we could have stopped *AFTER* pass 5 (Notice: we *DID* make a swap in pass 4, but not in pass 5). It does mean that we must make one extra pass (i.e., even if the list were sorted to begin with, we would have to make one pass through it), but if often saves us from having to make the maximum number of passes.

Is there any way to improve on the bubble sort???

When we examined our bubble sort, we noticed that not only did the larger (and certainly largest) numbers 'bubble' up, but in doing so, the smaller numbers were pushed down. We can also have a two-way bubble sort: in one pass we 'bubble' the larger numbers up, and in the subsequent pass, we 'bubble' the smaller numbers down.

How does this work???

Consider our original (unsorted) list:

Figure 9.20.

<u>Comp</u>: <u>Pass 1 (Bubble up):</u>

1	**6**	**9**	2	4	1	12	3	10
2	6	**9**	**2**	4	1	12	3	10
3	6	2	**9**	**4**	1	12	3	10
4	6	2	4	**9**	**1**	12	3	10
5	6	2	4	1	**9**	**12**	3	10
6	6	2	4	1	9	**12**	**3**	10
7	6	2	4	1	9	3	**12**	**10**

<u>Pass 2 (Bubble down):</u>

8	6	2	4	1	9	**3**	**10**	12
9	6	2	4	1	**9**	**3**	10	12
10	6	2	4	**1**	**3**	9	10	12
11	6	2	**4**	**1**	3	9	10	12
12	6	**2**	**1**	4	3	9	10	12
13	**6**	**1**	2	4	3	9	10	12

<u>Comp:</u> <u>Pass 3 (Bubble up):</u>

14	1	**6**	**2**	4	3	9	10	12
15	1	2	**6**	**4**	3	9	10	12
16	1	2	**4**	**6**	**3**	9	10	12
17	1	2	4	3	**6**	**9**	10	12
18	1	2	4	3	6	**9**	**10**	12

<u>Pass 4 (Bubble down):</u>

19	1	2	4	3	**6**	**9**	10	12
20	1	2	4	**3**	**6**	9	10	12
21	1	2	**4**	**3**	6	9	10	12
22	1	**2**	**3**	4	6	9	10	12

And the list will be sorted when we swap Elements after comparison 21.

<u>NOTE</u>: we still MUST make one more Pass to know that the list is sorted

Notice that after comparison 13, the smallest is known to be at the bottom of the list

This does not seem like any savings. With a simple 'bubble' up, the list was sorted after the 22nd comparison.

In this case, there was essentially no savings. If we were looking at the worst case scenario (when the list is in reverse order), there will be absolutely no savings. However, most of the time, the total number of comparisons will be less.

? ***What about the c code necessary to perform the bubble sort above???***

The code is provided in Program 9.4. The program initially ‘bubbles’ up, then ‘bubbles’ down, then repeats the process. It includes the flag sorted (an integer variable): if no swaps are made, the value is set to 0, otherwise the value is set to 1. If no stops are made, we can stop. The output which the program produces follows the source code.

C program 9.4.

```
#include <stdio.h>
void printarray(int a[], int head);                        // prototype
int  pass = 0, compare = 0, swaps = 0;                     // global vars
void main()
{  int i, temp, iarray[8] = {6,9,2,4,1,12,3,10},          // initialize
   int sorted = 1, bottom = 0, top = 7;                    // Sorted Flag
   printarray(iarray,0);                                   // print original
   while ((top > bottom) && (sorted == 1))                 // check end
 {     sorted = 0;                                         // assume in order
       pass++;                                             // increment counter
       for (i = bottom; i < top; i++)                      // move largest up
      {     compare++;                                     // increment counter
            if (iarray[i] > iarray[i+1])                   // ?? out of order
           {     sorted = 1; swaps++;                      // array NOT in orde
                 temp = iarray[i];                         // temp. storage
                iarray[i] = iarray[i+1];                   // swap
                iarray[i+1] = temp;    }
                printarray(iarray,1);      }               // print results
           top--;                                          // decrement top
     if (sorted == 1)                                      // array in order ??
     {    sorted = 0; pass++;                              // reset sorted flag
         for (i = top; i > bottom; i--)                    // move smallest down
     {   compare++;                                        // increment counter
         if (iarray[i] < iarray[i-1])                      // ?? out of order
        {   sorted = 1;                                    // array NOT in order
            swaps++;                                       // increment counter
            temp = iarray[i];                              // temp. storage
            iarray[i] = iarray[i-1];                       // swap
            iarray[i-1] = temp;   }
        printarray(iarray,1);        }                     // print results
    bottom++;                        }  }  }               // inc. array position
 void printarray(int a[], int head)                        // print array
{  int i;                                                  // index
   if (head == 0) printf("Original Series:  ");            // check header
   else  printf("Pass %d, Compares %d, Swaps %d: ",pass,compare,swaps);
   for (i = 0; i < 10; i++) printf("%3d", a[i]);           // Print element
   printf("\n");                                           // add newline
}
```

