# Experiment : (1)

Aim: study usage of GNU profiler tool and other time related functions. And use it to analyze recursive and iterative solutions of the problems: Fibonacci and Tower of Hanoi.

→ Introduction to GPROF tool.

gprof: gprof is a performance analysis tool used to profile applications to determine where time is spent during program execution.

File: second1.c

```
#include <stdio.h>

void fun2();

void fun1(){
    printf(" first line in fun1 function");
    fun2();
    for(long int i=0; i< 0XFFFFFF; i++);
    printf(" second line in fun1 function");
}
int main(){
    printf("in main function \n");
    fun1();
    return 0;
}
```

a.txt :

| % time | Cumulative seconds | self seconds | calls | self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|

File : second2.c

```
# include <stdio.h>

void fun2()
{
    printf (" first line in fun2 function \n");
    for (long int i=0; i< 1000 00000; i++);
    printf (" second line in fun2 function \n");
}
```

Command :

```
gedit second1.c
gedit second2.c
gcc -Wall -pg second1.c second2.c -o second
./second
gprof second gmon.out > a.txt
```

## Recursive fibonacci Algorithm :

Fibonacci ( size ) :

Input : A 'size' of fibonacci series in Integer.

output : Fibonacci Series of 'size'.

```
if (size == 1) then
    return 0

else if (size == 2) then
    return 1

else
    return (fibonacci (n-1) + fibonacci (n-2))
```

* **Recursive Fibonacci :**

```
#include <stdio.h>
#include <time.h>

int fibonacci (int n) {
    if (n == 0) { return 0; }
  else if (n == 1) { return 1; }
  else {
     return ( fibonacci (n-1) + Fibonacci(n-2));
}

int main ()
{
    int n;
    int i; clock_t t = clock();
  for (i=0; i<n; i++) {
  printf ("%d", fibonacci(i));
  }
  t = clock() - t;
  double taken_time = (double) t) / CLOCKS_PER_SEC;

  printf (" recursive fibonacci takes %f time for
          execution", taken_time);

  return 0;
}
```

## Iterative fibonacci Algorithm :

Fibonacci (n) :

Input : Here 'n' indicates the first 'n' values of fibonacci series to be displayed.

output : Display the series till 'nth' location.

```
if (n=1) then
   write ('0')

if (n=2) then
   write ('0', '1'):

else {
   num1 = 0
   num2 = 0
   write ('0', '1')
   for ( i←3, i<=n, i←i+1)
   {
      new num ← num1+num2
      num1 ←num2
      num2 ← new num
      write (new num)
   }
}
```

## * Iterative fibonacci :

```
#include <stdio.h>
#include <time.h>

void fibonacci (int n)
{
   if (n==1) { printf("0"); }
   if (n==2) { printf ("0, 1"); }
   else {
   int num1 = 0;
   int num2 = 1;
   int new num;

   printf(" 0, 1, ");

   for (int i=3; i<=n; i++)
   {
      new num = num1+num2;
      num1 = num2;
      num2 = new num;
      printf ( new num);
   }
}
```

Comparision Table :

| Total element of series | Time reported by gprof (recursive) | Time reported by gprof (Iterative) | Time reported by timing method (recursive) | Time reported by timing method (Iterative) |
|---|---|---|---|---|
| n = 10 | 0.00 | 0.00 | 0.000025 | 0.000000 |
| n = 30 | 50.83 | 0.00 | 0.036879 | 0.000003 |
| n = 50 | 101.66 | 0.00 | 0.037042 | 0.000009 |

```
int main()
{
    int n;
    clock_t t;
    t = clock();
    fibonacci(n);
    t = clock() - t;
    double taken_time = ((double) t) / CLOCKS_PER_SEC;

    printf(" Iterative fibonacci takes %f time
            for execution", taken_time);
    return 0;
}
```

## Recursive Tower of Hanoi Algorithm :

Tower-of-Hanoi (n, S, D, A) :
Input : Total 'n' number of discks in Integer.

output : Tower of hanoi of 'n' disks.

```
if (n>0) then
  Tower-of-Hanoi (n-1, S, D, A)
  write (" S to D")
  Tower-of-Hanoi (n-1, D, A, S)
```

## * Recursive Tower of Hanoi

```
#include <stdio.h>
#include <time.h>

void ToH (int n, char S, char A, char D)
{
    if(n>0) {
      ToH (n-1, S, D, A);
      printf ("\n %c to %c", S, D);
      ToH (n-1, D, A, S);
    }
}

int main()
{
    int n; clock_t t= clock();
    ToH (n, 'A', 'B', 'C');
    t= clock() - t;
    double taken_time = ((double) t)/CLOCKS_PER_SEC;

    printf (" Tower of hanoi takes %f time for execute",
            taken_time);

    return 0;
}
```

| Total elements (disks) | Time reported by gprof | Time reported by timing method. |
|---|---|---|
| n = 15 | 0.00 | 0.043178 |
| n = 18 | 0.09 | 0.0481459 |
| n = 20 | 0.03 | 1.811148 |
| n = 25 | 0.95 | 59.521751 |

