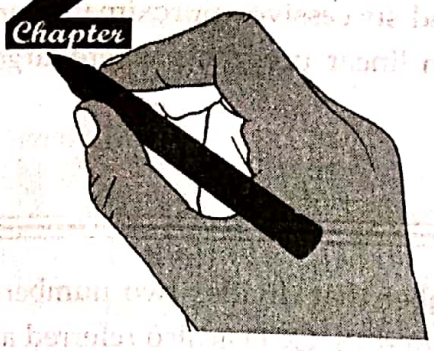
Chapter 2



ITERATIVE ALGORITHMS

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Basic Algorithms: Algorithm for GCD, Fibonacci Number and analysis of their time and space complexity
- Searching Algorithms: Sequential Search and its analysis
- Sorting Algorithms: Bubble, Selection, and Insertion Sort and their Analysis

Introduction

An iterative algorithm executes steps in iterations. It aims to find successive approximation sequence to reach a solution. They are most commonly used in linear programs where lar numbers of variables are involved.

Algorithm for GCD

The greatest common divisor (GCD) is the largest natural number that divides two numbe without leaving a remainder. E.g. GCD (10, 15) = 5 or GCD (12, 18) = 18. GCD is also referred highest common factor (HCF) or greatest common factor (GCF) or greatest common measu (GCM). The Euclidean algorithm is the efficient algorithm to find GCD of two natural numbe Algorithm is named after famous Greek mathematician Euclid.

The Euclidean algorithm is based on the principle that the greatest common divisor of tw numbers does not change if the larger number is replaced by its difference with the small number.

Algorithm to find the GCD of two numbers by Euclid's algorithm

1. Start
2. Read any two number say m and n
3. If n==0, return the value of m as the answer and stop;
4. If m==0, return the value of n as the answer and stop,
5. Divide m by n and assign the value of the remainder to r.
6. Assign the value of n to m and the value of r to n. Go to step 3.
7. Stop

Example: Find the GCD of 270 and 192

Solution:

Iteration 0: A=270, B=192

$A \neq 0$

$B \neq 0$

R=270%192=78

Iteration 1:

A=B=192

B=R=78

R=A%B=192%78 = 36

Iteration 2:

A=B=78

B=R=36

R=A%B=78%36 = 6

Iteration 3:

A=B=36

B=R=6

R=A%B=36%6 = 0

Iteration 4:

A=B=6

B=R=0

Since B==0 so GCD (270, 192) =A=6

⇨ GCD of number 270 and 192 is 6

Pseudo code for GCD

```
GCD(A, B)
{
    if (A==0)
        Print "B as GCD"
    else if (B==0)
        Print "A as GCD"
    else
    {
        while (B!=0)
        {
            R=A%B
            A=B
            B=R
        }
        Print "A as GCD"
    }
}
```

Analysis

Since while loop executes at most n times if n be the size of element B

Then their time complexity is,

$T(n) = O(n)$

Fibonacci Number

The Fibonacci numbers, commonly denoted F_n form a sequence, called the Fibonacci sequence such that each number is the sum of the two preceding ones, starting from 0 and 1. The Fibonacci numbers are generated by setting $F_0=0$, $F_1=1$, and then using the recursive formula

$$F_n = F_{n-1} + F_{n-2}$$

We get the rest. Thus the sequence begins: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34....

Algorithm

1. Start
2. Set first=0, second=1
3. Read term of Fibonacci number say it be n
4. Set i=3
5. While(i<=n)
   - Set temp=first + second
   - Set first=second
   - Set second= temp
   - Increment I by 1 as, i++
6. Print temp as required Fibonacci number
7. Stop

Analysis

Since while loop executes at most n times

So their time complexity is,

T(n)= O(n)

Tracing: Find 9th Fibonacci number

Iteration 1: A=0, B=1 (i=3)

C= A+B = 0+1=1

Iteration 2: A=1, B=1 (i=4)

C= A+B=1+1=2

Iteration 3: A=1, B=2 (i=5)

C= A+B=1+2=3

Iteration 4: A=2, B=3 (i=6)

C= A+B=2+3=5

Iteration 5: A=3, B=5 (i=7)

C= A+B=3+5=8

Iteration 6: A=5, B=8 (i=8)

C= A+B=5+8=13

Iteration 7: A=8, B=13 (i=9)

C= A+B=8+13=21

Thus 21 is the 9th number of Fibonacci series

Searching Algorithms

Linear search algorithm finds given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Algorithm

1. Start
2. Read the search element from the user
3. Compare, the search element with the first element in the list.
4. If both are matching, then display "Given element found!!!" and terminate the function
5. If both are not matching, then compare search element with the next element in the list.
6. Repeat steps 4 and 5 until the search element is compared with the last element in the list.
7. If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function
8. Stop

Pseudo code

```
LinearSearch(A, n, key)
{
        flag=0;
        for(i=0; i<n; i++)
        {
                if(A[i] == key)
                        flag=1;
        }
        if(flag==1)
                Print "Search successful"
        else
                Print "Search un-successful"
}
```

