Chapter Two

Simple Sorting and Searching Algorithms

- Storing and retrieving information is one of the most common application of computers now a day.
- According to time the amount of data and information stored and accesses via computer has turned to huge database
- So many techniques and algorithm have been developed to efficiently maintain and process information in database
- The process of looking the particular data records in database called **searching**
- The process of ordering the record in the database called **sorting**
- Sorting and searching together constitutes a major area of studying computational methods
- Both of them are the most important field of study in DSA

Why do we study sorting and searching algorithms?

- **These algorithms are the most common and useful tasks operated by computer system.**
- **Computers spend a lot of time searching and sorting.**

1. Simple Searching algorithms

Searching:- is a process of finding an element in a list of items or determining that the item is not in the list.

- **To keep things simple, we shall deal with a list of numbers.**
- **A search method looks for a *key*, arrives by parameter.**
- **By convention, the method will return the index of the element corresponding to the key or, if unsuccessful, the value -1.**
- **There are two simple searching algorithms:**
  a) Sequential Search, and
  b) Binary Search

a). Sequential Searching (Linear)

- **The most natural way of searching an item.**
- **Easy to understand and implement.**

Algorithm:

- In a linear search, we start with top (beginning) of the list, and compare the element at top with the key.
- If we have a match, the search terminates and the index number is returned.
- If not, we go on the next element in the list.
- If we reach the end of the list without finding a match, we return -1.

Implementation:

- **Assume the size of the list is n.**

```
int LinearSearch(int list[ ], int key)
{
   index=-1;
  for(int i=0; i<n; i++)
   {
     if(list[i]==key)
      {
         index=i;
         break;
       }
    }
return index;
 }
```


Example

- Search elements from the given array using linear search

10	6	7	8	3
0	1	2	3	4

- The searching element key element is 7 and the value of n is 5
- Based on the above algorithm
- int index=-1;
- (for i=0;i<n;i++) →here it the condition is true it directly proceed to the if condition before the value of i is incremented
- if(list[0]=key)→10=7? False it will go to the loop and the value of i incremented by 1 before the condition is checked then
- if(list[1]=key)→6=7? False again the value of I is incremented by one before the condition
- If(list[2]=key)→7=7? True then index=i→ index =2 and then break which means the loop is terminated and goes to return index=2

Complexity Analysis:

- Big-Oh of sequential searching ➔ How many comparisons are made in the worst case ? n
- O(n).

b). <u>Binary Searching</u>

- Assume sorted data.
- Use Divide and conquer strategy (approach).

Algorithm:

- In a binary search, we look for the key in the *middle* of the list. If we get a match, the search is over.
- If the key is greater than the element in the middle of the list, we make the top (upper) half the list to search.
- If the key is smaller, we make the bottom (lower) half the list to search.
- Repeat the above steps (1st, 2nd, and 3rd) until one element remains.
- If this element matches return the index of the element, else return -1 index. (-1 shows that the key is not in the list).

Implementation:

```
int BinarySearch(int list[ ], int key)
 {
   int found=0,index=0;
   int top=n-1,bottom=0,middle;
do{
     middle=(top + bottom)/2;
   if(key==list[middle])
     found=1;
   else{
     if(key<list[middle])
     top=middle-1;
     else
        bottom=middle+1;
      }
  }
}while(found==0 && top>=bottom);
if(found==0)
   index=-1;
 else
   index=middle;
return index;
}
```

Complexity Analysis:

Example: Find Big-Oh of Binary search algorithm in the worst case analysis.

Example

0	1	2	3	4
5	13	19	27	33

- From the given array initially
- Found=0,index=0,top=n-1 which is 4,bottom=0
- Let us search 19
- When we see the array it is already sorted then
- In side do middle=(top+bottom)/2 →4+0/2=2 then middle=2
- If(key=list[middle])→key=list[2]= key=list[2] →key=19 true (it is best case) then found=1 and the else part is scaped directly goes to while loop
- While(found==0&&top>=bottom) here found is 1 not 0 the condition is false if one condition is false the next also become false b/c connected by &&
- If the while loop condition is false it will proceed to the if condition
- If(found==0) →false b/c we get found 1 then it will goes to else condition
- Index=middle→index=2 then return the index i.e
- Return index which is 2 so what value is found at index 2 which is 19
- Let us change the key value as 13
- Found=0,index=0,top=n-1,bottom=0,middle;
- Inside do middle=(top+bottom)/2 →4+0/2=2, middle value
- If(key=list[middle])→13=list[2] which is 13 !=19 which is false
- The else stmt excuted →if(key<list[middle])→13<19 then
- Top=middle-1 →2-1=1 , top=1, then we proceed to while loop
- While(found==0 && top>=bottom) →found =0 T & 1>=0 T

