Edited and compiled by : Rinkaj Goyal , Lecturer , USIT, GGSIPU , Delhi-6

Lab Manual and notes -- Part 1

Design and analysis of Algorithm LAB(1-4)

(This Manual is prepared for B.Tech VIth Semester IT students)

Preamble:

I do not claim the originality of the matter provided here, but it's an attempt to provide the best and suitable information available across different resources i.e. Developers tutorials, Reference books, course homepages, etc… under a single roof. By using this manual students can sharpen the skills learned in Lab.

[Rinkaj Goyal]
Lecturer, USIT

Lab exercise 1-2: Analysis of Algorithms

Objectives:

To introduce the concept of efficiency of an algorithm
To study run-time efficiency of an algorithm
To introduce Big-O notation
To determine the Big-O of an algorithm

1. Run Time Efficiency of Algorithms

Efficiency of a software system is generally determined by two factors

(A) algorithms

(B) data structures.

goal is to design efficient algorithms that make good use of computer resources such as memory (space) and processing speed (time).

In this lab we will measure the efficiency of an algorithm by estimating its **run-time**. **Run-time** means the time it takes the CPU to execute an implementation of the algorithm.

Run-time is different than the **wall-clock-time** required to run program since there will be start up time, time-sharing time, I/O time, and so on.

The C++ compiler translates a C++ instruction into a group of machine language instructions (probably about 10 machine instructions for each C++ instruction).

Each computer has a "speed" often measured in MIPS, for millions of instructions per second, at which it can execute instructions. High-powered graphics workstations may run at more than 1000 MIPS. PCs run at 200 MIPS and higher.

There are many ways to measure "run-time" besides "CPU time of execution."

(A) count the number of machine instructions that are executed when the algorithm runs and expect that another algorithm that requires more machine instructions would be less efficient. (This approach is more appealing as it is machine (speed) independent.) But who wants to count machine instructions?

(B) Relatively just count the C++ instructions or the pseudo-code steps that are executed as the algorithm runs. "number of steps" depends on the number **n** of inputs to the algorithm. For instance,in searching of an array, it will usually take less steps if the size **n** of the array is smaller.

We use Approach (B)

2. Determine the Big-O of an Algorithm(Empirical Analysis)

Example 1 : following algorithm to calculate the sum of the n elements in an integer array a[0..n-1].

```
1.  sum = 0
2.  for (i = 0; i < n; i++)
3.       sum += a[i]
4.  print sum
```

How many steps are executed when this algorithm runs?

The answer is: 2*n + 3

Explanation : Line 1 and line 4 execute one time each. Line 3 executes n times since it lies inside the for loop. Line 2 causes the counter i to be changed and tested n + 1 times (one extra for the final test when i = n). Thus the total is:

- Line 1: 1
- Line 4: 1
- Line 3: n
- Line 2: n +1
- Total: 2n +3

The polynomial 2*n + 3 will be dominated by the 1st term as n (the number of elements in the array) becomes very large. In determining the Big-O category we ignore constants such as 2 and 3. Therefore the algorithm above is order n which is written as follows:

f(n) ϵ O(n)

run-time of this algorithm increases at roughly the rate of the size of the inputs (array size) n.

Example 2: This algorithm find the largest element in a square two-dimensional array a[0..n-1][0..n-1]

```
1.  max = a[0][0]
2.  for (row = 0; row < n; row++)
3.      for (col = 0; col < n; col++)
4.          if (a[row][col] > max)  max = a[row][col].
5.  print max
```

Line 1 and 5 execute one time each. For each execution of line 2 or each time **row** changes, line 3 executes n+1 times (or **col** changes n+1 times) and line 4 executes n times. Since the block of statements containing lines 3 and 4 executes n times (row = 0 to n-1) the number of steps executed in this algorithm is

- Line 1: 1
- Line 5: 1
- Line 2: n+1
- Line 3: n*(n+1)
- Line 4: n*(n)
- Total: $2n^2 + 2n + 3$

The n^2 term will dominate the polynomial therefore the Big-O of the algorithm is $f(n) \in O(n^2)$.