Analysis

Time complexity

Flg=0 takes 1 step

In for loop,

i=0 takes 1 step

i<n takes (n+1) steps

i++ takes n steps

if statement takes n steps

flag =1 takes n steps

if (flag==1) takes 1 step

printf takes 1 step

Last printf takes 0 step since either if statement or else statement executed

Time complexity = 1+1+(n+1)+n+n+n+1+1

= 4n+5

= O(1) x O(n) + O(1)

= O(n)+O(1)

= O(n)

Space complexity,

Variable i take 1 step

Variable n take 1 step

Variable flag take 1 step

Variable key take 1 step

Array A takes n steps

Total space complexity = n + 4 = O(n)+O(1) = O(n)

Sorting

Sorting is the process of ordering elements in an array in specific order e.g. ascending descending on the basis of value, chronological ordering a records, priority order.

Sorting is categorized as internal sorting and external sorting.

1. **Internal sorting:** By internal sorting means we are arranging the numbers within th array only which is in computer primary memory.
2. **External sorting:** External sorting is the sorting of numbers from the external file b reading it from secondary memory.

Let P be a list of n elements $P_1, P_2, P_3, \ldots\ldots\ldots\ldots, P_n$ in memory. Sorting P means arranging th contents of P in either increasing or decreasing order i.e.

$P_1 \leq P_2 \leq P_3 \leq P_4 \leq P_5 \ldots\ldots\ldots\ldots \leq P_n$

There are n elements in the list, therefore there are n! ways to arrange them.

In-place

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list. For example, Insertion Sort and Selection Sorts are in-place sorting algorithms as they do not use any additional space for sorting the list and a typical implementation of Merge Sort is not in-place, also the implementation for counting sort is not in-place sorting algorithm.

Stable

A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key remain sorted on the first key.

Why we using sorting?

We know that searching a sorted array is much easier than searching an unsorted array. This is especially true for people. That is, finding a person's name in a phone book is easy, but finding a phone number without knowing the person's name is virtually impossible. As a result, any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted. The following are some more examples.

- Words in a dictionary are sorted
- Files in a directory are often listed in sorted order.
- The index of a book is sorted
- The card catalog in a library is sorted by both author and title.
- A listing of course offerings at a university is sorted, first by department and then by course number.
- Many banks provide statements that list checks in increasing order by check number.
- In a news paper, the calendar of events in a schedule is generally sorted by date.
- Musical compact disks in a record store are generally sorted by recording artist.
- In the programs printed for graduation ceremonies, departments are listed in sorted order and then students in those departments are listed in sorted order.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. In Bubble sort, each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.

In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.

3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the ends of this pass. The smallest element of the list is placed at the first index of the list.

Characteristics of Bubble Sort

- Large values are always sorted first.
- It only takes one iteration to detect that a collection is already sorted.
- The best time complexity for Bubble Sort is O(n). The average and worst time complexity is $O(n^2)$.
- The space complexity for Bubble Sort is O(1), because only single additional memory space is required.

Tracing: Sort the following data items by using Bubble sort

A[ ]={25, 57, 48, 37, 12, 92, 86, 33}

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	25	48	37	12	57	86	33	92
Pass 2	24	37	12	48	57	33	86	92
Pass 3	24	12	37	48	33	57	86	92
Pass 4	12	24	37	33	48	57	86	92
Pass 5	12	24	33	37	48	57	86	92
Pass 6	12	24	33	37	48	57	86	92
Pass 7	12	24	33	37	48	57	86	92
Pass 8	12	24	33	37	48	57	86	92

Algorithm

Let's look at implementing the optimized version of bubble sort:

1. Start
2. For the first iteration, compare all the elements (n). For the subsequent runs, compare (n – 1) (n – 2) and so on.
3. Compare each element with its right side neighbor.
4. Swap the smallest element to the left.
5. Keep repeating steps 1-3 until the whole list is covered.
6. Stop

Pseudo code

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

Time Complexity

In pass 1: n-1 comparisons are required

In pass 2: n-2 comparisons are required

In pass 3: n-3 comparisons are required

.....

.....

In pass n-1: 1 comparisons is required

Total comparisons: $T(n) = (n-1) + (n-2) + \ldots\ldots + 1$

$= n(n-1)/2$

$= O(n^2)$

Therefore complexity is of order n^2

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array. The array with n elements is sorted by using n-1 pass of selection sort algorithm.

1. In 1st pass, smallest element of the array is to be found along with its index pos. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
2. In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
3. In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].....and so on.

