ESc101 : Fundamental of Computing

I Semester 2008-09

Lecture 37

- Analyzing the efficiency of algorithms.
- Algorithms compared
  - Sequential Search and Binary search
  - GCD_fast and GCD_slow
  - Merge Sort and Selection Sort

Problem 1 : Searching

Comparing Searching algorithms

Searching for an element **x** in a sorted array of **n** numbers.

- Sequential Search
- Binary search

Experimental observation : Binary search has been found to be much faster than sequential search. (given as assignment during Lab 11 for Thursday and Friday)

Problem 2 : Computing GCD of two positive numbers

Comparing Two GCD algorithms

We gave two algorithms for GCD computation long back in this course.

- GCD_fast : based on %
- GCD_slow : based on subtraction

The code of these algorithms is shown on the following page.

Why is GCD_fast faster than GCD_slow ?

```
//Assume m>=n
int GCD_fast(m,n)
{  while(n!=0)
   {  rem = m%n;
      m = n;
      n = rem;
   }
   return m;
}
--------------------------------
int GCD_slow(m,n)
{  while(n!=0)
   {   diff = m-n;
       if(diff>=n) m = diff
       else {m=n; n = diff;}
   }
return m;
}
```

Problem 3 : Sorting

Given an array storing **n** numbers, sort them

Comparing Three sorting algorithms

Experimental Observations

- **Quick sort** is more efficient than **merge sort**
- **Merge sort** is more efficient than **Selection Sort**

The code is given in **Three_sorting_algos.java**

What is the reason for different running times?

Given that all algorithms (for searching, GCD, sorting)

- have same input and output
- are executed in same environment (machine, operating system)

We need to analyze the number of steps/instruction taken by each algorithm for a problem

Algorithm design is incomplete until you analyze its running time

How many steps/instructions are executed by the following loop ?

```
for(int i=1; i<=n; i=i+1)
{
    sum = sum + i;
}
```

No. of Steps =

How many steps/instructions are executed by the following loop ?

```
for(int i=1; i<=n; i=i+1)
{
    sum = sum + i;
}
```

No. of Steps = $1 + 3n + 1$

How many steps/instructions are executed by the following loop ?

```
for(int i=1; i<=n; i=i+1)
{
    sum = sum + i;
}
```

For sake of simplicity, we can say that

No. of Steps = $an + b$, for some positive constants a, b

How many steps/instructions are executed by the following loop

```
for(int n=1;n<=m;n=n+1)
{
    for(int i=1; i<=n; i=i+1)
    {
      sum = sum + i;
    }
}
```

No. of Steps =

How many steps/instructions are executed by the following loop

```
for(int n=1;n<=m;n=n+1)
{
    for(int i=1; i<=n; i=i+1)
    {
      sum = sum + i;
    }
}
```

No. of Steps = $1 + m + \sum_{n=1}^{n=m}(1 + 3n + 1) + m = 3/2m^2 + 11/2m + 1$

How many steps/instructions are executed by the following loop

```
for(int n=1;n<=m;n=n+1)
{
    for(int i=1; i<=n; i=i+1)
    {
      sum = sum + i;
    }
}
```

For sake of simplicity, we can say that

No. of Steps = $am^2 + bm + c$, for some constants a,b,c

Analysis of Number of instructions of an algorithm

How many instructions are executed ...

- to search a number in an unsorted array storing n numbers.
- to search a number in a sorted array of m numbers.
- to sort n numbers by selection sort.
- to sort n numbers by merge sort.

Analysis of Number of instructions of an algorithm

How many instructions are executed ...

- to search a number in an unsorted array storing n numbers.
- to search a number in a sorted array of m numbers.
- to sort n numbers by selection sort.
- to sort n numbers by merge sort.

Observation : it is function of input size

We shall focus on worst case number of instructions taken by an algorithm

No. of Instructions executed during Sequential Search on n numbers

- For sequential search, you can write a for loop for the sequential search which iterates n times in the worst case.
- In each iteration, we perform constant number of instructions

No. of instructions in the worst case :

$$cn \quad \text{for some constant } c$$

No. of Instructions executed during Binary Search

Given that array A is sorted.

```
public static boolean Bin_search(int[] A, int x)
{   int left =0;
    int right=A.length-1;
    boolean Is_found = false;    int mid;

    while(Is_found==false && left>right)
    {
        mid = (left+right);
        if(A[mid]==x) Is_found=true;
        else if(A[mid]>x) right = mid-1;
        else left = mid+1;
    }
    return Is_found;
}
```

Analysis of Binary Search

- There are four instructions before entering the while loop.
- Number of instructions in each iteration of while loop is at most 5.
- After each iterations of the while loop, the search domain (A[left]..A[right]) reduces by at least a factor of 2
- Total number of iterations of loop : $\log_2 n$.