Example 3:

This algorithm calculates the sum of the powers of 2 that are less than n.

```
1.  sum = 1
2.  powerTwo = 2
3.  while powerTwo < n
4.      sum = sum + powerTwo
```

5. powerTwo = 2 * powerTwo
6. print sum

Explanation : Let K be the number of times the while loop executes in the above algorithm. The number of steps executed is 4 + 3 * K. From the algorithm, we see that K is the largest integer such that $2^K < n$. In other words, K should be such that 2^K is approximately equal to n. Suppose for example that $n = 2^K$, then $K = \log_2 n$ so we may say K is approximately $\log_2 n$. Thus the number of steps executed by this algorithm is approximately (that's all Big-O's are anyway)

$$\mathbf{f(n) = 4 + 3 * K = 4 + 3 * \log_2 n \in O(\log_2 n)}$$

3. Guidelines For Determining Efficiency

A. In general if the number of steps needed to carry out an algorithm can be expressed as follows:

$$\mathbf{f(n) = a_k\, n^k + a_{k-1}\, n^{k-1} + \ldots + a_1\, n^1 + a_0}$$

then f(n) is a polynomial in n with degree k and $f(n) \in O(n^k)$. To obtain the order of a polynomial function we use the term with the highest degree and disregard any constants and terms with lower degrees.

B.To determine the order of an algorithm look at the loops. If the algorithm contains one loop of the form **for (i = 0; i < n; i++)** then the loop will cause the statements in the loop to be executed n times. Thus the number of steps is approximately n * the number of instructions in the loop. Therefore if the loop causes the most steps to be executed in the algorithm, then the algorithm is O(n). Nested loops such as

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
```

are O(n x n) or $O(n^2)$.

4. Best-case, Average-case, and Worst-case Run-time

Some algorithms don't execute a fixed number of steps even for a fixed value of n. For example, a search algorithm may stop after one step if it finds the value it is searching for on the first comparison. On the other hand, it may search through all elements of the array and still not find the value being sought.

In such cases we usually calculate a **best-case** (the least number of steps executed for an input of size n), **average-case** (the average number of steps executed for any input of size n), and **worst-case** (the most number of steps executed for an input of size n) run-time for the algorithm.

Example : For a sequential search of an array a[0..n] with n elements:

best-case: element searched for appears in 1st position 1 comparison and the item is found so O(1)
average-case: element searched for near middle of array n/2 comparisons are required so O(n)
worst-case: element searched for in last array element n iterations of search loop so O(n)

Lab Exercise 1 :

In code Listing 1,three search functions in C++: binarySearch,sequentialSearch, and awfulSearch. Have been implemented experiment with these three algorithms for different values of n.

Lab Exercise 2 :

Remove the statements that print the array and add code to each function so it will calculate (and print) the number, *f(n)*, of times the comparison statement if (sArray[??] == Item) .... is executed. This value will essentially give the BIG-O for the function. Do a worst-case analysis (that is, search for a value that is not in the array). Create a table containing the number of times the comparison statement is execute for n = 100, 200, 300, 400, and 500 for all three searches. Use the table to guess the Big-O of each algorithm.

Code Listing 1

```
// Code Listing 1
//
// This program contains three algorithms for searching an array.
// They are:
//    1.  Sequential Search
//    2.  Binary Search
//    3.  Awful Search — a terribly inefficient method :-)

#include <iostream>
#include <cstdlib>
```

```
#include <unistd.h>
using namespace std;

const int SIZE = 25;