Tracing: Sort the following data items by using Selection sort

A[ ]={25, 57, 48, 37, 12, 92, 86, 33}

Solution:

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	12	57	48	37	25	92	86	33
Pass 2	12	25	48	37	57	92	86	33
Pass 3	12	25	33	37	57	92	86	48
Pass 4	12	25	33	37	57	92	86	48
Pass 5	12	25	33	37	48	92	86	57
Pass 6	12	25	33	37	48	57	86	92
Pass 7	12	25	33	37	48	57	86	92
Pass 8	12	25	33	37	48	57	86	92

Algorithm

1. Start
2. Consider the first element to be sorted and the rest to be unsorted
3. Assume the first element to be the smallest element.
4. Check if the first element is smaller than each of the other elements:
   - If yes, do nothing
   - If no, choose the other smaller element as minimum and repeat step 3
5. After completion of one iteration through the list, swap the smallest element with the first element of the list.
6. Now consider the second element in the list to be the smallest and so on till all the elements in the list are covered.
7. Stop

Pseudo code

```
SelectionSort(A)
{
        for( i = 0; i< n ; i++)
        {
                least =A[i];
                p=i;
                for ( j = i + 1; j < n ; j++)
                {
                        if (A[j] < A[i])
                        {
                                least = A[j];
                                p=j;
                        }
                }
                swap (A[i], A[p]);
        }
}
```

Time Complexity

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ... +2 +1

$= O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array.

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k]. All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Tracing: Sort the following data items by using Insertion sort

A[ ]={25, 57, 48, 37, 12, 92, 86, 33}

Solution:

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
After a[0..1] is sorted (pass 1)	25	57	48	37	12	92	86	33
After a[0..2] is sorted (pass 2)	25	57	48	37	12	92	86	33
After a[0..3] is sorted (pass 3)	25	48	57	37	12	92	86	33
After a[0..4] is sorted (pass 4)	25	37	48	57	12	92	86	33
After a[0..5] is sorted (pass 5)	12	25	37	48	57	92	86	33
After a[0..6] is sorted (pass 6)	12	25	37	48	57	92	86	33
After a[0..7] is sorted (pass 7)	12	25	37	48	57	86	92	33
After a[0..8] is sorted (pass 8)	12	25	33	37	48	57	86	92

Algorithm

1. Start
2. Consider the first element to be sorted and the rest to be unsorted
3. Compare with the second element:
   a. If the second element< the first element, insert the element in the correct position of the sorted portion
   b. Else, leave it as it is
4. Repeat 1 and 2 until all elements are sorted
5. Stop

Pseudo code

```
Insertion( A, n)
{
        for i =1 to n
        {
                temp =A[i]
                j = i-1
                while( j>= 0 && A[j] > temp)
                {
                        A[j+1] = A[j]
                        j = j - 1
                }
        }
        A[j+1] = temp
}
```

Analysis

Insertion sort runs in O(n) time in its best case and runs in $O(n^2)$ in its worst and average cases.

Best Case Analysis

Insertion sort performs two operations: it scans through the list, comparing each pair of elements, and it swaps elements if they are out of order. Each operation contributes to the running time of the algorithm. If the input array is already in sorted order, insertion sort compares O(n) elements and performs no swaps (in the Python code above, the inner loop is never triggered). Therefore, in the best case, insertion sort runs in O(n) time.

Worst and Average Case Analysis

The worst case for insertion sort will occur when the input list is in decreasing order. To insert the last element, we need at most n-1 comparisons and at most n-1 swaps. To insert the second to last element, we need at most n-2 comparisons and at most n-2 swaps, and so on. The number of operations needed to perform insertion sort is therefore: 2 x (1+2+3+.....+(n-2)+(n-1))

$= 2 \times [(n-1)(n-1+1)/2]$

$= 2 \times [n(n-1)/2]$

$= n(n-1)$

$= n^2-n$

$= O(n^2) - O(n)$

$= O(n^2)$

DISCUSSION EXERCISE

1. What is iterative algorithm? How it is differ from recursive algorithm? Explain.
2. Define GCD algorithm. And analyze it
3. Define Fibonacci series. Write an algorithm to find nth term of Fibonacci number.
4. How to analyze Fibonacci algorithm? Find 6th Fibonacci number.
5. What is searching? Describe sequential search algorithm with suitable example.
6. Write down the algorithm for sequential search then analyze it.
7. What is sorting? Describe bubble sort algorithm with suitable example.
8. Write down the algorithm for bubble sort then analyze it.
9. Trace the bubble sort algorithm for following data items,

   A[ ] = {4, 55, 3, 2, 88, 1, 98, 43, 66, 11, 93, 12. 4, 76}
10. What is stable sort? List out any two stable sort techniques and describe them.

11. Write down the working mechanism of insertion sort. Explain their working principle with suitable example.
12. Write down the algorithm for insertion sort algorithm then analyze it
13. Define selection sort algorithm. Write down algorithm for selection sort and analyze it.
14. Compare selection sort and insertion sort technique.
15. Trace the selection sort algorithm for following algorithm,

    A[ ]={3, 4, 55, 11, 2, 34, 98, 34, 33, 23}
16. Trace the insertion sort algorithm for following algorithm,

    A[ ] = {1, 23, 21, 66, 22, 14, 98, 45, 78}
17. Compare bubble sort and selection sort algorithm.
18. Bubble sort is called one of the worst sorting algorithms. Justify
19. Write down the algorithm for insertion sort then find their best case, average case and worst case time complexity.
20. Differentiate between internal and external sort algorithm.

❑❑❑