Data Structure and Algorithms

Chapter 3

Simple Searching and Sorting Algorithms

Simple Sorting and Searching Algorithms

- **Searching**
  - Searching is a process of looking for a specific element in a list of items or determining that the item is not in the list.
- There are two simple searching algorithms:
  - Sequential Search (linear Search)
  - Binary Search

Linear Search (Sequential Search)

Pseudocode

- Loop through the array starting at the first element until the value of target matches one of the array elements.
- If a match is not found, return -1.

- It has time complexity ***O(n)***.

Implementation:

```
int Linear_Search(int list[], int key)
{
        int index=0;
        int found=0;
        do{
                if(key==list[index])
                        found=1;
                else
                        index++;
        }while(found==0 && index<n);
        if(found==0)
                index=-1;
        return index;
}
```

Binary Search

- This searching algorithms works only on an **ordered list**.
- The basic idea is:
  - Locate midpoint of array to search
  - Determine if target is in lower half or upper half of an array.
  - If in lower half, make this half the array to search
  - If in the upper half, make this half the array to search
  - Loop back to step 1 until the size of the array to search is one and this element does not match, in which case return -1
- The computational time for this algorithm is proportional to $\log_2 n$. Therefore the time complexity is ***O(log n)***

Implementation:

```
int binarySearch( int list[ ], int key )
{
      int low = 0 ;          // low end of the search area
      int high = n - 1 ;     // high end of the search area
      int middle = ( low + high + 1 ) / 2 ;          // middle element
      int location = -1;  // return value; -1 if not found
     do              // loop to search for element
       {
         // if the element is found at the middle
         if ( key == list[ middle ] )
                location = middle;      // location is the current
```

```
 // middle element is too high
 else if ( key < list[ middle ] )
      high = middle - 1 ;      // eliminate the higher half
 else // middle element is too low
      low = middle + 1 ;    // eliminate the lower half
 middle = ( low + high + 1 ) / 2 ; // recalculate the middle
 } while ( ( low <= high ) && ( location == -1 ) );
 return location;// return location of search key
} // end
```

Sorting Algorithms

- **Sorting** is one of the most important operations performed by computers. Sorting is a process of reordering a list of items in either increasing or decreasing order.

The following are simple sorting algorithms

- Insertion Sort
- Selection Sort
- Bubble Sort

Insertion Sort

- The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list.

Basic Idea:

- Find the location for an element and move all others up, and insert the element.
- The process involved in insertion sort is as follows:

1. The left most value can be said to be sorted relative to itself. Thus, we don't need to do anything.
2. Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.
3. Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.
4. Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.

5. Now the first three are relatively sorted.
6. Do the same for the remaining items in the list

Implementation

```
void insertion_sort(int list[ ]){
int temp;
   for(int i=1;i<n;i++){
         temp=list[i];
         for(int j=i; j>0 && temp<list[j-1];j--)
         {
               list[j]=list[j-1];
               list[j-1]=temp;
         }//end of inner loop
   }//end of outer loop
}//end of insertion_sort
```

- Sort the following data using **Insertion Sort Algorithm**
- Assume name of the array is **list**

Analysis

How many comparisons?

$1+2+3+\ldots+(n-1)= O(n^2)$

How many swaps?

$1+2+3+\ldots+(n-1)= O(n^2)$

Selection Sort

- **Basic Idea:**
  - Loop through the array from i=0 to n-2.
  - Select the smallest element in the array from i+1 to n-1
  - Swap this value with value at position i.

Implementation:

```
void selection_sort(int list[ ])
{
int i,j, smallest;
for(i=0;i<=n-2;i++){
        smallest=i;
```

```
        for(j=i+1;j<=n-1;j++){
                if(list[j]<list[smallest])
                        smallest=j;
        }//end of inner loop
                temp=list[smallest];
                list[smallest]=list[i];
                list[i]=temp;
} //end of outer loop
}//end of selection_sort
```

- Sort the following data using **Selection Sort Algorithm**
- Assume name of the array is **list**

Analysis

How many comparisons?

$(n-1)+(n-2)+\ldots+1 = O(n^2)$

How many swaps? $1+1+\ldots+1 = (n-1)$ times

$n = O(n)$

Bubble Sort

- Bubble sort is the simplest algorithm to implement and the slowest algorithm on very large inputs.
- **Basic Idea:**
  - Loop through array from i=0 to n-1 and swap adjacent elements if they are out of order.
- **Implementation:**

```
void bubble_sort(list[])
{
        int i,j,temp;
        for(i=0;i<n-1; i++){
        for(j=n-1;j>i; j--){
```

```
                if(list[j]<list[j-1]){
                temp=list[j];
                        list[j]=list[j-1];
                        list[j-1]=temp;
                }//swap adjacent elements
        }//end of inner loop
        }//end of outer loop
}//end of bubble_sort
```

- Sort the following data using **Bubble Sort Algorithm**
- Assume name of the array is **list**

- **Analysis of Bubble Sort**

How many comparisons?

$(n-1)+(n-2)+\ldots+1 = O(n_2)$

How many swaps?

$(n-1)+(n-2)+\ldots+1 = O(n_2)$