Hence the number of instructions in the worst case =
$4 + 5\log_2 n = a\log_2 n + b$, for some constants a, b.

Hence Binary Search is exponentially faster than sequential search

Why is GCD_fast faster than GCD_slow ?

```
//Assume m>=n
int GCD_fast(m,n)
{  while(n!=0)
   {  rem = m%n;
      m = n;
      n = rem;
   }
   return m;
}
--------------------------------
int GCD_slow(m,n)
{  while(n!=0)
   {   diff = m-n;
       if(diff>=n) m = diff
       else {m=n; n = diff;}
   }
return m;
}
```

Analysis of GCD_fast

We shall bound the number of iterations of the while loop.

```
//Assume m>=n
int GCD_fast(m,n)
{  while(n!=0)
   {  rem = m%n;
      m = n;
      n = rem;
   }
   return m;
}
```

Observation : After an iteration $(m, n) \longrightarrow (n, m\%n)$.

Analysis of GCD_fast

Observation : After an iteration $(m, n) \longrightarrow (n, m\%n)$.

Consider any iteration.

- **Case 1 :** $m > \frac{3}{2}n$

- **Case 2 :** $m \leq \frac{3}{2}n$

Analysis of GCD_fast

Observation : After an iteration $(m, n) \quad \longrightarrow \quad (n, m\%n)$.

Consider any iteration.

- **Case 1 :** $m > \frac{3}{2}n$

  Inference : after the iteration

  the new value of $m < \mathbf{\frac{2}{3}}$ times old value of m

- **Case 2 :** $m \leq \frac{3}{2}n$

Analysis of GCD_fast

Observation : After an iteration $(m, n) \longrightarrow (n, m\%n)$.

Consider any iteration.

- **Case 1 :** $m > \frac{3}{2}n$

  Inference : after the iteration

  the new value of $m < \mathbf{\frac{2}{3}}$ times old value of m

- **Case 2 :** $m \leq \frac{3}{2}n$

  Inference : after the iteration

  The new value of $n \leq \mathbf{\frac{1}{2}}$ times old value of n

Analysis of GCD_fast

- It can be seen that once m or n becomes less than or equal to 2, at most one more iteration will be executed.
- Hence, based on previous slide the number of iterations of while loop is at most $\log_{3/2} m + \log_2 n$.

Analysis of GCD_slow

```
//Assume m>=n
int GCD_slow(m,n)
{  while(n!=0)
   {    diff = m-n;
        if(diff>=n) m = diff
        else {m=n; n = diff;}
   }
return m;
}
```

The worst case : m is much larger than n.

For example $m = 10000000002$, $n = 2$.

It follows from the code that the algorithm will perform m/n iterations which is close to m when n is a small number.

Comparing GCD_fast and GCD_slow

In the worst case,

GCD_fast is exponentially faster than GCD_slow.

Comparing Selection Sort and Merge Sort

No. of instructions executed in Selection Sort on n numbers

Number of instructions taken in Selection Sort

```
int  index_of_smallest_value(int[] A,int i)
//returns integer j such that A[j] is smallest among A[i], A[i+1],...

SelectionSort(int [] A)
{

    for(int count=0;count<A.length;count=count+1)
        {
            int j = index_of_smallest_value(A, count);
            if(j != count)
                swap_values_at(A,j,count);
        }
}
```

It follows easily that `index_of_smallest_value` takes $c(n - i)$ instructions in the worst case for some constant c.

Number of instructions taken in Selection Sort on n numbers

- There are ???? iterations of the for loop
- Number of instructions taken to find ith smallest element = ????

Number of instructions taken in Selection Sort on n numbers

- There are $n-1$ iterations of the for loop
- Number of instructions taken to find ith smallest element = ????

Number of instructions taken in Selection Sort on n numbers

- There are $n-1$ iterations of the for loop
- Number of instructions taken to find ith smallest element = $c(n-i)$ for some constant c.