Experiment : (2)

Aim : Do empirical comparision of following algorithms for solving searching problems.
- Sequential search
- Binary search

* Sequential Search.

```
#include <stdio.h>
#include <time.h>

int search (int arr[], int N, int X)
{
    int i;

    for (i=0; i<N; i++)
      if (arr[i]==X)
        return i;

    return -1;
}

int main()
{
    int n;
    int arr[n];
    int x = 1000000;
```

## Sequential Search Algorithm :

Sequential_search ( list, n, value) :

input : array 'list' having total 'n' elements of integer type and 'value' that be found.

output : items locations

```
for (i ← 1 ; i <= n ,  i ← i+1)
{
    if ( list[i] = value) then
      return i
}
```

```
clock_t t;
t = clock();

int result = search(arr, n, x);

t = clock() - t;

double time_taken = ((double)t)/CLOCKS_PER_SEC;

(result == -1)
? printf(" Element is not present");
: printf(" Element is present at index %d \n", result);

printf(" sequential search took %f seconds to
        execute \n", time_taken);

return 0;
}
```

| pass | Number of comparision |
|---|---|
| 1 | (n-1) |
| 2 | (n-2) |
| --- | --- |
| last | 1 |

Number of comparision $= (n-1)+(n-2)+(n-3)+\cdots+1 = \frac{n(n-1)}{2}$

Time complexity = O(n) in worst, and average case.

Time complexity = O(1) in best case

| Total elements | Time reported by gprof | Time reported by timing method |
|---|---|---|
| n = 10000 | 0.00 | 0.00 |

* Binary Search :

```
#include <stdio.h>
#include <time.h>

int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r-l)/2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);

        return binarySearch(arr, mid+1, r, x);
    }
    return -1;
}

int main()
{
    int n;
    int arr[n];
    int x = 1000000;
```

## Binary Search Algorithm :

binary-search( list, left, right, value ):

Input : array 'list' having total 'right+1' elements of integer type and 'value' that be found.

output : items location.

```
if (right >= left) then
{
    mid = left + (right-1) / 2

    if (list [mid] = value) then
       return mid

    if (arr [mid] > x) then
       return binarySearch (list, left, mid-1, x)

    return binarysearch (list, midt1, right, x)
}

return -1
```

```
clock_t t;
t = clock();

int result = binarySearch (arr, 0, n-1, x);

t = clock()-t;

double time_taken = ((double)t)/ CLOCKS_PER_SEC;

(result == -1)
? printf(" Element is not present \n");
: printf(" Element present at index %d \n", result);

printf(" binary search took %f second to execute
        \n", time_taken);

return 0;
}
```

Time complexity = O (log n) in worst and average case.

Time complexity = O(1) in best case.

| Total elements | Time reported by gprof | Time reported by timing method. |
|---|---|---|
| | | |

Experiment : (3)

Aim : Do empirical comparision of following algorithms for solving sorting problems.
- Selection sort
- Insertion sort.

* Selection Sort:

```
void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void selectionSort (int array[], int size)
{
    for (int i=0; i<size; i++)
    {
        int min_idx = i;

        for (int j= i+1; j< size; j++)
        {
            if (array[j] < array[min_idx])
                min_idx = j;
        }
        swap (&array[min_idx], &array[i]);
}
}
```

## Selection Sort Algorithm:

```
Selection_sort (list, n) :

Input : array 'list' having total 'n' elements of
        Integer type.

Output : Sorted array 'list'

for (i:= 1, i<n-1, i:= i+1)
{
    min := i

    for (j:= i+1, j<n, j:= j+1)
    {
        if (list [j] < list [min]) then
        min := j
    }

    swap list [min] and list [i]
}
```

```
int main()
{
    int n;
    int data [n];

    clock_t t = clock();

    SelectionSort (data, n );

    t = clock() - t;
    double time_taken = ((double) t)/ CLOCKS_PER_SEC ;

    printf(" selection sort tooks %f second to execute \n",
            time_taken );
}
```

→ Assume n=5:

pass 1: n=5, comparision = 4 → (n-1)
pass 2: n=5, comparision = 3 → (n-2)
pass 3: n=5, comparision = 2 → (n-3)
pass 4: n=5, comparision = 1 → 1

Time complexity = $(n-1)+(n-2)+\cdots+1$

$= \frac{n(n-1)}{2}$

$= \boxed{O(n^2)}$ (in each case)

| Total elements | Time reported by gprof | Time reported by timing method. |
|---|---|---|
| n=10000 (average) | 0.08. | 0.105411 |
| (worst) | 0.08 | 0.091506 |
| (best) | 0.08 | 0.101210 |
| n=10000000 (best) | 10.46 | 10.470619 |
| (worst) | 9.16 | 9.317602 |
| (average) | 10.23 | 10.344019 |