//Function:    sort()
//
//This function sorts the array using the selection sort.  The
//function receives the array and the size of the array.
void selectionSort(int array[], int arraySize)
{
    int i, j;       //loop counters
    int smallest;  //smallest array element on the current pass
    int temp;       //temporary location used in the swap

    for (i = 0; i < (arraySize - 1); ++i)
    {
        //find the index of the smallest element in                array[i..arraySize -1]
        smallest = i;
        for (j = i + 1; j < arraySize; ++j)
        {
            if (array[j] < array[smallest])
                smallest = j;
        }
        //place the smallest element in the ith position
        if (smallest != i)
        {
            temp = array[i];
            array[i] = array[smallest];
            array[smallest] = temp;
        }
    }
} // end sort




// Function:    binarySearch()
//
// This function looks for a number (valueToFind) using binary search.
// If the number is found, the function returns the index of the number
// in the array.  If  the number is not found the function returns -1.
// The binary search works by repeatedly dividing the array in half.
int binarySearch(int array[], int arraySize, int valueToFind)
{
    bool found = false;          //flags when an element is found
    int left = 0;                //the beginning subscript of the remaining array
    int right = arraySize - 1;   //the ending subscript of the remaining array
    int mid;                     //the midpoint of the array
    int result = -1;             //the index to be returned

    // Search until the valueToFind is found or until the array cannot be
    // divided further.
    while(!found && (left <= right))
    {
        mid = (left + right) / 2;          //find the middle subscript

        if (array[mid] == valueToFind)     //Is the vale at the midpoint?
        {
            result = mid;
            found = true;
        }
```

```
        else if (valueToFind < array[mid])  //Is the value to the left of midpoint?
        {
            right = mid-1;
        }
        else                                // or to the right of midpoint?
        {
            left = mid+1;
        }
    }
    return result;
} // end binarySearch


//Function:    sequentialSearch()
//
//Sequential search just starts at the first element and scans forward till
//it finds a match or reaches the end of the array.
int sequentialSearch(int array[], int arraySize, int valueToFind)
    {
    int result = -1;              //the index where the target was found
    int i;                        //index used in the search
    bool found = false;           //flags when found

    //loop until found or until the end of the array is reached
    for (i = 0; !found && (i < arraySize); ++i)


    {
        //Have we found the value?
        if (array[i] == valueToFind)
        {
            result = i;
            found = true;
        }
    }

    return result;
} // end sequentialSearch


//Function:    awfulSearch()
//
//This is a really awful search.  It looks in the first element, then in the
//first and second, then the first second and third, and so forth.
int awfulSearch(int array[], int arraySize, int valueToFind)
{
    int result = -1;             //the index at which target is found
    bool found = false;          //flags when target is found

    //loop until found or the end of the array is reached
    for (int i = 0; (!found) && (i < arraySize); ++i)
        for (int j = 0; j <= i; ++j)
            if (array[j] == valueToFind)
            {
                found = true;
                result = j;
            }

    return result;
} // end awful_search
```

```
// The main function fills the array with random numbers and sorts it.  The
// user is then asked for a number to find in the array.  This number is
// searched for using each of the search algorithms, and it's position in
// the array is printed on the screen.
int main()
{

    static int sArray[SIZE];      //an array of random integers
    int i;                        //loop counter
    int valueToFind;              //the value to search for
    int position;                 //where the value is found

    srand(getpid());    // seed the random # generator using this proc's PID

    cout << "Filling and sorting array..." << endl;

    //fill the array with random numbers
    for(i = 0; i < SIZE; ++i)
        sArray[i] = (rand() % 10000);

    //sort the array
    selectionSort(sArray, SIZE);

    //print the results
    for (i = 0; i < SIZE; ++i)
        cout << sArray[i]  << '¥n';

    //read in the number which should be searched for
    cout << "Enter number (0 to 9999) to search for.  " ;
    cin >> valueToFind;

    //Perform a sequential search.
    position = sequentialSearch(sArray, SIZE, valueToFind);
    cout << "¥nSequential search: ";
    if (0 <= position)
        cout << position << endl;
    else
        cout << "Not found" << endl;

