Lec. 4

Analysis of Recursive Algorithms

A **recursive** function is a function that is defined in terms of itself. Similarly, an algorithm is said to be **recursive** if the same algorithm is invoked in the body. An algorithm that **calls itself** is direct recursive.

These recursive algorithms are extremely powerful, but even more importantly, need more times and storage than ***iterative*** algorithms (that consist of iterative instruction like ***for, while,*** and ***repeat***), therefore , recursive algorithms are worse than iterative algorithms because they are used the stack, but they are more clearly than an iterative.

When analyzing a recursive function for its step count (analysis its running time), we often obtain a recursive formula. These recursive formulas are referred to as ***recurrence relations*** which are solved by repeated substitutions (***iterative substitution***) method.

Ex. 6:

Find the summation of one dimension array's elements.

```
Algorithm Rsum(A,n):
Input: a positive integer n and an array A indexed
       from 1 to n.
Output:S,the sum of the numbers in A.
1. if n ≤ 0 then
2.   S ← 0
3. else
4.   S ← Rsum(A,n-1)+ A[n]
5. end if
6. return S
```

The instance characteristics of this problem is **n** (number of elements).

Space Complexities:

Total space complexity of recursive functions = cells number for each calling * number of calling

Each call to *Rsum* function needs two cells (one for variable S and the other one for return address).

number of calling (depth of recursion) = | First size in first calling _ Final size in last calling | + 1

= | n-0 | +1 = n+1

Thus space complexities:

$S_{Rsum}(n) = 2(n+1) = 2n+2$ This is linear formula.

This space inconstant and dependents on instance characteristics (n).

Time complexities (step counts):

The time is formulated in a recursive formula and then using iterative substitution to solve it.

$$T_{Rsum}(n) = \begin{cases} 3 & , \quad n \le 0 \\ 3 + T_{Rsum}(n-1) & , \quad n > 0 \end{cases}$$

$T_{Rsum}(n) = 3 + T_{Rsum}(n-1)$

$= 3 + [3+ T_{Rsum}(n-2)]$

$= 2(3) + T_{Rsum}(n-2)$

$= 2(3) + [3+T_{Rsum}(n-3)]$

$= 3(3) + T_{Rsum}(n-3)$

.

.

.

$= m(3) + T_{Rsum}(n-m)$

Stop's condition is n-m=0 ⇨ m=n

$T_{Rsum}(n) = 3n + T_{Rsum}(0)$

$= 3n + 3$ This is linear formula.

Ex. 7:

Find factorial number n! .

Algorithm *Rfactorial* (n):
Input: a positive integer n.
Output: the result of n! .
1. if n=0 then
2. return 1
3. else
4. return n * *Rfactorial* (n-1)

The instance characteristics of this problem is **n**.

Space Complexities:

Each call to *Rfactorial* function needs one cell for return address.

number of calling (depth of recursion) $= |n-0| + 1 = n+1$

Thus space complexities:

$S_{Rfactorial}(n) = 1 * (n+1) = n+1$.

Time complexities (step counts):

$$T_{Rfactorial}(n) = \begin{cases} 2 & , n = 0 \\ 2 + T_{Rfactorial}(n-1) & , n > 0 \end{cases}$$

$T_{Rfactorial}(n) = 2 + T_{Rfactorial}(n-1)$

$= 2 + [2 + T_{Rfactorial}(n-2)]$

$= 2(2) + T_{Rfactorial}(n-2)$

$= 2(2) + [2 + T_{Rfactorial}(n-3)]$

$= 3(2) + T_{Rfactorial}(n-3)$

.
.
.

$= m(2) + T_{Rfactorial}(n-m)$

Stop's condition is $n-m=0 \Rightarrow m=n$

$T_{Rfactorial}(n) = 2n + T_{Rfactorial}(0) \Rightarrow 2n + 2$ This is linear formula.

Ex. 8:

Multiplication of two positive numbers.

Algorithm *Rmult* (n, m):
Input: two positive integer numbers n, m.
Output: the result of multiplying n by m.

1. if m=0 then
2. Z ← 0
3. else
4. Z ← *Rmult* (n, m-1) + n
5. end if
6. return Z

The instance characteristics of this problem is **m**.

Space Complexities:

Each call to *Rmult* function needs two cells for variable Z and return address.

number of calling (depth of recursion) $= \left| m-0 \right| + 1 = m+1$

Thus space complexities:

$S_{Rmult}(m) = 2 * (m+1) = 2m+2$.

Time complexities (step counts):

$$T_{Rmult}(m) = \begin{cases} 3 & , \ m = 0 \\ 3 + T_{Rmult}(m-1) & , \ m > 0 \end{cases}$$

$T_{Rmult}(m) = 3 + T_{Rmult}(m-1)$

$= 3 + [3+ T_{Rmult}(m-2)]$

$= 2(3) + T_{Rmult}(m-2)$

$= 2(3) + [3+ T_{Rmult}(m-3)]$

$= 3(3) + T_{Rmult}(m-3)$

.
.
.

$= k(3) + T_{Rmult}(m-k)$

Stop's condition is m-k=0 ⇨ m=k

T_{Rmult} (m) = 3m + T_{Rmult} (0)

= 3m + 3 .

H.W. :

Analyze the following problem using recursive schema:

1)) Algorithm for finding the summation of elements of two dimensions array.

2)) Algorithm for printing elements of one dimension array in reverse form.

//

***Best* , *Worst* and *Average* Cases**

We can extricate ourselves from the difficulties resulting from situations when the chosen parameters are not adequate to determine the step count uniquely by defining three kinds of step counts: best case, worst case, and average case.

1)) The best-case step count is the ***minimum*** number of steps that can be executed for the given parameters.

2)) The worst-case step count is the ***maximum*** number of steps that can be executed for the given parameters .

3)) The average step count is the ***average*** number of steps executed on instances with the given parameters.

In some instances, the above three cases are equivalent (there is one formula only) for the same value of instance characteristics, like in summation array element example (instance characteristics is n).

But in other instances, dependency on instance characteristics is not enough to determine time complexities, like in search problem about an element in an array (time complexities are effected with the position of required element in array in addition to instance characteristics, n.

Ex. 9:

Find max element in one dimension array.

```
Algorithm  arrayMax(A,n):
Input: An array A storing n integers.
Output: currentMax,the maximum element in A.
1.  currentMax← A[1]
2.  for i← 2  to  n
3.    if currentMax < A[i] then
4.     currentMax← A[i]
5.    end if
6.  end for
7.  return currentMax
```

The instance characteristics of this problem is **n**.

Time complexities (step counts):

1)) Best case:

It occurs if the first element (A[1]) is the max.

1. assignment statement …… *1*
2. for …… (n-2+1) + 1 = *n*
3. if …… *n-1*
4. replacement statement …….. ***0***
7. return …… *1*

$T^{B}_{arrayMax}(n) = 2n + 1$ It is linear function.

2)) Worst case:

It occurs when array A is sorted in increasing form, max element is A[n].

1. assignment statement …….. *1*
2. for …… *n*
3. if …… *n-1*
4. **replacement statement** …….. ***n-1***
7. return …… *1*

$T^{W}_{arrayMax}(n) = 3n$ It is linear function.

For all, the formula of worst case is greater than the formula of best case.

3)) Average case:

It is the average of complexities for all array values.

$$T^{A}_{arrayMax}(n) = \frac{\sum_{i=1}^{n}(2n+i)}{n} = \frac{5n+1}{2}$$

For all, best and worst cases are subset from the average case.