Analysis of Algorithm: Time & Space Complexity

Sudheer K Sharma

Agenda of the presentation

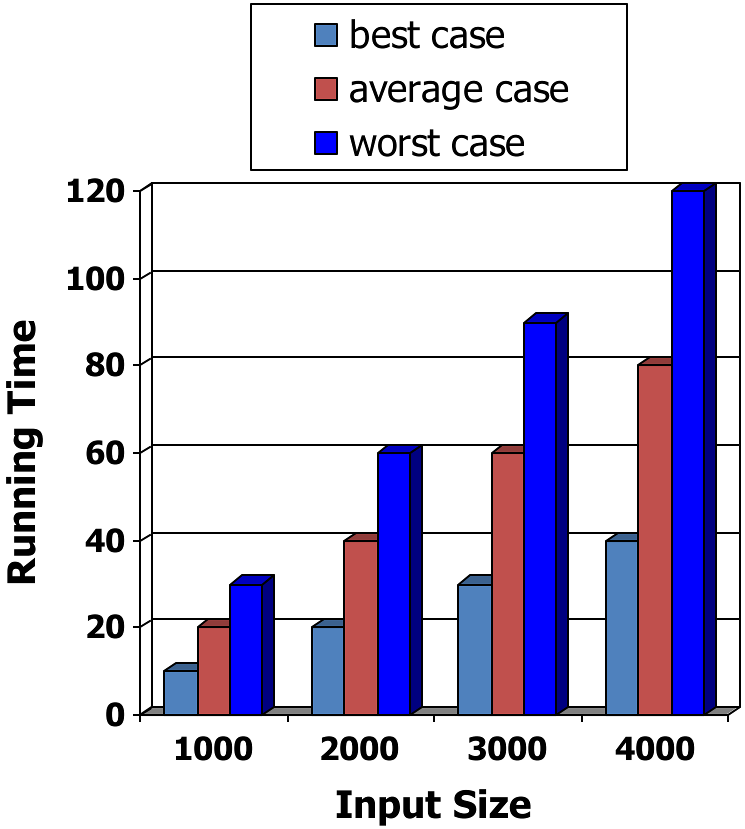
1. Introduction of Algorithm.
2. Theoretical Analysis.
3. Asymptotic Notations.
4. Types of Algorithm
   i. Iterative.
   ii. Recursive.
5. Time Complexity of Iterative Algorithms.
6. Time Complexity of Recursive Algorithms.
7. Space Complexity of Recursive Algorithms.

What is an Algorithm?

- An algorithm is any well defined computational procedure that takes some value or set of values as input and produce some value or set of values as output [Cormen at. Al.]
- An algorithm is thus a sequence of computational steps that transform the input into the output.