- When the condition inside the while loop is true it directly proceed to the loop do then
- Middle =(top+bottom)/2→1+0/2=0 it will take only integer
- If(key=list[middle])→13=list[0], 13=5 false then the else stmt is executed
- If(key<list[middle])→13<list[0]→13<5 false then the else stmt executed
- bottom=bottom+1→0+1=1 bottom is 1 then proceed to while loop and check(found==0 && top>=bottom)→true && 1>=1, true, then the condition is true
- It goes to do loop middle=(top+bottom)/2→1+1/2=1 then mid=1
- If(key=list[middle])→13=list[1] true, 13=13 and found=1 the it goes to while(found=0 && top>=bottom) it is false, then else condition is executed which is found in side while
- Index=middle, which is 1 then it returns 1.

2. Sorting Algorithms

2. Simple Sorting Algorithms

- **Sorting:** is a process of reordering a list of items in increasing or decreasing order.
- Ordering a list of items is fundamental problem of science.
- Sorting is the most important operation performed computers.
- Sorting is the first step in more complex algorithms.

Two basic properties of sorting algorithms:

In-place:

- It is possible to sort very large lists without the need to additional working storage.

Cont….

Stable:

If two elements that are equal remain in the same relative position after sorting is completed.

Two classes of sorting algorithms:

$O(n^2)$:

- Includes the bubble, insertion, and selection sorting algorithms.

O(nlog n):

- Includes the heap, merge, and quick sorting algorithms.

Cont….

- Simple sorting algorithms include:
  i. Simple sorting
  ii. Bubble Sorting
  iii. Selection Sorting
  iv. Insertion Sorting

Simple sorting Algorithm:

- In simple sort algorithm the first element is compared with the second, third and all subsequent elements.
- If any one of the other elements is less than the current first element then the first element is swapped with that element.
- Eventually, after the last element of the list is considered and swapped, then the first element has the smallest element in the list.
- The above steps are repeated with the second, third and all subsequent elements.

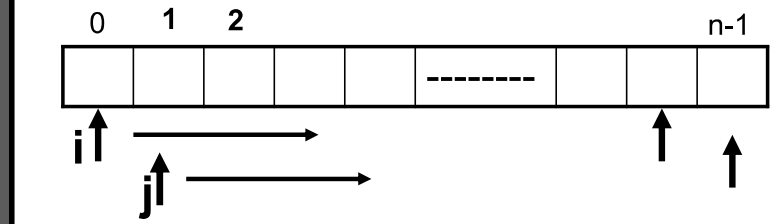
Implementation:

```
Void SimpleSort(int list[])
{
  for(int i=0; i<=n-2;i++)
      for(int j=i+1; j<=n-1; j++)
          if(list[i] > list[j])
           {
            int temp;
            temp=list[i];
            list[i]=list[j];
            list[j]=temp;
            }
}
```

Complexity Analysis:

- Analysis involves number of comparisons and swaps.
- How many comparisons?
  $1+2+3+\ldots+(n-1)= O(n^2)$
- How many swaps?
  $1+2+3+\ldots+(n-1)= O(n^2)$

Example:



		4	2	3	1
i=0	j=1	2	4	3	1
	j=2	2	4	3	1
	j=3	1	4	3	2
i=1	j=2	1	3	4	2
	j=3	1	2	4	3
i=2	j=3	1	2	3	4

Analysis: O(?)

1st pass-----➔ (n-1) comparisons

2nd pass----➔ (n-2) comparisons

|

|

|

(n-1)th pass---➔ 1 comparison

T(n)=1+2+3+4+--------+(n-2)+(n-1)

= (n*(n-1))/2

=n^2/2-n/2

=O(n^2)

Example

0	1	2	3	4
5	4	2	3	1

- Based on the above algorithms
- The value of i initially 0 and n is 5 and check the condition 0<3 T
- Before the value of i incremented proceed to next loop
- Here j=i+1→0+1=1 and check the condition j<=n-1→1<=5-1 T
- Then before the value of j incremented perform the if condition
- if(list[i]>list[j])→list[0]>list[1] i.e 5>4 T, then
- temp=list[i]→ temp holds list[0] which is 5
- List[i]=list[j]→ the value of list[j] will assign for list[i]→list[i]=4
- List[j]=temp→the value hold by temp will be assigned for list[j]=5
- Then goto the inner loop and increment the value of j by one

