CHAPTER 2

SIMPLE SEARCHING AND SORTING ALGORITHMS

Data Structure and Algorithm

CONTENT

- Searching algorithms
  - Linear Search
  - Binary Search,
- Sorting algorithms
  - Bubble sort
  - Insertion sort
  - Selection sort

SEARCHING ALGORITHMS

- One of the most **common and time consuming** tasks in computer science is the retrieval of target information from huge data.
- Searching is the process of finding the location of the target among a list of objects.
- The two basic search techniques are the following:
  - Sequential/linear search
  - Binary search
- A searching algorithm accepts two arguments as parameters—a target value to be searched and the list to be searched.

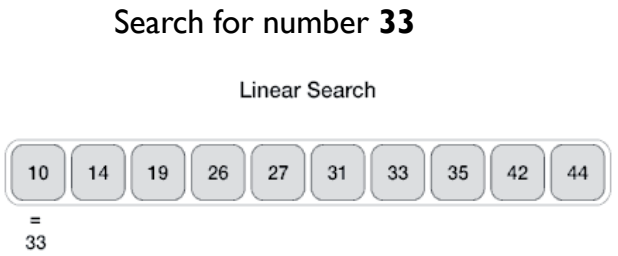
LINEAR SEARCH

- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
  - Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
- For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed.
- The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case n comparisons are needed.

LINEAR SEARCH ALGORITHM

Linear Search (Array A, Value x)

- Step 1: Set i to 1
- Step 2: if i > n then go to step 7 (list is emp
- Step 3: if A[i] = x then go to step 6
- Step 4: Set i to i + 1
- Step 5: Go to Step 2
- Step 6: Print Element x Found at index i and go to step 8
- Step 7: Print element not found
- Step 8: Exit



LINEAR SEARCH C++ IMPLEMENTATION

```
int SeqSearch (int A[], int key, int n)
{
        for(int i = 0; i < n; i++)
        {
                if(key == A[i])
                {
                        cout<<key<<" is found at position =
"<<i<<endl;
                        return i;
                }       }
        cout<<key<<" is not found in the list"<<endl;
        return(-1);
}
```

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.

BINARY SEARCH ALGORITHM

- Works on the principle of **divide and conquer**.
- For this algorithm to work properly, the data collection should be in the **sorted form**.
- Binary search looks for a particular item by comparing the **middle most** item of the collection.
- If a match occurs, then the index of item is returned.
- If the middle item is **greater than the item**, then the item is searched in the sub-array **to the right** of the middle item.
- Otherwise, the item is searched for in the sub-array **to the left** of the middle item.
- This process continues on the subarray as well until the size of the sub-array reduces to one.
- Binary search is a fast search algorithm with run-time complexity of **O(log n)**.

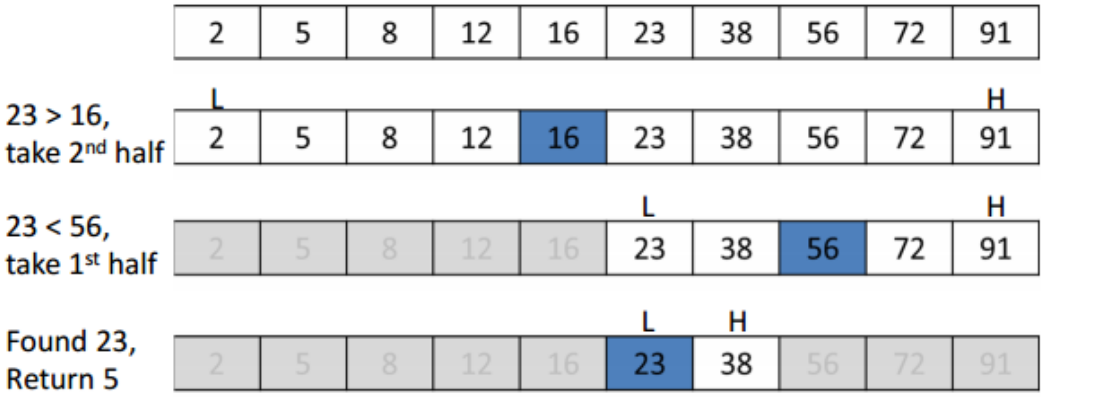
ALGORITHM FOR BINARY SEARCH

- Given an array A of n elements with values or records A0 ... An−1, sorted such that A0 ≤ ... ≤ An−1, and target value T, the following subroutine uses binary search to find the index of T in A

1. Set L to 0 and R to n − 1.
2. If L > R, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor of (L + R) / 2.
4. If A_m < T, set L to m + 1 and go to step 2.
5. If A_m > T, set R to m – 1 and go to step 2.
6. Now A_m = T, the search is done; return m.

