2.1. 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, and
- Binary Search

2.1.1. 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.

Time is proportional to the size of input (*n)* and

we call this time complexity *O(n)*

.

Example 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;
}
```

2.1.2. Binary Search

This searching algorithms works only on an ordered list.

The basic idea is:

- Locate midpoint of array to search
- Check if the value in the midpoint is the target value
  if value at midpoint matched the desired target value
  show flag indicating the presence of the item.
  else
  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 the target element, 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)*

Example Implementation:

```
int Binary_Search(int list[],int k)
{
int left=0;
int right=n-1;
int found=0;
do{
mid=(left+right)/2;
if(key==list[mid])
   found=1;
else{
    if(key<list[mid])
        right=mid-1;
    else
        left=mid+1;
    }
}while(found==0&&left<right);
if(found==0)
  index=-1;
else
  index=mid;
return index;
```

}

2.2. 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 used to sort small-sized lists.

- Insertion Sort
- Selection Sort
- Bubble Sort

2.2.1. Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

It's the most instinctive type of sorting algorithm. The approach is the same approach that you use for sorting a set of cards in your hand. While playing cards, you pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

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--)
           { // work backwards through the array finding where temp should go
              list[j]=list[j-1];
            list[j-1]=temp;
            }//end of inner loop
          }//end of outer loop
}//end of insertion_sort
```

Analysis

How many comparisons?

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

How many swaps?

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

How much space?

In-place algorithm

2.2.2. Selection Sort

Basic Idea:

- Loop through the array from i=0 to n-1.
- Select the smallest element in the array from i to n
- Swap this value with value at position i.

Implementation:

```
void selection_sort(int list[])
{
int i,j, smallest;
for(i=0;i<n;i++){
     smallest=i;
   for(j=i+1;j<n;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
```

Analysis

How many comparisons?

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

How many swaps?

$n=O(n)$

How much space?

In-place algorithm

2.2.3. 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 and swap adjacent elements if they are out of order.

Implementation:

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

Analysis of Bubble Sort

How many comparisons?

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

How many swaps?

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

Space?

In-place algorithm.