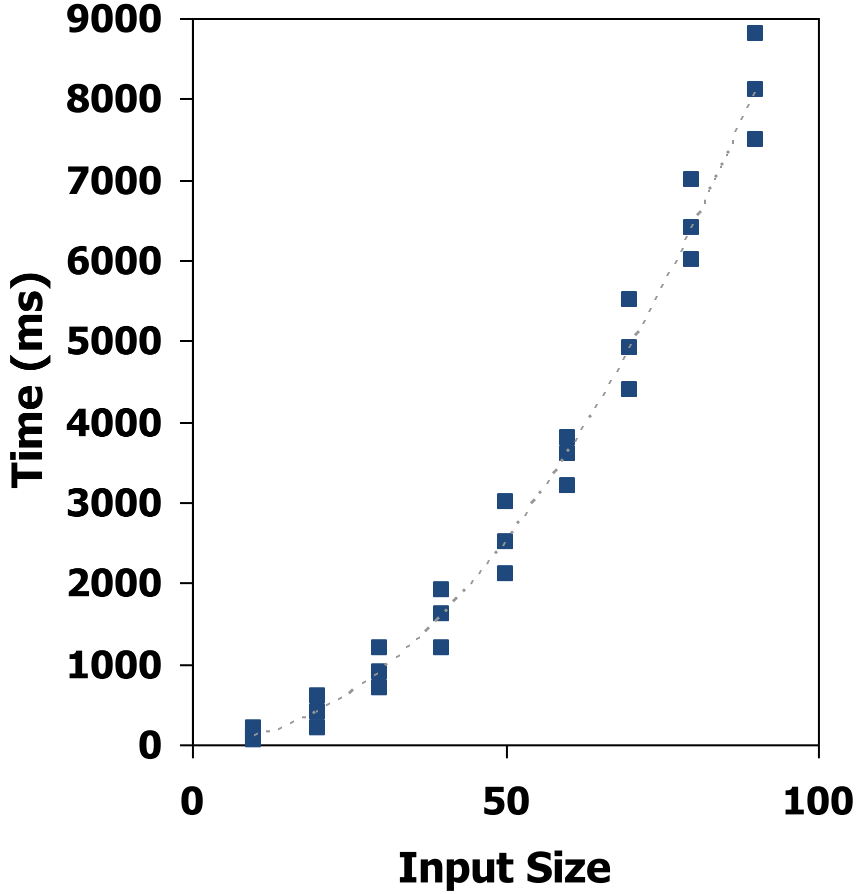
Running Time of an Algorithm

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult

- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
    Input array ***A*** of ***n*** integers
    Output maximum element of ***A***

    currentMax $\leftarrow$ ***A***[0]
    for ***i*** $\leftarrow$ 1 **to** ***n*** $-$ 1 **do**
        if ***A***[*i*] > ***currentMax*** **then**
            currentMax $\leftarrow$ ***A***[***i***]
    return ***currentMax***

Asymptotic Notations

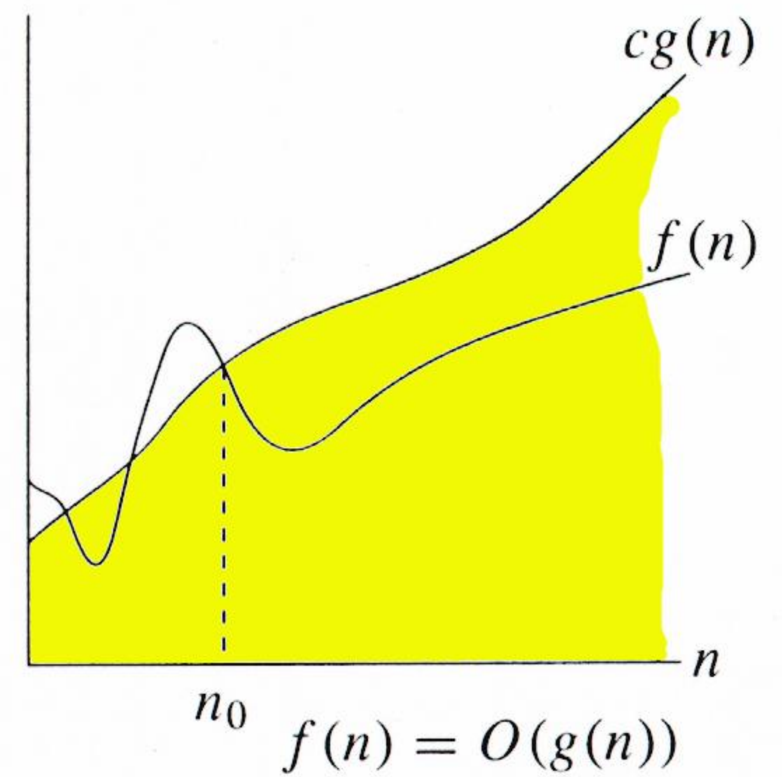
- **Θ, *O*, Ω, *o*, ω**
- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n)$:
$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ $\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.



$g(n)$ is an *asymptotic upper bound* for $f(n)$.

Examples

> ***O*(*g*(*n*)) = {*f*(*n*) : ∃ positive constants *c* and n_0, such that ∀*n* ≥ n_0, we have 0 ≤ *f*(*n*) ≤ c*g*(*n*) }**

- Any linear *function* $an + b$ is in $O(n^2)$. **How?**
- Show that $3n^3 = O(n^4)$ for appropriate c and n_0.