- The value of j is 2 then 2<=4 True,
- If(list[i]>list[j])→list[0]>list[2]→4>2 True, then swap them
- Temp holds the value of list[4] that is 4 →temp=list[i]
- List[i] holds the value of list[j] i.e list[2] which is 2 →list[i]=list[j]
- List[j] holds the value which the temp holds which is 4 →list[j]=temp

0	1	2	3	4
4	5	2	3	1

0	1	2	3	4
2	4	5	3	1

Example: Suppose we have 32 unsorted data.

a). How many comparisons are made by sequential search in the worst-case?

➔ Number of comparisons =**32.**

b). How many comparisons are made by binary search in the worst-case? (Assuming simple sorting).

➔ Number of comparisons = Number of comparisons for sorting + Number of comparisons for binary search

= (n*(n-1))/2 + logn

= 32/2(32-1)+ log 32

=**16*31 + 5**

c). How many comparisons are made by binary search in the worst-case if data is found to be already sorted?

➔ Number of comparisons = **$\log_2$ 32 = 5.**

Bubble sort

- Bubble sort is the **simplest algorithm** to implement and the **slowest algorithm on very large inputs.**

Algorithm:

- Compare each element (except the last one) with its neighbor to the right.
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last two) with its neighbor to the right.
  - If they are out of order, swap them
  - This puts the second largest element before last
  - The last two elements are now in their correct and final places

- Compare each element (except the last three) with its neighbor to the right.
- Continue as above until you have no unsorted elements on the left.
- Is the oldest, simplest, and slowest sort in use.
- It works by comparing each item in a list with an item next to it, and swap them if required.
- This causes the larger values to "bubble" to the end of the list while smaller values to "sink" towards the beginning of the list.
- In general case, bubble sort has $O(n^2)$ level of complexity.

Advantage:

- Simplicity and ease of implementation.

Disadvantage:

- Horribly inefficient.

Implementation:

```
Void BubbleSort(int list[ ])
 {    int temp;
   for (int i=n-2; i>=0; i--) {
     for(int j=0;j<=i; j++){
       if (list[j] > list[j+1]) {
         temp=list[j];
         list[j])=list[j+1];
         list[j+1]=temp;
       }
     }
 }}
```

or

```
Void BubbleSort(int list[ ])
 {    int temp;
   for (int i=0; i<n; i++) {
     for(int j=n-1;j>i; j--){
       if (list[j] < list[j-1]) {
         temp=list[j];
         list[j])=list[j-1];
         list[j-1]=temp;
         }
     }
 }}
```


Example of Bubble sort

		4 2 3 1
i=3	j=1	2 4 3 1
	j=2	2 3 4 1
	j=3	2 3 1 4
i=2	j=1	2 3 1 4
	j=2	2 1 3 4
i=1	j=1	1 2 3 4

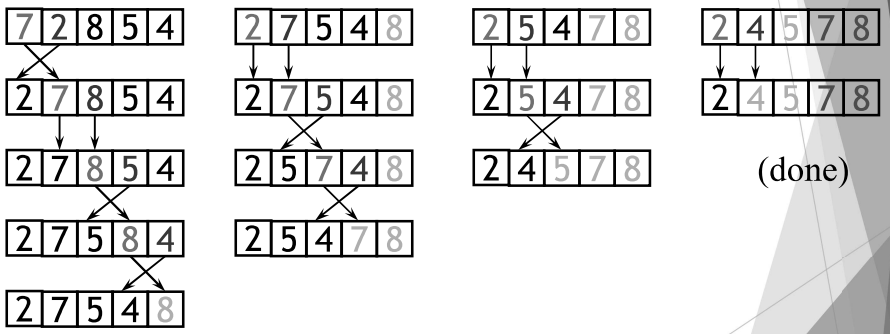
Complexity Analysis:

- Analysis involves number of comparisons and swaps.
- How many comparisons?

  $1+2+3+\ldots+(n-1)= O(n^2)$
- How many swaps?