    // Perform an awful search.
    cout << "Awful search: ";
    position = awfulSearch(sArray, SIZE, valueToFind);
    if (0 <= position)
        cout << position << endl;
    else
        cout << "Not found" << endl;

    // Perform a binary search.
    cout << "Binary search: ";
    position = binarySearch(sArray, SIZE, valueToFind);
    if (0 <= position)
        cout << position << endl;
    else
        cout << "Not found" << endl;

    return 0;
}
```

Lab Exercise 3-4 : Sorting

Objectives:

To study various sorting algorithms
To learn how to analyze the efficiency of different algorithms
To compare the efficiency of three different sorting algorithms

Sorting is a frequent activity in computing and it is important to develop efficient algorithms.. Assume an array a[0..n-1] is to be sorted and swap(x,y) is a function that exchanges two elements stored in x and y.

Simple Sort:

```
for (i=0; i < n; i++)
   for (j=0; j < n; j++)
      if (a[i] > a[j])  swap(a[i], a[j])
```

1 . Different sorting algorithms

Bubble Sort:

```
1.  k = n
2.  sorted = false
3.  While NOT sorted and we haven't exceeded the number of necessary passes
4.  {
5.     k = k - 1
6.     sorted = true              {See if this pass makes any changes}
7.     for (i=0; i<k; i++)
8.        if (a[i] > a[i+1])
9.        {
10.          swap(a[i], a[i+1])
11.          sorted = false       {Wasn't sorted ...}
12.       }
13. }
```

In the "best case" where the array a[0..n-1] is already sorted, the "while" loop will run only once. The "for" loop will iterate n-1 times, so the complexity is roughly n-1 = O(n) in the "best case." In the worst case, the while loop will run n-1 times and the "for" loop runs n-1, n-2, ..., 1 times respectively. The total number of comparisons (line 7) is

$$1 + 2 + 3 + ... + n\text{-}1 = (n\text{-}1)*n/2 = O(n^2).$$

This is also the average time. So, theoretically, in the average and worst cases, the bubble sort is no "better" than the simple sort. Though its magnitude (Big-O) is the same, it has a better run time in general for small and reasonable size lists (as we will see).

Quick Sort

The quick sort has an average case complexity of $O(n \log_2 n)$.

For small n, there is no great difference in $O(n \log_2 n)$ and $O(n^2)$, but for larger values of n, say n > 1000, the difference is significant:

For example, let n=1000:

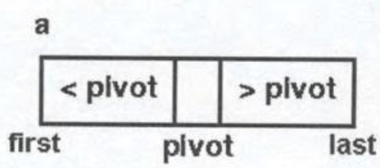
$n \log_2 n = 1000 \log_2 1000 = 1000 * 10 = 10{,}000$

and

$n^2 = 1000 * 1000 = 1{,}000{,}000.$

The basic idea behind the Quick Sort is as follows:

1. pick one element in the array, which will be the pivot.
2. make a pass through the array, called a partition step, re-arranging the entries so that:
    - the pivot is in its proper place.
    - entries smaller than the pivot are to the left of the pivot.
    - entries larger than the pivot are to its right.
3. recursively apply quicksort to the part of the array that is to the left of the pivot, and to the part on its right.



Quick Sort:

```
//This function sorts the items in an array into ascending order
void quickSort (DataType a[], int first, int last)
{
  int pivotIndex;         //index of pivot position after partitioning
```

```
    if (first < last)
    {
       //create the partition by placing the first
       //array element exactly where it should stay
       partition(a, first, last, pivotIndex);

       //sort the two partitions
       quickSort(a, first, pivotIndex - 1)
       quickSort(a, pivotIndex + 1, last)
     }
   }

   //Rearrange elements in a[first..last] so that a[first] is in
   //its final sorted position at pivotIndex, all elements less than
a[first]
   //are in positions less than pivotIndex, and all elements greater
than
   //a[first] are in positions greater than pivotIndex.
   void partition(DataType a[], int first, int last, int& pivotIndex)
   {
      DataType pivot = a[first];      //the pivot is the first element
      int lastRegion1 = first;        //index of last item in region 1