EXAMPLE

If searching for 23 in the 10-element array:



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(log2(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

BINARY SEARCH C++ IMPLEMENTATION

```
int Binary_Search_nr(int A[], int key, int n) {
        int low = 0, high = n - 1, mid;
        while(low <= high) {
                mid = (low + high)/2;
                if(A[mid] == key){
                        cout<<key<<" is found at position =
"<<mid<<endl<<endl;
                        return mid;
                }
                else if(key<A[mid]){
                        high = mid - 1;
                }
                else{
                        low = mid + 1;
                }
        }
        cout<<key<<" is not found in the list!"<<endl<<endl;
        return -1;
}
```

SORTING ALGORITHMS

- Sorting is the process of rearranging a sequence of objects so as to put them in some logical order.
- Sorting plays a major role in commercial data processing and in modern scientific computing.
- Type of sorting
  - Internal sorting: all the records to be sorted are kept internally in the main memory
    - uses main memory exclusively
  - External sorting: all the records to be sorted are kept in external files on auxiliary storage
    - uses external memory
- Internal sorting is faster than external sorting.

GENERAL SORT CONCEPTS

- **Sort Order**: -The order in which the data is organized, either *ascending* or *descending*, is called **sort order**.
- **Sort Stability:** -
  - A sorting algorithm is said to be stable, if two objects with equal keys appears in same order in sorted output as they appear in the input list to be sorted.
  - Example: -

John , A	Bob , A	Eric , B	Abebe , B	Daniel , A	Original list
Abebe , B	Bob , A	Daniel , A	Eric , B	John , A	Sorted by name
John , A	Bob , A	Daniel , A	Abebe , B	Eric , B	Sorted by section

Is it stable?

CONT..

- **Sort Efficiency**
  - Sort efficiency is a measure of the relative efficiency of a sort.
  - It is usually an estimate of the number of comparisons and data movement required to sort the data.
- **Passes**
  - During the sorted process, the data is traversed many times.
  - Each traversal of the data is referred to as a sort pass.
  - Depending on the algorithm, the sort pass may traverse the whole list or just a section of the list.
  - In addition, the characteristic of a sort pass is the placement of one or more elements in a sorted list.

CONT..

- **Adaptive sorting algorithm**
  - If the order of elements to be sorted of an input list matters or affects the time complexity of a sorting algorithm, then that algorithm is called **Adaptive** sorting algorithm.

BUBBLE SORT

- Is the oldest and the simplest sort in use. Unfortunately, it is also the slowest.
- Is comparison-based algorithm
  - Each pair of adjacent elements is compared and the elements are swapped if they are not in order.
  - The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm is named for the way smaller or larger elements "bubble" to the top of the list.
- it is too slow and impractical for most problems
- It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of **O(n2)**

BUBBLE SORT ALGORITHM

ALGORITHM

```
1. Let A be the array to be sorted
2. for i = 1 to n - 1
   for j = 0 to n - i
   begin
      if A[j] > A[j+1] then
         Swap A[j] with A[j + 1] as follows
         temp = A[j]
         A[j] = A[j + 1]
         A[j + 1] = temp
      end
   end
3. stop
```

EXAMPLE

- Sort the following list using bubble sort algorithm

76	67	36	55	23	14	06

Array		76	67	36	55	23	14	6
Pass 1	Step 1	67	76	36	55	23	14	6
	Step 2	67	36	76	55	23	14	6
	Step 3	67	36	55	76	23	14	6
	Step 4	67	36	55	23	76	14	6
	Step 5	67	36	55	23	14	76	6
	Step 6	67	36	55	23	14	6	76
Pass 2	Step 1	36	67	55	23	14	6	76
	Step 2	36	55	67	23	14	6	76
	Step 3	36	55	23	67	14	6	76
	Step 4	36	55	23	14	67	6	76
	Step 5	36	55	23	14	6	67	76

