9 SEARCHING AND SORTING

OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Basic search and sort algorithms
- Algorithms with respect to time and space complexity
- Appropriate algorithms suitable for practical applications

One of the most common and time consuming tasks in computer science is the retrieval of target information from huge data, which needs searching. Searching is the process of finding the location of the target among a list of objects. The two basic search techniques are the following:

1. Sequential search
2. Binary search

There are certain ways of organizing data, which make the search process more efficient. If the data is kept in a proper order, it is much easier to search. Sorting is a process of organizing data in a certain order to help retrieve it more efficiently.

In this chapter, we shall study searching and sorting methods. We shall also analyse the algorithms in terms of time complexity.

9.1 SEARCHING

The process of locating target data is known as *searching*. Consider a situation where you are trying to get the phone number of your friend from a telephone directory. The telephone directory can be thought of as a table or a file, which is a collection of records. Each record has one or more fields such as name, address, and telephone number. The fields, which are used to distinguish records, are known as *keys*. While searching, we are asked to find the record which contains information along with the target key. When we think of a telephone directory, the search is usually by name. However, when we try to locate the record corresponding to a given telephone number, the key will be the telephone number.

If given an address and the person's name and telephone number need to be located, the person's address will be the key.

If the key is unique and if it determines a record uniquely, it is called a *primary key*. For example, telephone number is a primary key. As any field of a record may serve as the key for a particular application, keys may not always be unique. For example, if we use 'name' as the key for a telephone directory, there may be one or more persons with the same name. In addition, sorted organization of a directory makes searching easier and faster.

We may use one of the two linear data structures, arrays and linked lists, for storing the data. Search techniques may vary according to data organization. The data may be stored on a secondary storage or permanent storage area. If the search is applied on the table that resides at the secondary storage (hard disk), it is called as *external searching*, whereas searching of a table that is in primary storage (main memory) is called as *internal searching* which is faster than external searching.

A searching algorithm accepts two arguments as parameters—a target value to be searched and the list to be searched. The search algorithm searches a target value in the list until the target key is found or can conclude that it is not found.

One of the most popular applications of search algorithms is adding a record in the collection of records. While adding, the record is searched by key and if not present, it is inserted in the collection. Such a technique of searching the record and inserting it if not found is known as *search and insert algorithm.*

9.2 SEARCH TECHNIQUES

Depending on the way data is scanned for searching a particular record, the search techniques are categorized as follows:

1. Sequential search
2. Binary search
3. Fibonacci search
4. Index sequential search
5. Hashed search

The performance of a searching algorithm can be computed by counting the number of comparisons to find a given value. We shall study these algorithms with respect to arrays. For sequential search, the same concept applies for searching data in linked lists as well as files.

9.2.1 Sequential Search

The easiest search technique is a sequential search. This is a technique that must be used when records are stored without any consideration given to order, or when the storage medium lacks any type of direct access facility. For example, magnetic tape and linked

list are sequential storage media where the data may or may not be ordered. There are two ways for storing the collection of records namely, sequential and non-sequential. For the time being, let us assume that we have a sequential file *F*, and we wish to retrieve a record with a certain key value *k*. If *F* has *n* records with the key value k_i such as $i = 1$ to *n*, then one way to carry out the retrieval is by examining the key values in the order of their arrangement until the correct record is located. Such a search is known as *sequential search* since the records are examined sequentially from the first till the last.

Hence, a sequential search begins with the first available record and proceeds to the next available record repeatedly until we find the target key or conclude that it is not found. Sequential search is also called as *linear search*.

Algorithm 9.1 depicts the steps involved in sequential search.

ALGORITHM 9.1

```
1. Set i = 0, flag = 0
2. Compare key[i] and target
   if(key[i] = target)
      Set flag = 1, location = i and goto step 5
3. Move to next data element
   i = i + 1
4. if(i < n) goto step 2
5. if(flag = 1) then
      return i as position of target located
   else
      report as 'Target not found'
6. stop
```

Figure 9.1 shows a sample sequential unordered data and traces the search for the target data of 89.