The output from the above program would appear as:

C Program 9.4. Output

```
Original Series:                6  9  2  4  1 12  3 10
Pass 1, Compares 1, Swaps 0:    6  9  2  4  1 12  3 10
Pass 1, Compares 2, Swaps 1:    6  2  9  4  1 12  3 10
Pass 1, Compares 3, Swaps 2:    6  2  4  9  1 12  3 10
Pass 1, Compares 4, Swaps 3:    6  2  4  1  9 12  3 10
Pass 1, Compares 5, Swaps 3:    6  2  4  1  9 12  3 10
Pass 1, Compares 6, Swaps 4:    6  2  4  1  9  3 12 10
Pass 1, Compares 7, Swaps 5:    6  2  4  1  9  3 10 12
Pass 2, Compares 8, Swaps 5:    6  2  4  1  9  3 10 12
Pass 2, Compares 9, Swaps 6:    6  2  4  1  3  9 10 12
Pass 2, Compares 10, Swaps 6:   6  2  4  1  3  9 10 12
Pass 2, Compares 11, Swaps 7:   6  2  1  4  3  9 10 12
Pass 2, Compares 12, Swaps 8:   6  1  2  4  3  9 10 12
Pass 2, Compares 13, Swaps 9:   1  6  2  4  3  9 10 12
Pass 3, Compares 14, Swaps 10:  1  2  6  4  3  9 10 12
Pass 3, Compares 15, Swaps 11:  1  2  4  6  3  9 10 12
Pass 3, Compares 16, Swaps 12:  1  2  4  3  6  9 10 12
Pass 3, Compares 17, Swaps 12:  1  2  4  3  6  9 10 12
Pass 3, Compares 18, Swaps 12:  1  2  4  3  6  9 10 12
Pass 4, Compares 19, Swaps 12:  1  2  4  3  6  9 10 12
Pass 4, Compares 20, Swaps 12:  1  2  4  3  6  9 10 12
Pass 4, Compares 21, Swaps 13:  1  2  3  4  6  9 10 12
Pass 4, Compares 22, Swaps 13:  1  2  3  4  6  9 10 12
Pass 5, Compares 23, Swaps 13:  1  2  3  4  6  9 10 12
Pass 5, Compares 24, Swaps 13:  1  2  3  4  6  9 10 12
Pass 5, Compares 25, Swaps 13:  1  2  3  4  6  9 10 12
```

Which corresponds to our illustration, except that the swaps are made before the comparison number is printed (e.g., the list is sorted after comparison 21).

? ***Is the bubble sort the quickest search technique???***

No, not at all. It may be adequate for smaller lists, but for larger lists, it is quite slow. The Quick sort technique is generally considered the fasted internal sort technique, especially for larger lists.

Quick Sorts

We noticed that the smaller the list, the faster the sort. For example, we saw in Table 9.16. that if a list consisted of 100 elements, the maximum number of comparisons $((n^2 - n)/2)$ was 4,950. If the list was 10 times larger (1,000 elements), the maximum number of comparisons needed was 499,500 (a 100 fold increase). In other words, as the list length increases, the number of comparisons needed increases *exponentially*. The idea behind a quick sort is to break the list up into smaller lists, sort the smaller lists, and then merge the lists together.

Consider the following list consisting of 10 elements:

Figure 9.21.