      //move one item at a time until unknown region is empty
      for (int firstUnknown = first + 1; firstUnknown <= last;
++firstUnknown)
      {
         //move item from unknown to proper region
         if (a[firstUnknown] < pivot)
         {
            //item from unknown belongs in region 1
            ++lastRegion1;
            swap (a[firstUnknown], a[lastRegion1]);
         }
      }

      //place the pivot in the proper position and mark its position
      swap (a[first], a[lastRegion1]);
      pivotIndex = lastRegion1;

   }
```

Here is an example of how the "partition" works:

The array a contains:

0	1	2	3	4	5
40	50	20	60	30	70

When the partition() function is activated the first time,

first = 0 and last = 5.
Next pivot = a[0] = 40 (you can see this item in "yellow" above).
Also lastRegion1 = 0

The for loop is started and:
firstUnknown = 1
a[1] < pivot is false

firstUnknown = 2
a[2] < pivot is true
lastRegion1 = 1
swap a[2] with a[1]
The table then becomes:

0	1	2	3	4	5
40	20	50	60	30	70

firstUnknown = 3
a[3] < pivot is false

0	1	2	3	4	5
40	20	50	60	30	70

firstUnknown = 4
a[4] < pivot is true
lastRegion1 = 2
swap a[4] with a[2]
The table then becomes:

0	1	2	3	4	5
40	20	30	60	50	70

firstUnknown = 5
a[5] < pivot is false

the loop is exited
swap a[0] with a[lastRegion1] or a[2]
pivotIndex = lastRegion1 = 2

Partition is finished and the pivot is in the correct position. All of the items to the left (in green) are less than the pivot and all of the items to the right (in blue) are greater than the pivot.

0	1	2	3	4	5
30	20	40	60	50	70

After the return from "partition" **pivotIndex** contains 2 and two recursive

```
calls will be made to the quick sort routine:
    one to sort a[0..1] and
    another to sort the list a[3..5].
```

Like all divide-and-conquer routines, it is important that the two sublists are roughly the same size. For that to happen, the element separating them must belong near the middle of the sorted list.

Better techniques for selecting the pivot do exist. We will not give the argument that "quick sort" is an $O(n \log_2 n)$ algorithm in the average case but it is! However it is $O(n^2)$ in the worst case.

The worst case occurs when the array is already sorted. a[first] will already be in its final sorted position and one of the two sublists will be empty while the other will have one less element than the original.

2. . Efficiency of Sorting Algorithms

C++ has a built-in function called **clock()** This function returns the amount of CPU time (in microseconds) used since the first call to clock(). To determine the time in seconds, the value returned must be divided by the value of CLOCKS_PER_SEC which is defined along with clock() in the **ctime** library.

Lab Exercise 3 :

Test the CPU time used by each algorithm for a random list of

n = 100 integers,

n = 1000 integers,

and n = 10,000 integers.

a routine has been included in Code Listing 2 to generate a specified quantity of random integers. Only the quick sort is done recursively and recursion takes more CPU time so this puts quick sort at a slight disadvantage but it should win

Lab Exercise 4 :

Draw a graph of the results from the above tests using an X-axis of n and a Y-axis of CPU_TIME. Put all three graphs on one coordinate system so we can compare them.

The "clock()" function requires the ctime header file to be included, and a structure of type **clock_t** to be declared.

Read the code Listing to see how "clock()" is used.

Code Listing 2

```
// Code Listing 2
//
// This program compares different sorting algorithms.  It implements quicksort
// bubble sort, and simple sort.  They are all used to sort the
// same array, and the time required for each is printed on the screen.


