Design and Analysis of Algorithms

L07:
Recursive and Non-Recursive Algo

Dr. Ram P Rustagi
Sem IV (2019-H1)
Dept of CSE, KSIT
rprustagi@ksit.edu.in

Resources

- T1: Levitin
- T2: Horowitz

Efficiency of Non-Recursive Algos

- Generic plan for non-recursive algorithms
  - Decide on parameter n indicating *input size*
  - Identify algorithm's *basic operation*
    - The operation that is most executed
  - Determine *worst*, *average*, and *best* cases for input of size n
    - Does input data affects the performance of algo
  - Set up a sum for the number of times the basic operation is executed
  - Simplify the sum using standard formulas and rules

Ex 01: Finding Maximum Element

- **Prog:** `FindMax(A[1..n])`

```
// Input: array A
// Output: The value of largest element
max ← A[1]
for i = 2 to n, do
 if A[i] > max, then
  max ← A[i]
 fi
end // for
return max
```

- **Efficiency:** $\Theta(n)$, $O(n)$
  - The operation `A[i]>max` is executed `n−1` times
    - $\Sigma_{2 \le i \le n} 1 = n-1 = \Theta(n)$

Ex02: Uniqueness problem

- Verify if input array `A[1..n]` has all unique elements
- Output: `True` if all elements in array are unique, `False` otherwise.
- Algo

```
for i =1 to n-1, do
  for j = i+1 to n, do
    if A[i] == A[j], then
      return False
return True
```

- Efficiency: basic operation `A[i] == A[j]`

$T(n)=\Sigma_{1\le i\le n-1}\ \Sigma_{i+1\le j\le n}\ 1)$

$=(n-1)+(n-2)+...+2+1 = (n-1)n/2$

$=\Theta(n^2)$ comparisons

Ex03: Matrix Multiplication

- Multiply two `nxn` matrices `A` and `B`
- Output: Matrix `C=AB`.
- Algo

```
for i=1 to n, do
  for j=i to n, do
    C[i,j]=0
    for k=i to n, do
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
return C
```

- Efficiency: basic operation `C[i,j]+A[i,k]*B[k,j]`

$$T(n)=\Sigma_{1\le i\le n}\ \Sigma_{1\le j\le n}\ \Sigma_{1\le k\le n}\ 2)$$

$$= 2n^3$$

$$= \Theta(n^3)$$

Ex04: Binary Digits in a Number

- Find the number of binary digits in a +ve decimal integer
- Input: a positive decimal integer n
- Output: number of binary digits
- Algo : //we can't use for loop anymore

```
count ← 0
while n>1, do
  count++
  n ← ⌊n/2⌋
return count
```

- Efficiency (basic operations): comparison `n>1`

  division: $n \leftarrow \lfloor n/2 \rfloor$

  = Each iteration, number is halved. total iterations $\log_2 n$

  $T_n = \Theta(\log n)$

Efficiency of Recursive Algos

- Generic plan for recursive algorithms
  - Decide on parameter n indicating *input size*
  - Identify algorithm's *basic operation*
    - The operation that is most executed
  - Determine *worst*, *average*, and *best* cases for input of size n
    - Does input data affects the performance of algo
    - Investigate the three cases separately
  - Set up a recurrence relation
    - How many times the number basic op. is executed.
  - Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method

Ex05: Computation of Factorial `n`

- General Defintion `n! = n*(n-1)*…*2*1`
- Recursive definition `F(n) = n * F(n-1)`
  – Recursion exit on `n=1`
- Algorithm F(n)

  if `n` equals `0` or `n` equal `1`, then
    return `1`
  else
    return `n*F(n-1)`

- Efficiency: Basic operation : multiplication
  - Number of recursion invocations : `n`