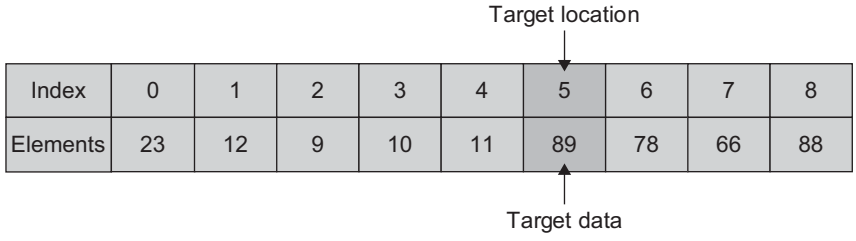

Fig. 9.1 Sequential search for target data of 89

Initially, $i = 0$ and the target element 89 is to be searched. At each pass, the target 89 is compared with the element at the i^{th} location till it is found or the index *i* exceeds the size. At $i = 5$, the search is successful.

Algorithm 9.1 for sequential search is implemented in C++ as shown in Program Code 9.1.

PROGRAM CODE 9.1

```
int SeqSearch (int A[max], int key, int n)
{
   int i,  flag = 0, position;
   for(i = 0; i < n; i++)
   {
      if(key == A[i])
      {
         position = i;
         flag = 1;
         break;
      }
   }
   if(flag == 1)      // if found return position
      return(position);
   else      // return -1 if not found
      return(-1);
}
```

The function `SeqSearch()` is defined with three parameters—the element to be searched, the array `A` where the element is to be searched, and the total number of elements in the array. The function `SeqSearch()` returns the location of the element if found or returns −1 if the element is not found.

Let us compute the amount of time the sequential search needs to search for a target data. For this, we must compute the number of times the comparisons of keys is done. In general, for any search algorithm, the computational complexity is computed by considering the number of comparisons made.

The number of comparisons depends on where the target data is stored in the search list. If the target data is placed at the first location, we get it in just one comparison. Two comparisons are needed if the target data is in the second location. Similarly, i comparisons are required if the target data is at the i^{th} location and n comparisons, if it is at the n^{th} location. As the total number of comparisons depends on the position of the target data, let us compute the average complexity of the algorithm. Average complexity is the sum of number of comparisons for each position of the target data divided by n and is given as follows:

$$\begin{aligned}\text{Average number of comparisons} &= (1 + 2 + 3 + \ldots + n)/n \\ &= (\Sigma n)/n \\ &= ((n(n+1))/2) \times 1/n \\ &= (n+1)/2\end{aligned}$$

Hence, the average number of comparisons done by the sequential search method in the case of a successful search is $(n + 1)/2$. An unsuccessful search is given by n comparisons. The number of comparisons is n and the complexity is denoted as $O(n)$.

The worst case complexity is n, which means that the target data element is at the n^{th} location and hence requires n comparisons. The best case complexity is 1, as the target data element is at the first location and requires only a single comparison. Sequential search is suitable when the data is stored in an unordered manner and also when there is no way to directly access the data elements. For example, to search the data record stored on a magnetic tape, it has to be searched sequentially from the first location till the n^{th} location. The linear list implemented using a linked list cannot access any i^{th} element directly except $(i = 1)$. We need to search through the whole list to retrieve a target data. Hence, sequential search is used if the data is unsorted and if the storage does not provide direct access to the data.

Pros and Cons of Sequential Search

The following lists detail the pros and cons of sequential searching:

Pros

1. A simple and easy method
2. Efficient for small lists
3. Suitable for unsorted data
4. Suitable for storage structures which do not support direct access to data, for example, magnetic tape, linked list, etc.
5. Best case is one comparison, worst case is n comparisons, and average case is $(n + 1)/2$ comparisons
6. Time complexity is in the order of n denoted as $O(n)$.