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

Example of bubble sort



Example

0	1	2	3	4
9	2	4	7	6

- Here the value for n is 5 then we can get the value for i from the above algorithm
- For(int i=n-2;i>=0;i--)→i=5-2=3 then
- For(j=0;j<=i;j++)→0<=3 true then it enter into if condition
- If(list[0]>list[0+1]) →which means 9>2 true then swap
- temp=list[j]→ 9 will be hold by temp
- List[j]=list[j+1]→ 2 will goes to list[0] since 2 found at list(1)
- List[j+1]=temp → the value hold by the tem will be given for list[i] which is 9 is assigned for list[1]. Then goto the loop until finished

Selection Sort Algorithm

- The selection sort algorithm is in many ways similar to simple sort algorithms.
- The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part is empty, while unsorted one contains whole array.
- At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one.
- When unsorted part becomes empty, algorithm *stops*.

- **Works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled.**
- **Similar to the more efficient insertion sort.**
- **It yields a 60% performance improvement over the bubble sort.**

Advantage:

- **Simple and easy to implement.**

Disadvantage:

- **Inefficient for larger lists.**

Implementation:

```
void selectionSort(int list[ ] ) {
      int minIndex, temp;
      for (int i = 0; i <= n - 2; i++) {
            minIndex = i;
           for (j = i + 1; j <= n-1; j++)
                 if (list[j] < list[minIndex])
                       minIndex = j;
            if (minIndex != i) {
                 temp = list[i];
                 list[i] = list[minIndex];
                 list[minIndex] = temp;
             }
       }
}
```


Example:

i=0	J=1	7	9	11	3
	J=2	7	9	11	3
	J=3	7	9	11	3
i=1	J=2	3	9	11	7
	J=3	3	9	11	7
i=2	J=3	3	7	11	9
	Finally	3	7	9	11

Example

0	1	2	3	4
8	7	9	11	3

- ✓ After the loop the value of minindex=i→0 then
- ✓ The value of j is i+1 which is 1
- ✓ List[j]<list[minindex] → 7<8 true then
- ✓ The value of minindex=j which means the value of minindex=1
- ✓ The value of j will incremented by 1 and the condition 2<=4 then
- ✓ List[j]<list[minindex] →list[2]<list[1] →9<7 False again to loop
- ✓ The value of j is incremented 1
- ✓ List[3]<list[1]→11<7 False again to loop the value of j inc by 1
- ✓ List[4]<list[1]→3<7 true then the value of minindex=4 then
- ✓ If(minindex!=i)→if(4!=0) true
- ✓ temp=list[i] which is temp holds list[0] →8 then

- List[i]=list[minindex]→list[i]=list[4] which is 3 then 3 will be placed at list[0].
- List[minindex]=temp→list[4]=8 the place the value 8 at list[4]. So the area of list[0] is sorted.
- Then we will goto the first loop and the value of i is incremented by on and go until all items will be sorted

Complexity Analysis

- Selection sort stops, when unsorted part becomes empty.
- As we know, on every step number of unsorted elements decreased by one.
- Therefore, selection sort makes n-1 steps (*n* is number of elements in array) of outer loop, before stop.
- Every step of outer loop requires finding minimum in unsorted part. Summing up, (n - 1) + (n - 2) + ... + 1, results in $O(n^2)$ number of comparisons.
- Number of swaps may vary from zero (in case of sorted array) to n-1 (in case array was sorted in reversed order), which results in O(n) number of swaps.
- Overall algorithm complexity is $O(n^2)$.
- Fact, that selection sort requires n-1 number of

  swaps at most, makes it very efficient in situations, when write operation is significantly more expensive, than read operation.

- Number of swaps may vary from zero (in case of sorted array) to n-1 (in case array was sorted in reversed order), which results in O(n) number of swaps.
- Overall algorithm complexity is $O(n^2)$.
- Fact, that selection sort requires n-1 number of

  swaps at most, makes it very efficient in situations, when write operation is significantly more expensive, than read operation.

Insertion Sort

Algorithm:

- Insertion sort algorithm somewhat resembles Selection Sort and Bubble sort.
- Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part contains first element of the array and unsorted one contains the rest.
- At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one.
- When unsorted part becomes empty, algorithm *stops*.

Using binary search

It is reasonable to use binary search algorithm to find a proper place for insertion.

- This variant of the insertion sort is called binary insertion sort.
- After position for insertion is found, algorithm shifts the part of the array and inserts the element.