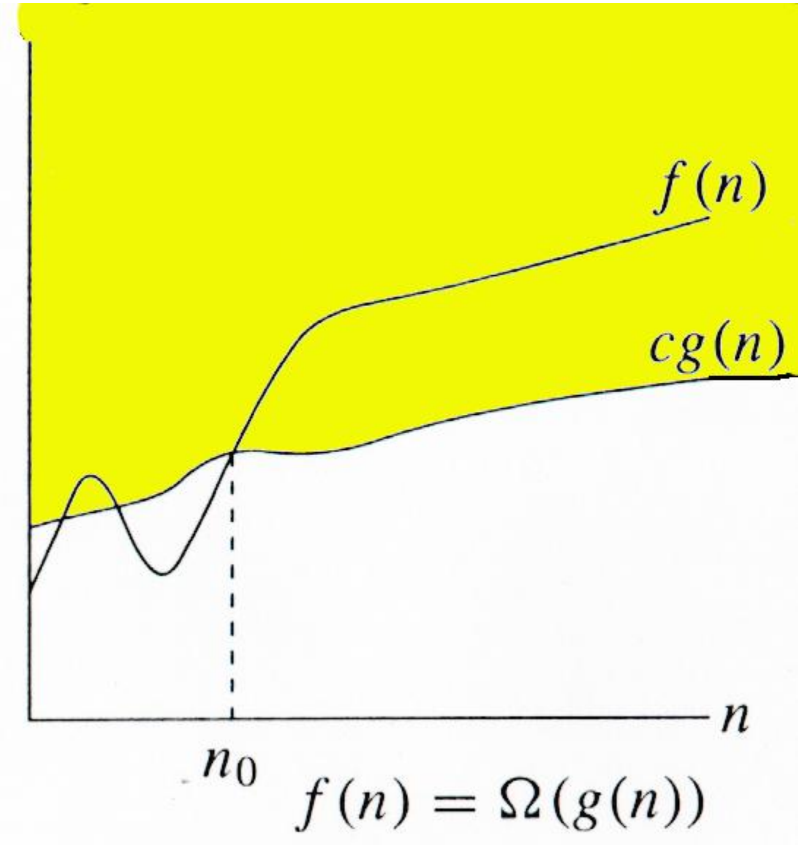
Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n))$ = {$f(n)$:
$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,

we have $0 \leq cg(n) \leq f(n)$}

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.



***g*(*n*) is an *asymptotic lower bound* for *f*(*n*).**

Example

$\Omega(g(n)) = \{f(n) :$ $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

- $\sqrt{n} = \Omega(\lg n)$. Choose c and n_0.

