Unit 1: Foundation of Algorithm analysis

Properties of algorithm:

\* Finite number of steps.

\* Executing the steps(instructions) in finite time to complete.

Characteristics of algorithms:

\* Input/Output: every algorithm must take input value and produce output value.

\* Correctness: an algorithm should produce output according to the requirement.

\* Finiteness: algorithm should come to halt after a finite number of steps of instructions are executed.

\* Feasibility: each instruction should be feasible to execute.

\* Flexibility: algorithm must be modifiable as per the problem statement.

\* Efficient: algorithm should use less computation and memory.

\* Definiteness: each line should be well defined.

Random Access Machine(RAM) model:

\* RAM model is a model to count the steps in algorithm in order to analyze the complexity.

\* Basic operations(+,-,\*,/) as 1 step.

\* Memory reference(R/W) as 1 step.

\* Loops are not basic operations, so step count depends on loop.

Example:

int sum(int i){

int total=0; 1

for(i=1,i<=n,i++){ 1+(n+1)+n

total=total+(i\*i); 3(1+1+1)

}

return total; 1

}

total = 1+1+(n+1)+n+3(1+1+1)

= 5n+4

≈ O(n)

Space and Time Complexity:

\* Time complexity of an algorithm is the total time it takes to solve a problem. This is measured in terms of computational steps.

\* Space complexity of an algorithm is the total memory space it takes to store data required to solve a problem. This is measured in terms of data variables used during computation.

Best case complexity: It gives the lower bound on the running time of the algorithm for any instance of input. This means the algorithm can never have lower running time than the best case for particular class of problem.

Worst case complexity: It gives the upper bound on the running time of the algorithm for any instance of input. This means no input can overcome the running limit posed by the worst case complexity.

Average case complexity: This gives you the average number of steps required on any set of inputs.

Lower bound < Average case < Upper bound

Time Complexity Analysis:

\* Time complexity for simple operations takes 1 step. (add,assign,etc)

\* Statements like scanf(), printf(), return takes very small amount of time which will not impact the complexity of our algorithm so it is neglected.

\* For loop conditions we can assume following complexities:

1) For case for(i=0;i<n;i++), the loop runs from 0 to n with

increment of 1 so complexity is O(n)

2) For nested loops, time complexity is O(nk) where k is the

number of loops

3) For case for(i=0;i<n;i\*s), time complexity is O(logsn)

Space Complexity Analysis:

\* Space complexity is the total memory references used by the algorithm.

\* If the memory references(variables) used by algorithm is constant (ie. 1,2,3,…,n) the space complexity is O(1).

\* If an array is taking “n” memory references then the space complexity is O(n).

Q1. Find the detailed analysis of the following factorial algorithm.

#include <stdio.h>

int main(){

int i,n,fact=1;

printf(“enter no. to calculate factorial\n”);

scanf(“%d”,&n);

for(i=1;i<=n;i++){

fact=fact\*i;

}

printf(“factorial of %d is %d”,n,fact);

return 0;

}

for time complexity,

variable assignment cost = 1

loop cost = (1+(n+1)+1)+n(1+1)

simple operation cost = 1

total time complexity = 1+3+n+2n

= 4+3n

= ~~O(4)~~+O(3n)

= O(n) (3 is not necessary)

for space complexity,

total number of variables = 3 (i,n,fact)

total time complexity = O(3)

Q2. Find the detailed analysis of the following algorithm.

#include <stdio.h>

int main(){

int a[]={5,2,4,6,3,1};

for(j=2;j<=a.lenght();j++){

key=a[j];

i=j-1;

while(i>0 && a[i]>key){

a[i+1]=a[i];

i+=1;

}

a[i+1]=key;

}

}

for time complexity,

variable assignment cost = 1

loop cost = 1+(n+1)+1+n(1+2+n(1+1)+n(1+2)+1)

total cost = 4+n+n+2n+n+2n+2n2+n2+2n2+n

= 4+7n+5n2

as 5n2 is the biggest contributor,

time complexity = O(n2)

for space complexity,

total variables = 6+1+1+1

= 9

space complexity = O(9)

= O(1)

Aggregate Analysis:

\* It determines the upper bound T(n) on the total cost of sequence of n operations, then calculates the average cost to be T(n)/n.

\* Aggregate analysis has two steps,

a) we show that a sequence of operations take T(n) time in worst

case

b) we show the average with T(n)/n

Ex:

In a stack push() puts element on the top of the stack and pop() takes the top element of the stack and returns it. The operations are both constant time, so for n operation the complexity is O(n).

=> aggregate analysis = T(n)/n

= O(~~n~~)/~~n~~

= O(1)

Mathematical Foundations:

1) Exponents:

xa.xb = xa+b

xa/xb = xa-b

(xa)b = xab

xn+xn = 2xn

2n+2n = 2n+1

2) Logarithms:

logab = logcb/logca ; c>0

log(ab) = log(a)+log(b)

log(a/b) = log(a)-log(b)

log(ab) = blog(a)

log(x) < x ~~v~~ x>0

log21 = 0, log22=1 , log21024=10

alog(bn) = nlog(ba)

3) Series

summation i from i=0 to n = n(n+1)/2

summation i2 from i=1 to n = (n(n+1)(n+2))/6

summation ai from i=0 to n = 1/(1-a);if 0 < a < 1

summation 2i from i=0 to n = 2n+1-1

Asymptotic Notations:

The complexity analysis of an algorithm is tee mathematical function of the size of the input(best,worst,average). Analyzing algorithms in terms of bound is easier, thus concept of asymptotic notation is required.

1)Big oh Notation:

When we have only asymptotic upper bound then we use O notation. Mathematically, a function f(n) is said to be Big oh of another function g(n), if there exists two constraints xo and c\* such that,

f(n)<=c\*g(n) ∀ x>xo

Here, f(n)=O(g(n)) and g(n) is the upper bound of f(n)

Some properties of Big oh notation are:

a)Transitivity: f(n)=O(g(n)) & g(n)=O(h(n)) then f(n)=O(h(n))

b)Reflexivity: f(n)=O(f(n))

Q1. Prove that f(n)=O(g(n)) if f(n)=3n2+4n+7 and g(n)=n2

=> let us suppose no as 1, then the value of c can be assumed as 14

f(n) =3n2+4n+7

cg(n)=14n2

∴ f(n)<14n2

thus,

f(n)=O(g(n))

≃O(n2)

Q2. Prove that nlog(n3) is O(√n3)