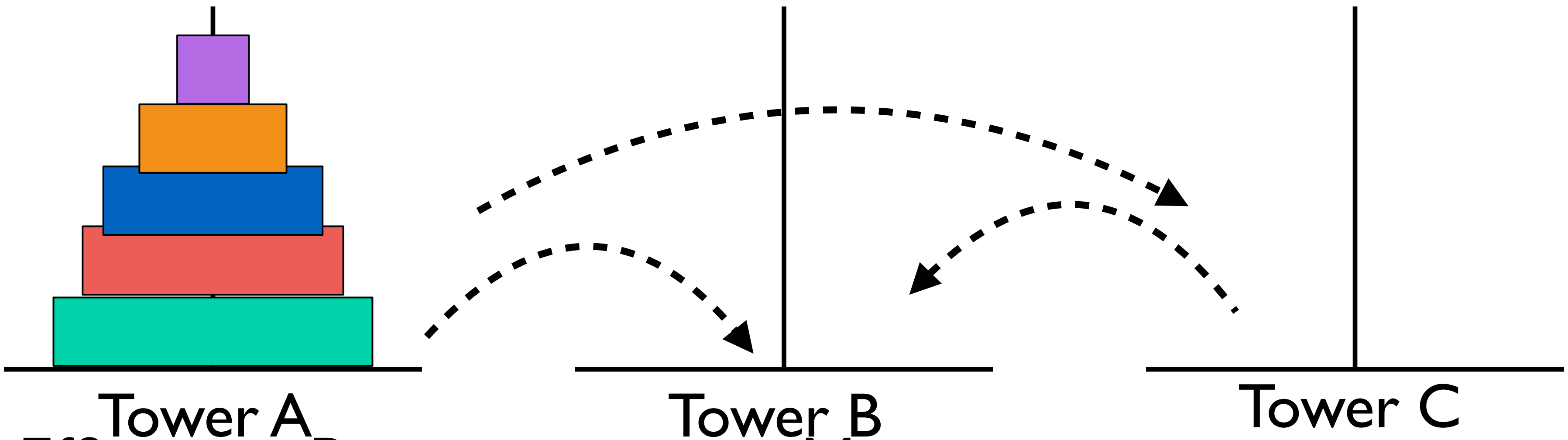
$T(n)=1+T(n-1)$

$= 1+1+T(n-2) = n$

$T(n)= \Theta(n)$

Ex 06: Tower of Hanoi

- Task: Tranfer n discs from tower A to tower B using tower C while following the rule of discs placement



- Efficiency: Basic operations: Move (n-1),1,(n-1)

$$
\begin{aligned}
T(n) &= T(n-1)+1+T(n-1) \\
&= 1+2*T(n-1) \\
&= 1+2(1+2*T(n-2)) \\
&= 2^0+2^1+2^2+\ldots+2^{n-1} = 2^n - 1 \\
&= \Theta(2^n)
\end{aligned}
$$

Ex07: Binary Digits in a Number

- Find the number of binary digits in a +ve decimal integer
- Input: a positive decimal integer n
- Output: number of binary digits
- Algo : `BinDigits(n)`

  if `n` equals `1`

  return `1`

  else

  return `1 + BinDigits(`$\lfloor n/2 \rfloor$`)`
- Efficiency: Basic operations: Halving the value

$$
\begin{aligned}
T(n) &= 1 + T(\lfloor n/2 \rfloor) \\
&= 1 + 1 + T(\lfloor n/2^2 \rfloor) \\
&= 1 + 1 + \ldots + 1 \quad (\log_2 n \text{ times}) \\
&= \log_2 n \\
&= \Theta(\log_2 n)
\end{aligned}
$$

Solving Recursion Relations

- Method of forward substitution
  `T(n) = aT(n-1) + 1`
- Method of backward substitution
  `T(n) = T(n-1) + n`
- Decrease by `1`
  `T(n) = T(n-1) + f(n)`
- Decrease by a constant factor
  `T(n) = T(n/b) + f(n)`
- Divide and conquer
  `T(n) = aT(n/b) + f(n)`
- …

Method of Forward Substitution

$T(n) = aT(n-1) + 1$, **and** $T(0) = 1$

$= a(aT(n-2) + 1) + 1$

$= a^2T(n-2) + 1 + 1$

$= a^2(aT(n-3) + 1) + 1 + 1$

$= a^3T(n-3) + 1 + 1 + 1$

:

:

$= a^nT(0) + 1 + 1 + \ldots + 1$ (n times)

$= a^n + n$

$= \Theta(a^n)$

Method of Backward Substitution

$T(n)=T(n-1)+n$, **and** $T(0)=1$

$=T(n-2)+(n-1)+n$

$=T(n-3)+(n-2)+(n-1)+n$

$=T(0)+1+2+\dots+n$

:

:

$=n(n+1)/2$

$=\Theta(n^2)$

Decrease by 1

$T(n) = T(n-1) + f(n)$, **and** $T(0) = 1$

$= T(n-2) + f(n-1) + f(n)$

$= T(n-3) + f(n-2) + f(n-1) + f(n)$

:

:

$= T(0) + \Sigma_{1 \le i \le n} f(i)$

- Growth dependes upon how $f(n)$ behaves.
  - For $f(n)=1,\ T(n)=n$
  - For $f(n)=\log_2 n,\ T(n)=n\log_2 n$
  - For $f(n)=n,\ T(n)=n(n+1)2 = \Theta(n^{2)}$
  - For $f(n)=n^k,\ T(n)= \Theta(n^{k+1})$

Decrease by Constant Factor

$T(n) = T(n/b) + f(n)$, **and** $T(0) = 1$

$= T(n/b^2) + f(n/b) + f(n)$

$= T(n/b^3) + f(n/b^2) + f(n/b) + f(n)$

:

:

$= T(1) + \Sigma_{1 \le i \le k} f(i)$, **where** $n = b^k$

- Growth dependes upon how $f(n)$ behaves.
  - For $f(n)=1,\ k=\log_b n,\ T(n)=\log_b n$
  - For $f(n)=n, k=\log_b n;\ T(n)=\Sigma_{1 \le i \le k} f(b^i)$

    $T(n)=\Sigma_{1 \le i \le k} b^i = (b^k-1)/(b-1) = \Theta(b^k)$

    $= \Theta(n)$

Summary

- Analysis of Non Recursive algorithms
- Analysis of recursive algorithms
- Recurrence relation examples