- Insertion sort works by inserting item into its proper place in the list.
- Insertion sort is simply like playing cards: To sort the cards in your hand, you extract a card, shift the remaining cards and then insert the extracted card in the correct place.
- This process is repeated until all the cards are in the correct sequence.
- Is over twice as fast as the bubble sort and is just as easy to implement as the selection sort.

Advantage:

- Relatively simple and easy to implement.

Disadvantage:

- Inefficient for large lists.

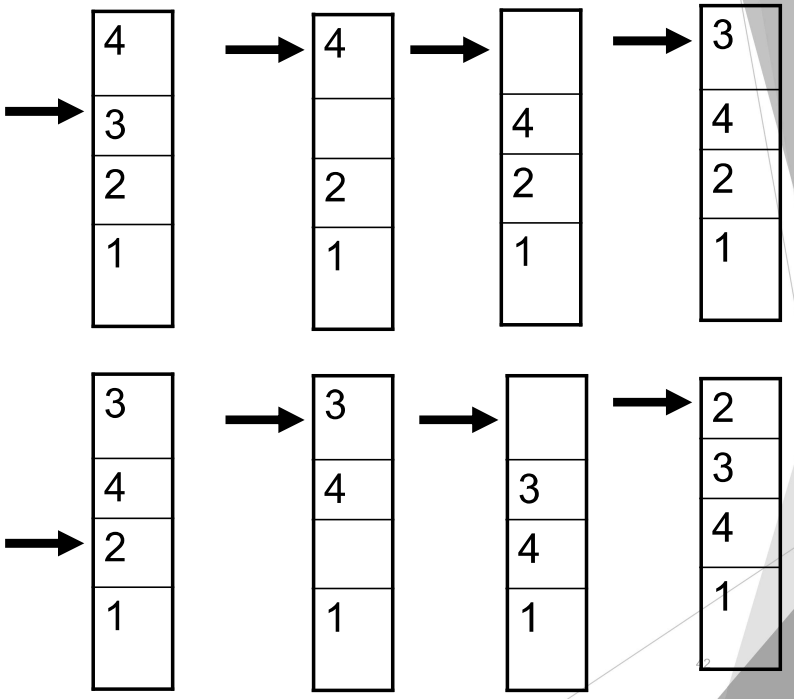
C++ implementation

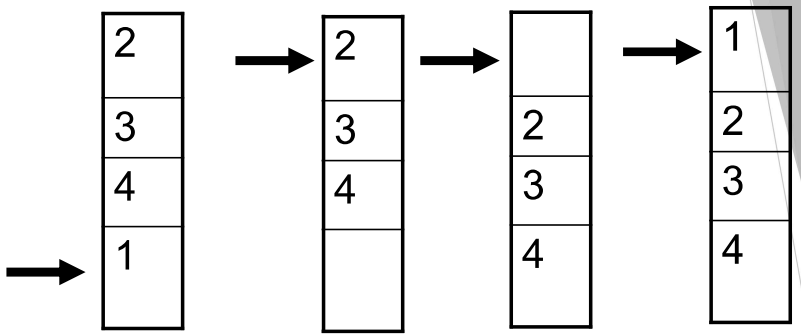
```
void InsertionSort(int list[])
 {
      for (int i = 1; i <= n-1; i++) {
        for(int j = i;j>=1; j--) {
          if(list[j-1] > list[j])
           {
             int temp = list[j];
             list[j] = list[j-1];
             list[j-1] = temp;
           }
          else
             break;
        }
    }
 }
```

Complexity Analysis

- The complexity of insertion sorting is O(n) at best case of an already sorted array and $O(n^2)$ at worst case, regardless of the method of insertion.
- Number of comparisons may vary depending on the insertion algorithm.
  - $O(n^2)$ for shifting or swapping methods.
  - O(nlogn) for binary insertion sort.

41





43

7	-5	2	16	4	unsorted
7	-5	2	16	4	-5 to be inserted
?	7	2	16	4	7 > -5, shift
-5	7	2	16	4	reached left boundary, insert -5
-5	7	2	16	4	2 to be inserted
-5	?	7	16	4	7 > 2, shift
-5	2	7	16	4	-5 < 2, insert 2
-5	2	7	16	4	16 to be inserted
-5	2	7	16	4	7 < 16, insert 16
-5	2	7	16	4	4 to be inserted
-5	2	7	?	16	16 > 4, shift
-5	2	?	7	16	7 > 4, shift
-5	2	4	7	16	2 < 4, insert 4
-5	2	4	7	16	sorted

?

Thank You