Pass 3	Step 1	36	55	23	14	6	67	76
	Step 2	36	23	55	14	6	67	76
	Step 3	36	23	14	55	6	67	76
	Step 4	36	23	14	6	55	67	76
Pass 4	Step 1	23	36	14	6	55	67	76
	Step 2	23	14	36	6	55	67	76
	Step 3	23	14	6	36	55	67	76

Pass 5	Step 1	14	23	6	36	55	67	76
	Step 2	14	6	23	36	55	67	76
Pass 6	Step 1	6	14	23	36	55	67	76
Final sorted array		6	14	23	36	55	67	76

BUBBLE SORT C++ IMPLEMENTATION

```
void bubblesort(int A[], int n){
        int i, j,temp;
        for(i = 1; i < n; i++){
                for(j = 0; j < n - i; j++){
                        if( A[j] > A[j + 1] ){
                                temp = A[j]; // swap A[j]
with A[j + 1]
                                A[j] = A[j + 1];
                                A[j + 1] = temp;
                        }
                }
        }
}
```

ANALYSIS OF BUBBLE SORT

- The number of comparisons made at each of the iterations is as follows:

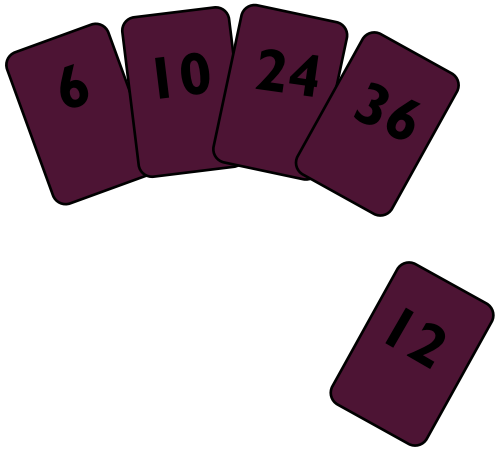
$$(n - 1) + (n - 2) + (n - 3) + \ldots + 1 = n(n - 1)/2$$

- The total number of comparisons is **n(n - 1)/2**, which is **O(n^2)**.
- Hence, the time complexity for each of the cases is given by the following:
  - Average/best/worst case complexity = **O(n^2)**
- **Space complexity:** it is in place algorithm so O(1)
- Stability: - it is stable because in this algorithm equal elements will never be swaped.
- Adaptability: - no/yes ? it can be adaptable if we put early exit condition when there is no swap done in one pass

INSERTION SORT

- This is an in-place comparison-based sorting algorithm.
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

CONT....



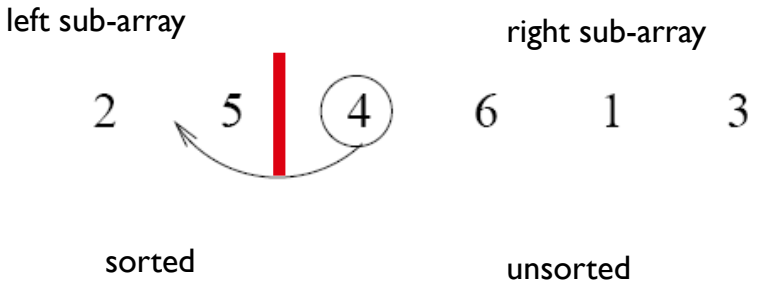
To insert 12, we need to make room for it by moving first 36 and then 24.