Number of instructions taken in Selection Sort on n numbers

$$\sum_{0 \leq i < n-1} c(n-i) = c\frac{n(n-1)}{2}$$

Let us analyse Merge Sort on n numbers

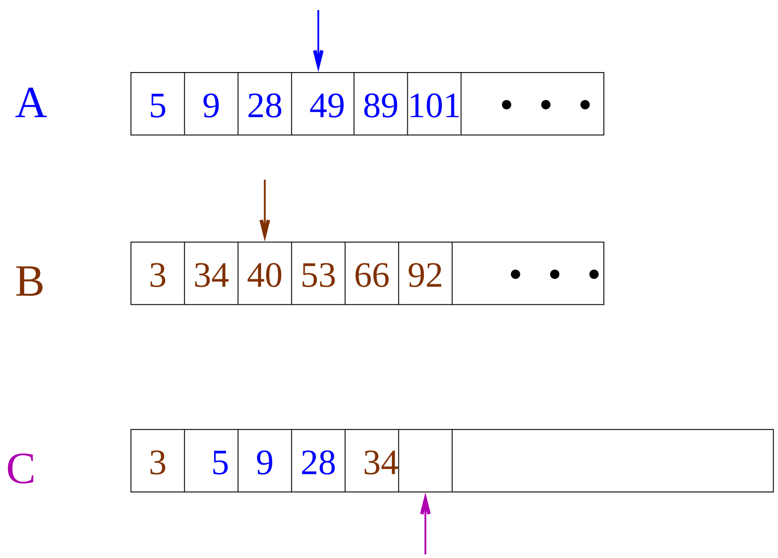
It uses a method to merge two sorted arrays

number of instructions for merging two sorted arrays of size n

Recall that the algorithm proceeds like :

start scanning A and B from left, compare two elements of A and B, copy the smaller one into C and continue ...

Merging two sorted arrays



Number of instructions : cn for some constant c

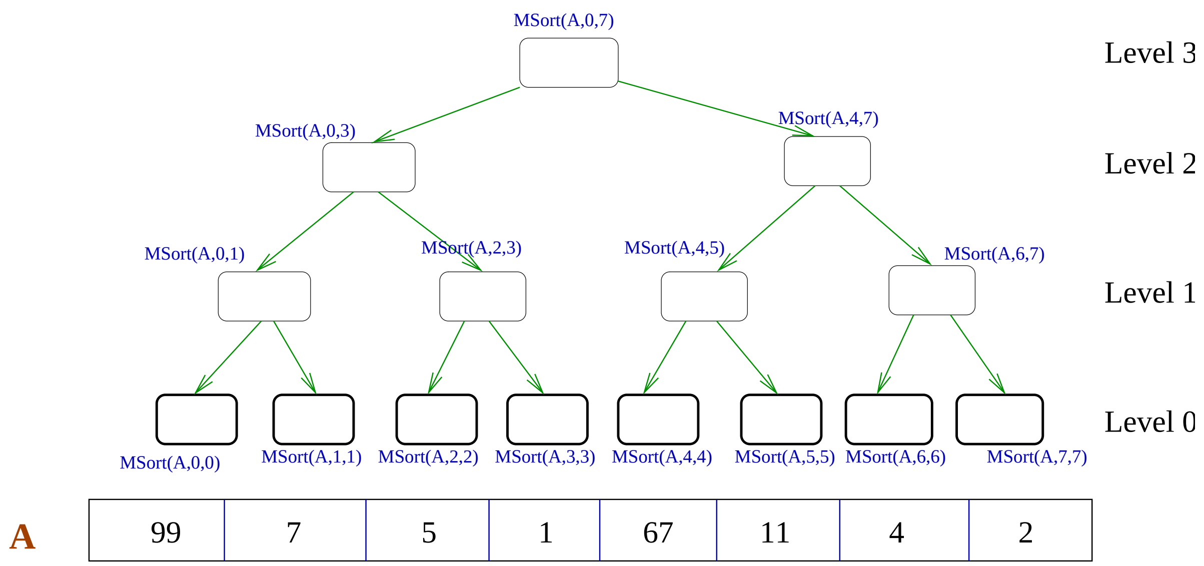
Number of instructions taken in Merge Sort

```
public static void mergesort(int[] A, int left, int right)
    {  if(left!=right)
        {   int mid = (left+right)/2;
            mergesort(A, left, mid);
            mergesort(A, mid+1, right);
            merge(A,left,mid,right);
        }
    }
```

Each call of mergesort does two tasks

- Invoking two calls recursively
- Merging of two sorted portions of array A

Number of instructions taken in Merge Sort



Number of instructions taken in Merge Sort

- We perform cn instructions at each level of the recursion tree.
- There are $\log_2 n$ levels in the tree.

Hence in the worst case there are $cn \log_2 n$ instructions executed in Merge Sort on an array of size n.

Alternate analysis of Merge Sort

Let $T(n)$ be the number of instructions executed by merge sort on n numbers. The following recurrence captures the running time of merge sort exactly.

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ cn + 2T(n/2) & \text{otherwise} \end{cases}$$

Here cn is the no. of instructions for merging two halves of the array.

Alternate analysis of Merge Sort

Gradually unfold the recurrence.

$$
\begin{aligned}
T(n) &= cn + 2(cn/2 + 2T(n/2^2)) \\
&= cn + cn + 2^2T(n/2^2) \\
&= cn + cn + cn + ... + 2^iT(n/2^i) \\
&= cn + cn + cn + ... \log_2 n \text{ terms...} \\
&= cn \log_2 n
\end{aligned}
$$

Efficiency of Quick Sort to be analysed in next lecture class

Please come to Wednesday lecture with any question/doubt