Cons

1. Highly inefficient for large data
2. In the case of ordered data other search techniques such as binary search are found more suitable.

Variations of Sequential Search

The time complexity of sequential search is $O(n)$; this amounts to one comparison in the best case, n comparisons in the worst case, and $(n + 1)/2$ comparisons in the average case. The algorithm starts at the first location and the search continues till the last element. We can make a few changes leading to a few variations in the sequential search algorithm. There are three such variations:

1. Sentinel search
2. Probability search
3. Ordered list search

Sentinel search We note that in steps 2–4 of Algorithm 9.1, there are two comparisons—one for the element (key) to be searched and the other for the end of the array. The algorithm ends either when the target is found or when the last element is compared. The algorithm can be modified to eliminate the end of list test by placing the target at the end of list as just one additional entry. This additional entry at the end of the list is called as a *sentinel*. Now, we need not test for the end of list condition within the loop and merely check after the loop completes whether we found the actual target or the sentinel. This modification avoids one comparison within the loop that varies *n* times. The only care to be taken is not to consider the sentinel entry as a data member.

Algorithm 9.2 depicts the steps involved in sentinel search.

ALGORITHM 9.2

```
1. Set i = 0
2. list[n] = target {add sentinel}
3. Compare key[i] and target
   if(key[i] = target)
     Set location = i and goto step 6
4. Move to next data element
   i = i + 1
5. goto step 3
6. if(location < n) then
     return location as position of target
7. else
     report as 'Target not found' and return -1
8. stop
```

Algorithm 9.2 is implemented in C++ as in Program Code 9.2.

PROGRAM CODE 9.2

```
int SeqSearch_sentinel (int A[max], int key, int n)
{
   int i, position;
   A[n] = key;      // place target at end of the list
   while(key != A[i])
   {
      i = i + 1;
   }
   //if found at sentinel then return position
   if(i < n)
      return(i);
   else      // return -1 if not found
      return(-1);
}
```

Probability search In probability search, the elements that are more probable are placed at the beginning of the array and those that are less probable are placed at the end of the array.

Ordered list search When elements are ordered, binary search (discussed in Section 9.2.2) is preferred. However, when data is ordered and is of smaller size, sequential search with a small change is preferred to binary search. In addition, when the data is ordered but stored in a data structure such as a linked list, modified sequential search is preferred. While searching an ordered list, we need not continue the search till the end of list to know that the target element is not in the list. While searching in an ascending ordered list, whenever an element that is greater than or equal to the target is encountered, the search stops. We can also add a sentinel to avoid the end of list test.

9.2.2 Binary Search

As discussed, sequential search is not suitable for larger lists. It requires *n* comparisons in the worst case. We have a better method when the data is sorted. Let us consider a typical game played by kids. You are asked to guess the number thought of by your friend in the range of 1 to 100. You are to guess by asking him a minimum number of questions. Of course, you are not allowed to ask him the number itself. The easiest approach is to start asking him, 'Is it 1?' In case the answer is 'No', then ask, 'Is it 2?' Continue this process in the ascending order of integers till you get the answer as 'Yes'.

What if the number your friend has in mind is 99? Obviously, this approach is not an efficient one. The solution to this problem is to ask him a question, 'Is it 50?' If no, another question to be asked is, 'is it greater than 50?' If the answer is 'Yes', then the range to be searched is 51 to 100, which is half of the previous range. If the answer is 'No', the range is 1 to 49, which is again half of the original. You may continue doing so till you guess the number. Surely, the second approach reduces the total number of questions asked on an average.

This method is called *binary search*, as we have divided the list to be searched every time into two lists and the search is done in only one of the lists. Consider that the list is sorted in ascending order. In binary search algorithm, to search for a particular element, it is first compared with the element at the middle position, and if it is found, the search is successful, else if the middle position value is greater than the target, the search will continue in the first half of the list; otherwise, the target will be searched in the second half of the list. The same process is repeated for one of the halves of the list till the list is reduced to size one.