INSERTION SORT EXAMPLE

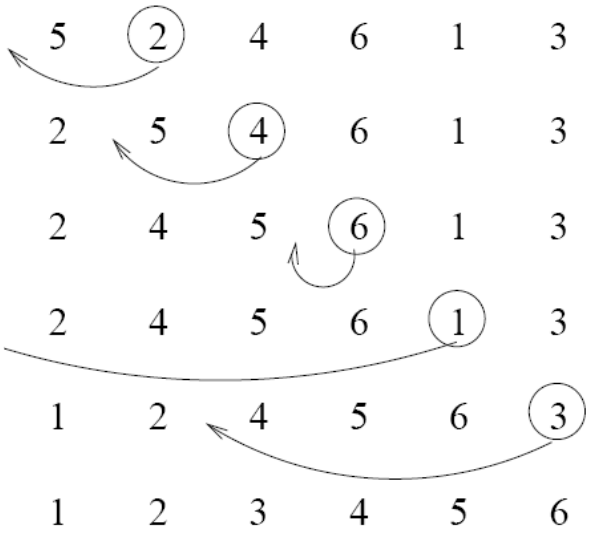
input array

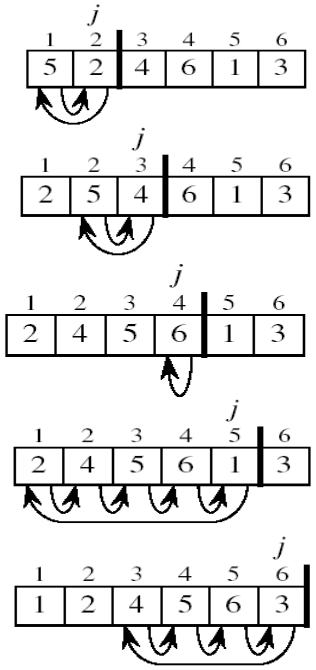
5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:



EXAMPLE


5 2 4 6 1 3
2 5 4 6 1 3
2 4 5 6 1 3
2 4 5 6 1 3
1 2 4 5 6 3
1 2 3 4 5 6


j
1 2 3 4 5 6
5 2 4 6 1 3
j
1 2 3 4 5 6
2 5 4 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
1 2 4 5 6 3

INSERTION SORT ALGORITHM

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

ANALYSIS OF INSERTION SORT

- Although the insertion sort is almost always better than the bubble sort, the time required in both the methods is approximately **the same**.
- The total number of comparisons is given as follows:

  $(n - 1) + (n - 2) + \ldots + 1 = (n - 1) * n/2$

  which is $O(n^2)$.
- Worst Case Analysis– The simplest worst case input is an array sorted in reverse order. **$O(n^2)$**
- Best Case Analysis– if the list is already sorted. **$O(n)$**
- Average case **$O(n^2)$**

ANALYSIS OF INSERTION SORT

- **Space complexity:** it is in place algorithm so O(1)
- **Stability**: - it is stable because in this algorithm equal elements will never be swaped.
- **Adaptability**: - yes because no swapping needed if the initial list sorted completely

PROS AND CONS OF INSERTION SORT

The following are the pros and cons of insertion sort:

- Pros
  1. The insertion sort is an in-place sorting algorithm so the space requirement is minimal.
  2. Good running time for "almost sorted" arrays $\Theta(n)$
  3. It also exhibits a good performance when dealing with a small list
- Cons
  1. The insertion sort is particularly useful only when sorting a list of few items.
  2. The insertion sort does not deal well with a huge list. $\Theta(n^2)$ running time in worst and average case

SELECTION SORT

- In this sorting algorithm the list is divided into two parts,
  - the sorted part at the left end (Initially, the sorted part is empty)
  - the unsorted part at the right end (the unsorted part is the entire list)
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
- This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of **O(n2)**,

EXAMPLE


Selection Sort
1st
12
10
16
11
9
7
2nd
7
10
16
11
9
12
3rd
7
9
16
11
10
12
4th
7
9
10
11
16
12
5th
7
9
10
11
16
12
6th
7
9
10
11
12
16

SELECTION SORT ALGORITHM

Alg.: SELECTION-SORT(*A*)

```
n ← length[A]
for j ← 1 to n - 1
  do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                  then smallest ← i
       exchange A[j] ↔ A[smallest]
```

ANALYSIS OF SELECTION SORT

- The total number of comparisons is as follows:
  - $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$
- Therefore, the number of comparisons for the selection sort is proportional to n^2, which means that it is $O(n^2)$. The different cases are as follows:
  - Average case: $O(n^2)$ Best case: $O(n^2)$ Worst case: $O(n^2)$
- **Space complexity:** it is in place algorithm so O(1)
- **Stability**: - It's not stable, because it swaps elements to random location.
- **Adaptability**: - no because there are same number of swapping no matter initial list order is.