7	**2**	**6**	**9**	**4**	**3**	**8**	**10**	**1**	**5**

The first step is to pick a **pivot** element which will allow us to break the list in half. Ideally, we would like to pick the element which contains the median value, but we do not know that in advance. For our example let's choose the median element on the list (i.e., the 5th element in the list, or the element in offset 4, or the numeric value 4).

The next step is to **partition** the list. By this we mean that we start comparing elements on the list with our pivot element (the value 4): If the number is greater, we will move it to the right of our pivot element. If the number is less than 4, we will move it to the left of the pivot element.

We start by comparing the leftmost element with the rightmost element, swapping as necessary:

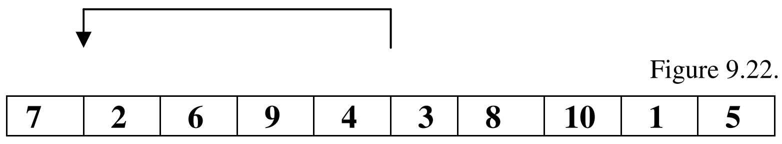
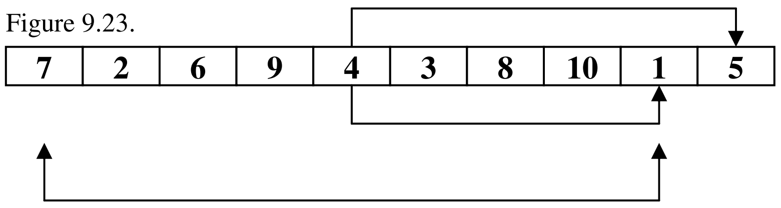


Figure 9.22.

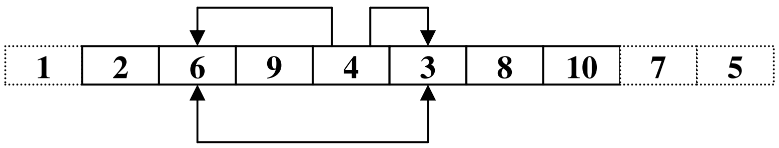
The value 7 (in offset 0) is NOT In the correct position (it should be to the right of the pivot element). Now we need to find an element which is to the right of the pivot element BUT should be to the left of the pivot element. When we do, we can swap the two misplaced elements.

Figure 9.23.



Since 5 is greater than 4, it is positioned correctly (i.e., to the right of 4) we want to leave it. But since 1 is not (it should be to the left of the pivot), we will swap it with 7 (relative to 4, both will be in the correct position).

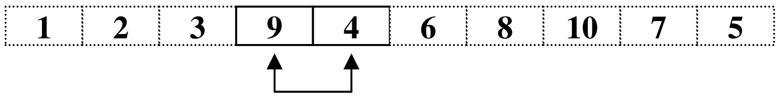
After the swap, the array appears as: Figure 9.24.



Continuing as before, we can see that the 2 (in offset 1) is correctly positioned, but the 6 (in offset 2) is not.

To the right of the pivot element, the 10 (offset 7) and the 8 (offset 6) are greater than the pivot element, and should remain where they are. The 3 (offset 5) is not in the correct position, and should be swapped with the 6 (in offset 2).

Figure 9.25.



At this point, we have only one element left (the 9 at offset 3). Since the 9 should be to the right of the pivot element, we must swap the pivot element with the 9. We now have 2 lists:

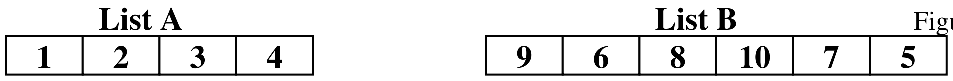


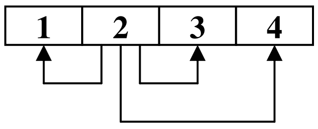
Figure 9.26.

We further know that the smaller numbers are in (partitioned) List A and the larger numbers in (partitioned) List B.

? ***Now what???***

We repeat the procedure above on each of the lists. Starting with List A, we once again pick a pivot element (let’s again choose the midpoint element, or the element in offset 1 (the number 2)), and see where it should be in the list (relative to the other elements).