#include <iostream>
#include <cstdlib>
#include <unistd.h>            // for getpid()
#include <ctime>               // for clock(), CLOCKS_PER_SEC
using namespace std;

const int SIZE = 5000;         //size of the array

//function prototypes
void partition(int a[], int first, int last, int& pivotIndex);
void quickSort (int a[], int first, int last);
void bubbleSort(int a[], int size);
void simpleSort(int a[], int size);
void swap (int&, int&);

// This function is used by each of the sort functions to swap
// two integer values.
void swap (int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

// This function implements the Quick Sort algorithm.  There is a standard C
// function 'qsort()' which does this.  This is a simpler implementation
// to illustrate how the algorithm works.  This function sorts the items in
// an array into ascending order
void quickSort (int a[], int first, int last)
{
   int pivotIndex;          //index of pivot position after partitioning

   if (first < last)
   {
       //create the partition by placing the first
       //array element exactly where it should stay
```

```
        partition(a, first, last, pivotIndex);

        //sort the two partitions
        quickSort(a, first, pivotIndex - 1);
        quickSort(a, pivotIndex + 1, last);
    }
}

//This function partitions elements in a[first..last] so that a[first] is in
//its final sorted position at pivotIndex, all elements less than a[first]
//are in positions less than pivotIndex, and all elements greater than
//a[first] are in positions greater than pivotIndex.
void partition(int a[], int first, int last, int& pivotIndex)
{
    int pivot = a[first];      //the pivot is the first element
    int lastRegion1 = first;        //index of last item in region 1

    //move one item at a time until unknown region is empty
    for (int firstUnknown = first + 1; firstUnknown <= last; ++firstUnknown)
    {
        //move item from unknown to proper region
        if (a[firstUnknown] < pivot)
        {
            //item from unknown belongs in region 1
            ++lastRegion1;
            swap (a[firstUnknown], a[lastRegion1]);
        }
    }

    //place the pivot in the proper position and mark its position
    swap (a[first], a[lastRegion1]);
    pivotIndex = lastRegion1;

}

// This is the simple sort.  It receives the array and its size.  It makes
// a fixed number N-squared) of passes.
void simpleSort(int a[], int size)
{
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size - 1; ++j)
        {
            if (a[j] > a[j+1])
            {
                swap (a[j], a[j+1]);
            }
        }
} // end Sort

// This function is the bubble sort.  It is faster if the array is mostly sorted.  It
// checks after each pass to see if the array is sorted, and exits if it is.
void bubbleSort(int a[], int size)
{
    bool done = false;        //used to exit the loop when sorted

    //as long as a swap is made on a pass, consider
    //the array out of order
    while(!done)
    {
        //assume this is the last pass and the
        //array is in order
        done = true;
```

```

        //compare each successive set of array elements
        //and swap them if they are out of order
        for (int i = 1; i < size; ++i)
            if (a[i] < a[i-1])
            {
                done = false;
                swap (a[i], a[i-1]);
            }
    }
} // end bubbleSort



int main(void)
{
    //initialize the random number seed
    srand(getpid());

    // Create the arrays to be sorted.
    int qArray[SIZE];            //array for the QuickSort
    int bArray[SIZE];            //array for the BubbleSort
    int sArray[SIZE];            //array for the SimpleSort

    // Fill all of the arrays with the same set of random numbers.
    for (int i=0; i<SIZE; ++i)
        sArray[i] = qArray[i] = bArray[i] = rand();

    //perform the simple sort and measure time
    clock_t start = clock();
    simpleSort(sArray, SIZE);
    clock_t stop = clock();

    cout << "simple sort took " << float(stop-start)/CLOCKS_PER_SEC
         << " seconds." << endl;

    //perform the quick sort and measure time
    start = clock();
    quickSort(qArray, 0, SIZE - 1);
    stop = clock();

    cout << "quicksort sort took " << float(stop-start)/CLOCKS_PER_SEC
         << " seconds." << endl;

    //perform the bubble sort and measure time
    start = clock();
    bubbleSort(bArray, SIZE);
    stop = clock();

    cout << "bubble sort took " <<  float(stop-start)/CLOCKS_PER_SEC
         << " seconds." << endl;

    return 0;
}
```