Algorithm 9.3 depicts the logic behind this type of search.

ALGORITHM 9.3

```
1. Let n be size of the list
   Let target be the element to be searched
   Let flag = 0, low = 0, high = n–1
2. if low ≤ high, then
      middle = (low + high)/2
```

```
   else goto step (5)
3. if(key[middle] = target)
      Position = middle, flag = 1
   Goto step (5)
   else if(key[middle] > target) then
      high = middle – 1
   else
      low = middle + 1
4. Goto step(2)
5. if flag = 1
      report as target element found at location 'position'
   else
      report that element is not found in the list
6. stop
```

The effectiveness of the binary search algorithm lies in its continual halving of the list to be searched. For an ordered list of 50,000 keys, the worst case efficiency is a mere 16 accesses. One may note that the dramatic increase in efficiency is noticed as the list gets larger. We can check with a calculator as to how many times 50,000 must be halved to be reduced to 1. The same list that would have necessitated an average wait of two minutes using a sequential search will give a virtually instantaneous response when the binary search is used. In more precise algebraic terms, the halving method yields a worst case search efficiency of $\log_2 n$.

A non-recursive code in C++ that demonstrates the implementation of Algorithm 9.3 is given in Program Code 9.3 and a recursive code for the same is given in Program Code 9.4.

PROGRAM CODE 9.3

```
int Binary_Search_non_recursive(int A[], int n, int key)
{
   int low = 0,high = n-1,mid;
   while(low <= high)
   {  //iterate while first <= last
      mid = (low + high)/2;      //calculate
      mid = (first + last)/2)
      if(A[mid] == key)      //found
         return mid;      // return position (mid)
      else if(key<A[mid])
         //not found; look in upper half of list
         high = mid – 1;
      else
         low = mid + 1;      //look in lower half
   }
   return –1;      //return "not found"
}
```

Although this is a more direct implementation it uses needless stack space, and is much slower on most systems. In addition, this form of recursion is called *tail recursion*, which is the most wasteful form of recursion. Recursion is a powerful tool, which must be used with care. A recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call. Tail recursion is also used to return the value of the last recursive call as the value of the function. It is advantageous as the amount of information which must be stored during computation is independent of the number of recursive calls.

Program Code 9.4 is the recursive code in C++ that demonstrates the implementation of Algorithm 9.3 of binary search.

PROGRAM CODE 9.4

```
// Function binary search (recursive)
int Binary_Search(int A[],int low,int high,int key)
{
   int mid;
   if(low <= high)
   {
      mid = (low + high)/2;
      if(A[mid] == key)
         return mid;
      else if(key < A[mid])
         return Binary_Search(A, low, mid - 1, key);
      else
         return Binary_Search(A,mid + 1,high, key);
   }
   return -1;
}
```

Time Complexity Analysis

Time complexity of binary search is O(log(n)) as it halves the list size in each step. It is a large improvement over linear search; for a list with 10 million entries, linear search will need 10 million key comparisons in the worst case, whereas binary search will need just about 24 comparisons.

The time complexity can be written as a recurrence relation as

$$T(n) = \begin{cases} T(1), & n = 1 \\ T(n/2) + c, & n > 1 \end{cases}$$

The most popular and easiest way to solve a recurrence relation is to repeatedly make substitutions for each occurrence of the function T on the right-hand side until all such occurrences disappear.

$$
\begin{aligned}
\text{Therefore, } T(n) = T(n/2) + c &= T(n/2^2) + 2c \quad (\text{after } 2^{\text{nd}} \text{ substitution}) \\
&= T(n/3^3) + 3c \quad (\text{after } 3^{\text{rd}} \text{ substitution}) \\
&\vdots \\
&= T(n/2^i) + ic \quad (\text{after } i^{\text{th}} \text{ substitution}) \\
&\vdots \\
&= T(2^k/2^k) + kc \quad (\text{after } k \text{ steps}) \\
&= T(1) \\
\text{where } 2^k &= n,\ k = \log_2 n \\
T(n) &= O(\log_2 n)
\end{aligned}
$$

Although binary search is good, it can again be slightly improved using Fibonacci search.