* Insertion Sort :

```
#include <stdio.h>
#include <time.h>

void insertionsort (int arr[], int n)
{
   int i,key, j ;

   for (i= 1; i<n; i++)
   {
      key = arr[i];
      j= i-1;

      while (j>=0 && arr[j] >key)
      {
         arr[j+1] = arr[j];
         j= j-1 ;
      }
      arr [j+1] = key;
   }
}

int main()
{
   int n ;
   int arr[n];
```

## Insertion Sort Algorithm :

Insertion-sort ( array, n ) :

Input : Array 'array' having total 'n' elements of type Integer

Output : Sorted 'array'.

```
for ( i←1, i<n, i←i+1)
{
    key ← array [i]
     j← i-1
    while ( j>=0 && array [j] >key)
    do
        array [j+1] ← array [j]
         j←j-1
    array [j+1]← key
}
```

```
clock_t t = clock();

insertionSort (arr,n);

t= clock()-t;

double time_taken = ((double)t)/CLOCKS_PER_SEC;

printf ("insertion sort took %f seconds to execute\n",
        time_taken);

return 0;
}
```

| Total elements | Time reported by gprof | Time reported by timing method. |
|---|---|---|
| n = 10000 (average) | 0.04 | 0.057211 |
| (best) | 0.00 | 0.00000 |
| (worst) | 0.10 | 0.116563 |
| n = 100000 (average) | 5.19 | 5.308163 |
| (best) | 0.00 | 0.00000 |
| (worst) | 8.06 | 10.661298 |

EXPERIMENT : (4)

Aim : DO empirical comparision of following algorithms for solving sorting problems.
- Quick sort
- Merge Sort

* Quick Sort:

```
#include <stdio.h>
#include <time.h>

void swap (int arr[], int i, int j)
{
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}

void sort (int arr[], int left, int right)
{
   if (left < right)
   {
      int pivot = arr[left];
      int i = left;
      int j = right + 1;
```

```
    do {
        do { i++; } while (pivot > arr[i]);
        do { j--; } while (pivot < arr[j]);
        if (i<j) { swap(arr, i, j); }
    } while (i<j);

    if (i>j) { swap(arr, left, j); }

    sort(arr, left, j-1);
    sort(arr, j+1, right);
  }
}

int main()
{
    int n;
    int arr[n];

    int left = 0;
    int right = n-1;

    clock_t t = clock();
    sort(arr, left, right);

    t = clock() - t;
    double time_taken = ((double)t) / CLOCKS_PER_SEC;
    printf(" quick sort took %.f seconds to execute \n",
            time_taken);
}
```

| Total elements | Time reported by gprof | Time reported by timing method. |
|---|---|---|
| n = 10000 | 0.06 | 0.089264 |
| (best) | 0.00 | 0.000597 |
| (avg) | 0.08 | 0.085529 |
| (worst) | | |
| n = 100000 | 8.54 | 8.901078 |
| (best) | 0.00 | 0.017342 |
| (avg) | 8.41 | 8.823798 |
| (worst) | | |

* merge sort :

```
#include <stdio.h>
# include <time.h>
# include <stdlib.h>

void merge (int a[], int l, int mid, int h)
{
    int i = l, j = mid + 1, k = l ;
    int b[h+1] ;

    while (i <= mid && j <= h)
    {
        if (a[i] < a[j])
            b[k++] = a[i++];
        else
            b[k++] = a[j++] ;
    }

    for ( ; i <= mid ; i++)
     b[k++] = a[i];

    for ( ; j <= h ; j++)
     b[k++] = a[j] ;

    for (i = l ; i <= h ; i++)
      a[i] = b[i];
}
```

## Merge Sort Algorithm :

merge-sort ( array , left, middle, right)

Input: 'array' of size 'n' with Integer elements.
declare array's 1st element as left,
last as 'right' and middle element.

output : sorted array

```
i ← left, j ← middle+1, K ← left
array 1 [right+1]

while (i<= middle && j<= right)
do
    if(array[i] < array[j]
        array1[K+1]:= array[i+1]
    else
        array1[K+1]:= array[j+1]

for ( , i<= middle, i← i+1)
    array1[K+1] ← array[i]

for ( , j<= Right, j← j+1)
    array1[K+1] = array[j]

for ( i← left, i<= right, i← i+1)
    array[i] = array1[i]
```

```
void merge-sort (int a[], int l, int h)
{
    if (l<h)
    {
        int mid = (l+h)/2;
        merge-sort (a, l, mid);
        merge-sort (a, mid+1, h);
        merge (a, l, mid, h);
    }
}

int main()
{
    int n;
    int a[n];

    for (int i=0; i<n; i++)
      a[i] = rand()%n;

    clock_t t;
    t = clock();
    merge-sort (a, 0, n-1);
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;

    printf ("insertion sort took %f seconds to execute\n",
            time_taken);
}
```

| number of element | Time reported by gprof | Time reported by timing method. |
|---|---|---|
| n = 10000 (best) | 0.00 | 0.0000 |
| (avg) | 0.00 | 0.000899 |
| (worst) | 0.00 | 0.0000 |
| n=100000 (best) | 0.02 | 0.047061 |
| (avg) | 0.02 | 0.024641 |
| (worst) | 0.01 | 0.020433 |