CS350 ABET Objective 4

Use the mathematical techniques required to prove the time complexity of a program/algorithm.

Textbook Section 2.3, 2.4 (2.6) ~10%.

Apply Big Oh framework to analyzing the efficiency of non-recursive and recursive algorithms.

Use summations and recurrence relations to produce close forms of the running time of a program/algorithm.

Contrast mathematical techniques with empirical analysis.

Example 1

ALGORITHM *MaxElement*($A[0..n-1]$)

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Analysis steps:

1. Choose input size.
2. Choose basic operation.
3. Dependency size-basic op
4. Function input to count basic op

Verify $\Theta(n)$

Example 2

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Input: array size

Basic op: number of comparisons

Dependency: worst case

Function: $C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

Verify $\Theta(n^2)$

Example 3

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Input: matrix order

Basic op: number of multiplications

Dependency: none

Function: $M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$

Verify $\Theta(n^3)$

Example 4

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
  $count \leftarrow count + 1$
  $n \leftarrow \lfloor n/2 \rfloor$
return *count*

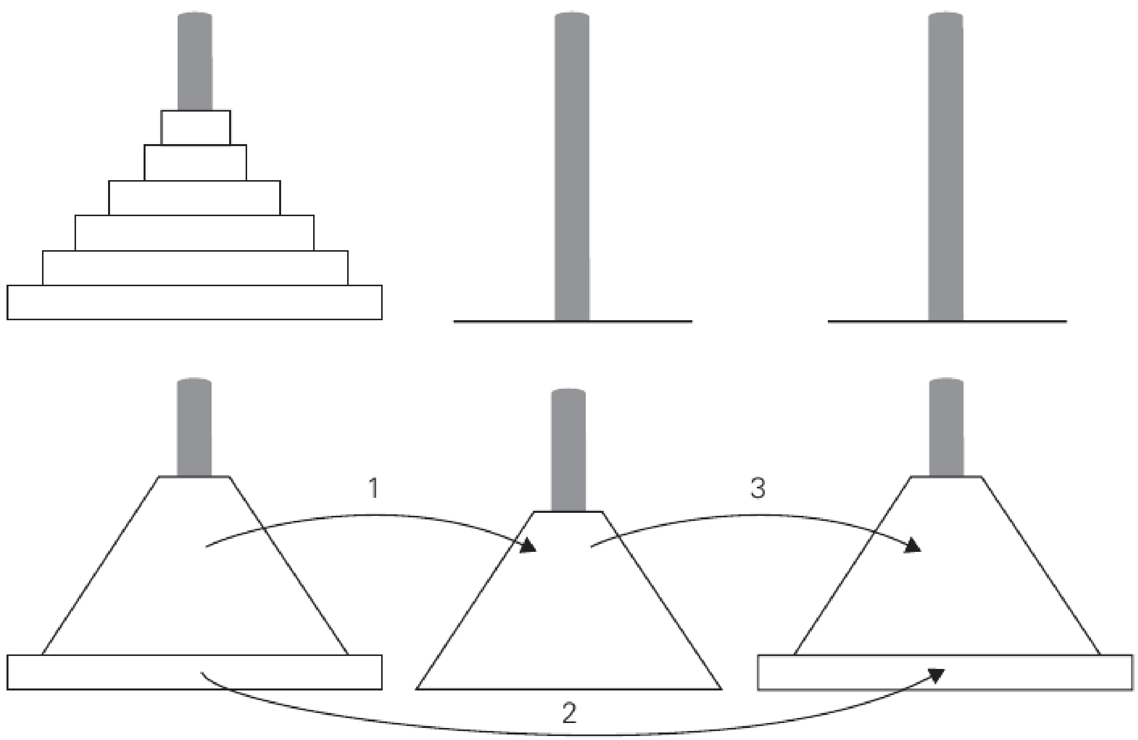
Input:	integer *n*
Basic op:	while test
Dependency:	none
Function:	$T(n) = \lfloor \log_2 n \rfloor + 1$

Verify $\Theta(\log_2 n)$

Recursive Algorithms

Are analyzed in a very similar way, but counting the number of times that the basic operation is executed involves a recurrence relation.

Example: Tower of Hanoi puzzle



Example 5

```
hanoi(0, _, _, _) = do nothing
hanoi(n, from, to, spare) =
   hanoi(n-1, from, spare, to)
   move(n, from, to)
   hanoi(n-1, spare, to, from)
```

Input: number of disks
Basic op: move a disk
Dependency: none
Recurrence:

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 0$$
$$M(0) = 0$$

Verify $M(n) = 2^n - 1$

Verification

$$M(0)=0$$
$$M(1)=2\cdot 0+1 \qquad 0=2^{1-1}-1$$
$$M(2)=2\cdot 1+1 \qquad 1=2^{2-1}-1$$
$$M(3)=2\cdot 3+1 \qquad 2=2^{3-1}-1$$
$$M(4)=2\cdot 7+1 \qquad 3=2^{2-1}-1$$
$$\ldots$$
$$M(n)=2\cdot(2^{n-1}-1)+1$$

The last line is a guess (inductive inference) provable by induction. Simplification gives:

$$M(n)=2^{n}-1$$

Example 6

ALGORITHM $F(n)$

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) ∗ n
```

Count multiplications as a function of *n*.

$$
\begin{aligned}
&M(0)=0\\
&M(1)=M(0)+1=1\\
&M(2)=M(1)+1=2\\
&M(3)=M(2)+1=3\\
&\ldots\\
&M(n)=M(n-1)+1=n
\end{aligned}
$$

Empirical Analysis

Analysis steps:

1. State experiment purpose.
2. Set measure (count vs time).
3. Generate input.
4. Code program.
5. Record execution data.
6. Analyze data.

Example 7.1

Empirical analysis of Hanoi puzzle.

Steps:

1. State experiment purpose: verify that the program of Example 5, for an instance with *n* disks, makes exactly $2^n - 1$ moves.
2. Set measure: obviously “count moves”.
3. Generate input: each instance with *n* from 1 to 10.
4. Code program: see program below.
5. Record execution data: for each instance *n*, print both counted number of moves and $2^n - 1$.
6. Analyze data: compare two values for equality.

Example 7.2

Ruby code of the instrumented Hanoi puzzle:

```
def count_moves(disks, from, to, spare)
  if disks>0 then
    return count_moves(disks-1, from, spare, to) +
           1 + count_moves(disks-1, spare, to, from)
  else
    return 0
  end
end

def instance(n)
  return count_moves(n,"A","B","C")
end

(1..10).each { |x|
  printf("Hanoi %2d is %4d (%d)\n",
      x, instance(x), 2**x-1)
}
```

Example 7.3

Output of instrumented Hanoi program:

```
Hanoi  1 is    1 (1)
Hanoi  2 is    3 (3)
Hanoi  3 is    7 (7)
Hanoi  4 is   15 (15)
Hanoi  5 is   31 (31)
Hanoi  6 is   63 (63)
Hanoi  7 is  127 (127)
Hanoi  8 is  255 (255)
Hanoi  9 is  511 (511)
Hanoi 10 is 1023 (1023)
```

Since the last two columns are equal, the experimental analysis confirms the theory.

References

Textbook Section 2.3, 2.4, 2.6

Web:

http://en.wikipedia.org/wiki/Tower_of_Hanoi
http://mathworld.wolfram.com/TowerofHanoi.html