Pros and Cons of Binary Search

The following are the pros and cons of a binary search:

Pros

1. Suitable for sorted data
2. Efficient for large lists
3. Suitable for storage structures that support direct access to data
4. Time complexity is $O(\log_2(n))$

Cons

1. Not applicable for unsorted data
2. Not suitable for storage structures that do not support direct access to data, for example, magnetic tape and linked list
3. Inefficient for small lists

9.2.3 Fibonacci Search

We all know about Fibonacci numbers. It has many diverse applications from estimation of the number of cells in successive reproductions to the number of leaves on branches. The Fibonacci series has 0 and 1 as the first two terms, and each successive term is the sum of the previous two terms. Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... with $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$ where, $F_0 = 0$ and $F_1 = 1$.

Fibonacci search modifies the binary search algorithm slightly. Instead of halving the index for a search, a Fibonacci number is subtracted from it. The Fibonacci number to be subtracted decreases as the size of the list decreases.

Fibonacci search starts searching for the target by comparing it with the element at the F_k^{th} location. Here, $F_k \geq n$ and $F_{k-1} < n$. The Fibonacci search works like the binary search but with a few modifications. In binary search, we have low, high, and mid positions for

the sub-list. Here, we have mid = $n - F_{k-1} + 1$, $F_1 = F_{k-2}$, and $F_2 = F_{k-3}$. The target to be searched is compared with `A[mid]`. `mid` is computed as follows:

Case 1 if equal the search terminates;

Case 2 if the target is greater and F_1 is 1, then the search terminates with an unsuccessful search; else the search continues at the right of the list with new values of low, high, and mid as

$$\text{mid} = \text{mid} + F_2,\ F_1 = F_{k-4}, \text{ and } F_2 = F_{k-5}$$

Case 3 if the target is smaller and F_2 is 0, then the search terminates with an unsuccessful search; else the search continues at the left of the list with new values of low, high, and mid as

$$\text{mid} = \text{mid} - F_2,\ F_1 = F_{k-3} \text{ and } F_2 = F_{k-4}$$

The search continues by either searching at the left of mid or at the right of mid in the list. Algorithm 9.4 explains the working of this search technique.

ALGORITHM 9.4

1. Set k = m
2. if k = 0, finish and display message "not found" and goto 6
3. if item = A[F_{k-1}], print "found" and goto 6
4. if(item < A[F_{k-1}]), discard entries from positions F_{k-1} + 1 to n, set k = k – 1, and goto 2
5. if item > A[F_{k-1}], discard entries from positions 1 to F_{k-1}, renumber remaining entries from 1 to F_{k-2}, set k = k – 2, and goto 2
6. stop

Program Code 9.5 implements Algorithm 9.4.

PROGRAM CODE 9.5

```
// Function to find nth Fibonacci number
int fibo(int n)
{
   if(n == 0 || n == 1)
      return 1;
   else
      return(fibo(n - 1) + fibo(n - 2));
}

// Function for Fibonacci search
int Fibonacci_Search(int A[],int n, int key)
{
   int f1, f2, t, mid, j, f;
   j = 1;
   while(fibo(j) <= n)
```

```
    {       //find fibo(j) such that fibo(j) >= n
       j++;
    }
    f = fibo(j);
    f1 = fibo(j - 2);       //find lower Fibonacci numbers
    f2 = fibo(j - 3);
    mid = n - f1 + 1;
    while(key != A[mid])       // if not found
    {
       if(mid < 0||key > A[mid])
       {       //look in lower half
          if(f1 == 1)
             return -1;
          mid = mid + f2;       //decrease Fibonacci numbers
          f1 = f1 - f2;
          f2 = f2 - f1;
       }
       else
       {      //look in upper half
          if(f2 == 0)       //if not found return -1
             return -1;
          mid = mid - f2;       //decrease Fibonacci numbers
          t = f1 - f2;       //this time, decrease more
          f1 = f2;       //for smaller list
          f2 = t;
       }
    }
    return mid;
}
```

