LAB MANUAL 01

CSC2211 Algorithms

Summer 2019-2020

Courtesy

Part of the content of the lab manual has been taken from previous lab manuals made by different faculty members of CSE. So special thanks to all of them.

TITLE

An Introduction to Design and Analysis of Algorithms

PREREQUISITE

- Have a clear understanding of arrays
- Have a basic idea about the data structures
- Have a basic idea about pseudo-code
- Have a basic idea about sort and search algorithms

OBJECTIVE

- To understand the significance of Algorithm
- To know about Naive Algorithm
- To be able to implement Linear Search Algorithm
- To be able to implement Bubble Sort Algorithm
- To find out the running time of Algorithms
- To be able to solve the exercise and lab work at the end of this manual

THEORY

Algorithm

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

In simple terms, it is possible to say that an algorithm is a sequence of steps which allow to solve a certain task. Now, an algorithm should have three important characteristics to be considered valid:

- **It should be finite:** If your algorithm never ends trying to solve the problem it was designed to solve then it is useless
- **It should have well defined instructions:** Each step of the algorithm must be precisely defined; the instructions should be unambiguously specified for each case.
- **It should be effective:** The algorithm should solve the problem it was designed to solve. And it should be possible to demonstrate that the algorithm converges with just a paper and pencil.

Also, it is important to point out that algorithms are not just used in Computing Sciences but are a mathematical entity. Basic ally algorithm is a recipe for solving problems. Algorithms can be implemented by different programs in different programming languages.

Significance of Algorithm

Algorithms are extremely important while solving any type of computer science problems. Ideally, when faced with a problem, the first thing we should do is draw a flowchart of how we can solve the problem. The flowchart makes the problem easier to understand, for us humans. A flowchart must be completely independent of any programming language. A flowchart can be written/ drawn in your native language. The next step towards solving the problem is developing the algorithm. Now that we have a complete understanding of the problem (by virtue of flowchart), we start writing steps to solve it. These steps are nothing but algorithm. The algorithm is programming language independent. But it is written in a language which is easy for humans to understand and can be converted to working code very quickly.

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.
If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement.
Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.
Naive Algorithm

An algorithm is said to be naive when it is simple and straightforward but does not exhibit a desirable level of efficiency (usually in terms of time, but also possibly memory) despite finding a correct solution or it does not find an optimal solution to an optimization problem, and better algorithms can be designed and implemented with more careful thought and clever techniques. Naive algorithms are easy to discover, often easy to prove correct and often immediately obvious to the problem solver. They are often based on simple simulation or on brute force generation of candidate solutions with little or no attempt at optimization. Despite their inefficiency, naive algorithms are often the steppingstone to more efficient, perhaps even asymptotically optimal algorithms, especially when their efficiency can be improved by choosing more appropriate data structures.

Searching

Searching means to find out or locate any element from a given list of elements. In other words, to identify or locate an element or position of the element from a list of elements is called searching.

Linear Search

In this method we search the target element one by one until we find the element, or we go beyond the list. In linear searching, at first, we take an element from the list and compare it with the target element. If the first element is the target element, then we have found the element and search is successfully completed. On the other hand, if the first element is not the target element, then we take second element from the list and compare t to the target element. If at this stage the second element is the target element, then searching is successfully completed. If the second element is not target element, then repeat the process as for the first and second element. Thus, we shall repeat the process until we find the target element in the list or we go beyond the list.

The Linear Search Pseudo-code is given below

```
LinearSearch(A,n, v)
    1.  i ← 0
    2.  while i<n and A[i]≠v
    3.      i← i+1
    4.  If A[i]≠v
    5.      Output "Not found"
    6.  Else
    7.      Output "found"
```

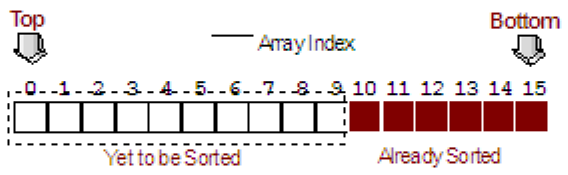
Analyze Runtime using RAM model also find Big O:

Sorting

The process of sorting or creating a linear ordering of a list of objects, is one of the most fundamental of all operations. The objects to be sorted are assumed to be records, one field of which is the sort key. The sort problem is to arrange the objects so that the keys form a monotonic (non-decreasing or non-increasing) sequence.

Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items, and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The bubble sort swaps overlapping pairs so that the largest (smallest) item is at the end of the list each pass. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position.



On each pass i, each pair of items A[0] .. A[n+i] is compared and the largest (or smallest) placed in the lower location such that at the end of the pass, the smallest (or largest) i items are in locations A[n - i + 1] .. A[n]. (i = 6 is shown).

```
//C++ code

void bubble( int a[], int n ) {
  int pass, j, flag;
  for(pass=1;pass<n;pass++) {//break if no swap
            flag = 0;
            for(j=0;j<(n-pass);j++) { //discard the last
            if( a[j]>a[j+1] ) {
                        SWAP(a[j+1],&a[j]); flag = 1;}
      }
            if (flag==0) break;
    }
}
```

Analyze Runtime using RAM model also find Big O:

Counting Sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example. For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index:	0	1	2	3	4	5	6	7	8	9
Count:	0	2	2	0	1	1	0	1	0	0

2) Modify the count array such that each element at each index stores the sum of previous counts.

Index:	0	1	2	3	4	5	6	7	8	9
Count:	0	2	4	4	5	6	6	7	7	7

The modified count array indicates the position of each object in the output sequence.

3) Output each object from the input sequence followed by decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2.

Position of 1 is 2.
Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index.

```
for $i \leftarrow 1$ to $k$
        do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $n$
        do $C[A[j]] \leftarrow C[A[j]] + 1$
for $i \leftarrow 2$ to $k$
        do $C[i] \leftarrow C[i] + C[i-1]$
for $j \leftarrow n$ downto $1$
        do $B[C[A[j]]] \leftarrow A[j]$
           $C[A[j]] \leftarrow C[A[j]] - 1$
```

Analyze Runtime using RAM model also find Big O:

LAB WORK

- **Problem Set 1**
  I. Implement Linear Search Algorithm using C++.
  II. Write a program that finds the maximum marks obtained by the student.
  12 17 19 11 10 8 15 7 13 12 6 14
  III. Print the number of basic number of operations using RAM model

- **Problem Set 2**
  I. Implement Bubble Sort Algorithm using C++.
  II. Write a program that sorts the student list below according to the descending order of CGPA of the students.

Student List (CGPA)
3.75
3.98
2.67
3.11
4.00
3.75
3.88

  III. Print the number if basic number of operations using RAM model

- **Problem Set 3**
  I. Implement Counting Sort Algorithm using C++.
  II. Write a program that sorts the student list below according to the descending order of MARK in a quiz of the students.

Student List (MARK)
9
3
8
7
4
10
9

  III. Print the number if basic number of operations using RAM model