Figure 9.27.



When we check the position of our pivot element relative to all of the other elements in the array, we find that we do not need to make any swaps. Furthermore, since there is only one element to the left of the pivot element, and the element is smaller than the
pivot element, we know that these two (elements 1 and 2) are in the correct position. We now have two new sublists:

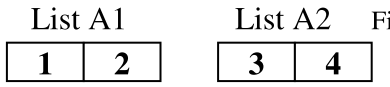


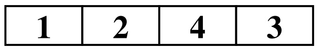
Figure 9.28.

And we know that sublist A1 is sorted.

? ***Wait, BOTH sublists are sorted!!!***

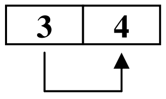
Figure 9.29.

But we don't know that. Suppose our original List A were:



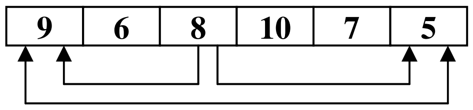
Relative to our pivot element (the value 2), all of the elements are in the correct position. However, we can see that the partitioned sublist containing the elements 4 and 3 is not in order.

Figure 9.30.



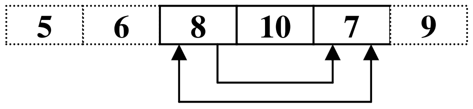
We must therefore examine the partitioned sublist A2. Again we must first choose a pivot element (let's say the value 3 at offset 0). Comparing it with the only other element, we find that are in the correct positions. Since the list only contains two elements, and they are in the correct positions, NOW we know the list is sorted.

Figure 9.31.



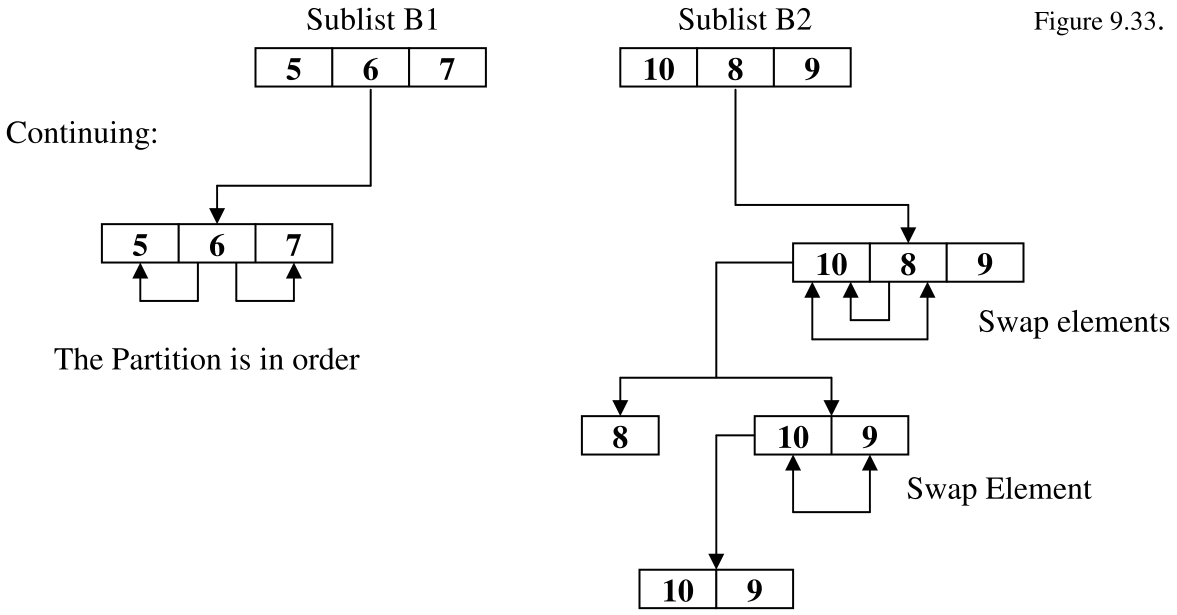
We must now perform the same activities with List B. Again, choosing the midpoint of the array as our pivot element (the value 8 in offset 2), we begin comparing the elements, beginning with the outer-most ones first. We can see that the leftmost element (the number 9) and the rightmost element (the value 5) are incorrectly positioned, and should be swapped.