Example 9.1 illustrates a Fibonacci search in a given list.

EXAMPLE 9.1 Search for 81 using Fibonacci search in the list {6, 14, 23, 36, 55, 67, 76, 78, 81, 89}, where $n = 10$.

Solution

Step 1: Compute F_k such that $F_k \geq 10$

`fibo(7)` = 13, which is greater than 10. Hence, $k = 7$.

Step 2: Compute the initial values of mid, F_1 and F_2.

Now, $F_1 =$ `fibo(7 - 2)` $=$ `fibo(5)` $= 5$

$F_2 =$ `fibo(7 - 3)` $=$ `fibo(4)` $= 3$

`mid` $= 10 - F_1 + 1 = 10 - 5 + 1 = 6$

Step 3: Let us search the target by comparing it at `A[mid]`. Now,

(a) compare `A[mid]`, that is 76, and the number to be searched, that is 78, which are not equal.

(b) 78 > `A[mid]`, and F_1 is not 1; hence, let us compute `mid`, F_1, and F_2.

(c) `mid` = `mid` + F_2 = 6 + 3 = 9, and F_1 = 2, and F_2 = 1.

Step 4: Again, 78 is not equal to `A[9]` and is lesser. Hence, let us search the lower half as `mid` = `mid` − F_2 = 9 − 1 = 8, F_1 = 1, and F_2 = 1

Step 5: Now, compare 78 and `A[mid]`, which are equal; hence, the search stops.

The search terminates with a successful search by locating the target at the eighth location in the second iteration.

Time Complexity of Fibonacci Search

When we solve a recurrence relation $F_n = F_{n-1} + F_{n-2}$ for Fibonacci numbers, we get the solution as F_n = (1/sqrt(5)) × [((1 + sqrt(5))/2)^n + ((1 + sqrt(5))/2)^n]. For large n, the term ((1 − sqrt(5))/2)^n tends to zero. Hence F_n is bounded by ((1 − sqrt(5))/2)^n. Hence, $F_n \leq n \times \log[(1 + \text{sqrt}(5))/2]$. The number of comparisons is of the order of n, and the time complexity is O(log(n)).

Hence, the algorithm for Fibonacci search is O(log(n)) algorithm. Consider an example where for a list of 10 numbers, each element of the 10 numbers is to be searched once. For an unsuccessful search, the algorithm needs a total of 13 searches. In case of binary search, the number of comparisons would be 40, and for Fibonacci search, it will be 41. Since this is a small-scale example, binary search will score, but in larger instances, it may be the other way around.

Fibonacci search is more efficient than binary search for large lists. However, it is inefficient in case of small lists.

Pros

1. Faster than binary search for larger lists
2. Suitable for sorted lists

Con

1. Inefficient for smaller lists

9.2.4 Indexed Sequential Search

Indexed sequential search is suitable for sequential files. A sequential file with an associated index is just like an index associated with books. File index is a data structure similar to a list of keys and their location or reference to the location of the record associated with the key.

We discussed the drawbacks associated with searching a record sequentially in a file or a table. An index file can be used to effectively overcome the problem associated with sequential files and to speed up the key search. The simplest indexing structure is the single-level one: a file whose records are pairs (key and a pointer), where the pointer is the position in the data file of the record with the given key. Only a subset of data records, evenly spaced along the data file, is indexed to mark the intervals of data records.