Figure 9.32.



Since all of the elements to the right of the pivot element are in their correct positions, we must check to see if we can swap the pivot element with an element to the right of it. In this case we know that the 7 (in offset 4) should be to the right of our pivot element, and so we swap the two. Once again, we end up with two partitioned sublists:

Figure 9.33.



? ***This seems very complex, and hardly worth it!!!***

It is more complex, and the code necessary to perform a quick sort takes more time to execute than does the code for a bubble sort. But the number of comparisons necessary, especially for longer lists, is considerably less also. For our example, it took 22 comparisons and 7 swaps (vs. the 30 comparisons and 15 swaps it would have taken with a two-way bubble sort with checks).

For a quick sort, the maximum number of comparisons is: $\lceil \log_2 n! \rceil$ Formula 9.3.
Where n is the number of elements on the list.

Consider the following comparison with the bubble sort technique:

Table 9.17.

No. Elements	Max. Bubble Sort Compares: $(n^2 - n)/2$	Max. Quick Sort Compares: $\lceil \log_2 n! \rceil$
10	45	22
100	4,950	525
1,000	999,500	9,965
10,000	49,995,000	132,877

A significant difference.

? ***What about the c code necessary for a quick sort???***

The code necessary to perform a quick sort relies on a technique called recursion (where a function calls itself). We are not quite ready for it yet. However, in later chapters, we will revisit the quick sort algorithm, and we will see the code then.

Summary

In this chapter, we have noticed a few interesting trade-offs which occurs:

1. Finding elements in an array is much faster if the array is sorted.
2. Sorting (and maintaining a sorted list) is a difficult and time-consuming task.
3. Improved algorithms for sorting (and maintaining) lists are programmatically more complex than simpler approaches.
4. The fastest searching approach, a binary search, requires that the elements be stored as a contiguous block of RAM.

In the next chapters, we will start developing data types which are intended to deal with some of these trade-offs (not that they won't have different trade-offs to be considered).

Chapter Terminology: Be able to fully describe these terms

Ave. No. Binary Comparisons	Max. No. Sequential Comparisons
Ave. No. Sequential Comparisons	Max. Sort Comparisons
Binary Searches	Max. Sort Passes
Bubble Sorts	Partitioned Lists
Bubble-Down Sorts	Pivot Element
Bubble-Up Sorts	Quick Sorts
Comparisons	Recursion
Element Position	Selection Sorts
Exchange Sorts	Sequential Searches
External Sorts	Sort Flags
Index (Indices)	Sort Passes
Insertion Sorts	Sublists
Internal Sorts	Swapping
Max. No. Binary Comparisons	Trade-off btw. Sorted and Unsorted Lists

Review Questions

1. What are the advantages and Disadvantages of unsorted lists? When should they be used?

2. What are the advantages and disadvantages of sorted lists? When should they be used?

3. Given an unsorted list of 173 names, what is the maximum number of comparisons will it take to find an name? The average number of comparisons?

4. Given a sorted list of 173 names, what is the maximum number of comparisons will it take to find an name using a binary search? The average number of comparisons?

5. Given the sorted array: **12, 21, 29, 34, 35, 37, 42, 48, 50** show in detail the steps which will be used to find the numbers **12**, **48**, and **39** using a binary search.

6. Describe the differences between internal and external sorting methods? When should each be used?

7. Define: Exchange Sorts, Selection Sorts, and Insertion Sorts.

8. Describe how a bubble sort (one-way, two-way, with and without sorted flags) works. When is a bubble sort appropriate?

9. Given a list of 1,453 (unsorted) elements to be sorted using a bubble-sort, what is the maximum number of passes needed? The maximum number of comparisons?

10. Explain, in general terms, how a quick sort works.

11. Given a list of 41 elements, what is the maximum number of comparisons needed? How does this contrast with a bubble sort?

12. What is a recursive function?

Review Question Answers (NOTE: checking the answers before you have tried to answer the questions doesn't help you at all)