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

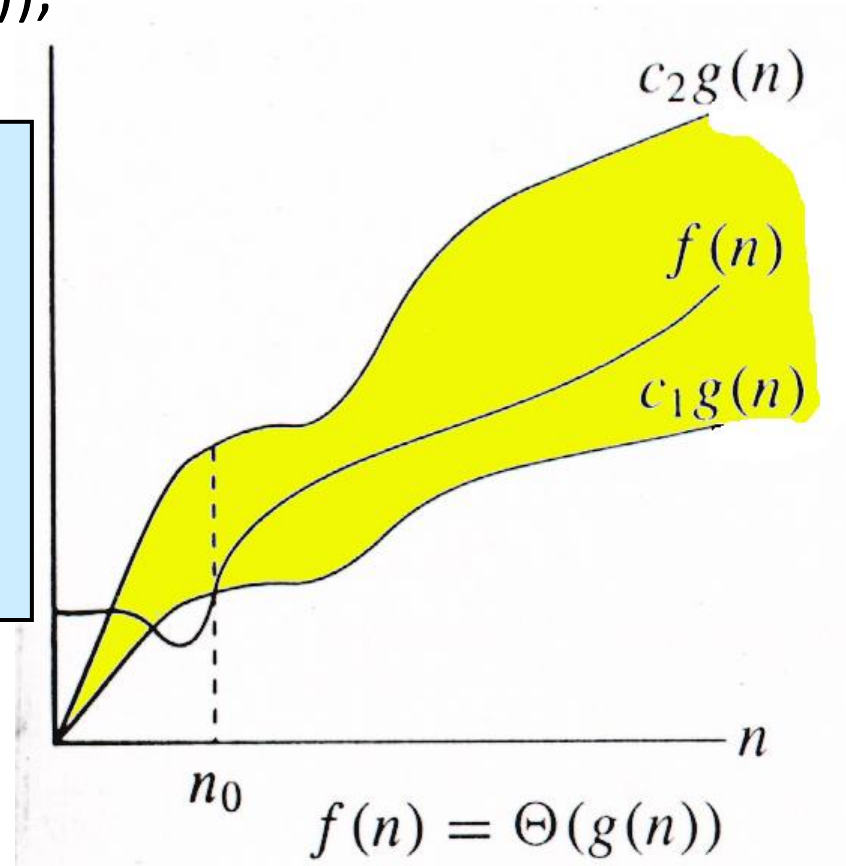
$\Theta(g(n))$ = **{$f(n)$:**

$\exists$ positive constants c_1, c_2, and n_0,

such that $\forall n \geq n_0$,

we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

}



Technically, $f(n) \in \Theta(g(n))$.
Older usage, $f(n) = \Theta(g(n))$.
I'll accept either…

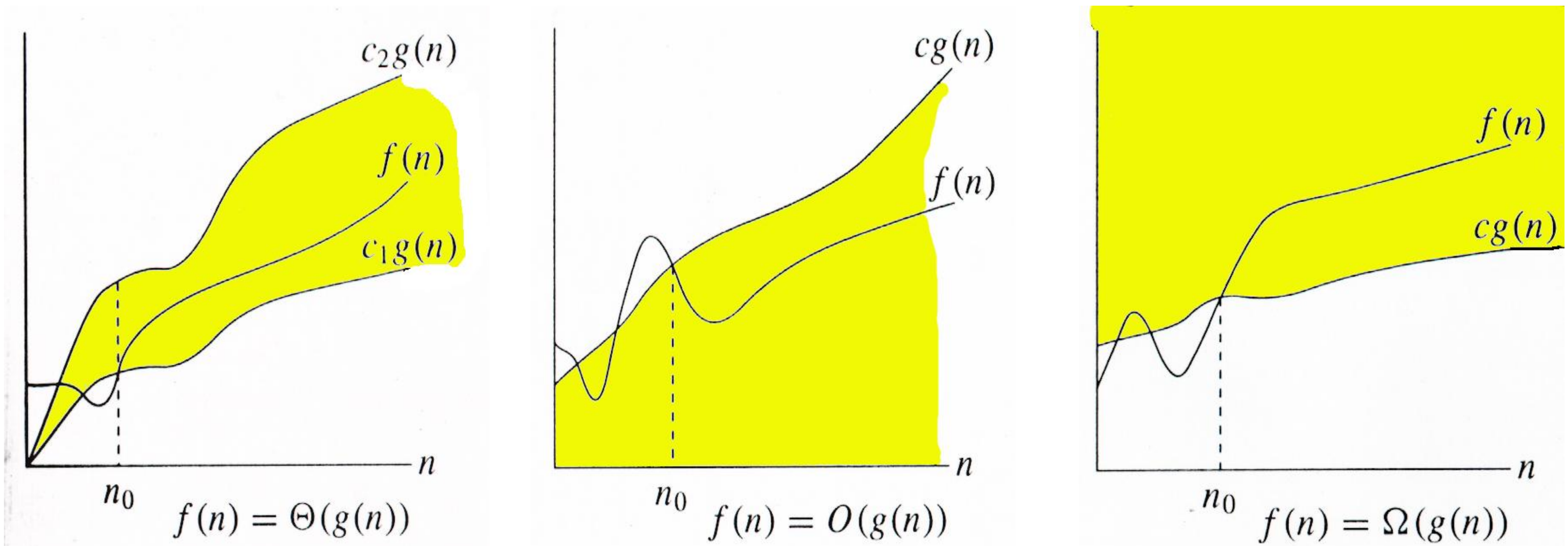
$f(n)$ and $g(n)$ are nonnegative, for large n.

Example

$\Theta(g(n))$ = $\{f(n)$: $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- $10n^2 - 3n = \Theta(n^2)$
- What constants for n_0, c_1, and c_2 will work?
- Make c_1 a little smaller than the leading coefficient, and c_2 a little bigger.
- ***To compare orders of growth, look at the leading term.***
- Exercise: Prove that $n^2/2-3n= \Theta(n^2)$

Relations Between Θ, *O*, Ω



Types of Algorithm

- There are two key types of algorithms
  - Iterative Algorithms:

    An **iterative algorithm** executes steps in iterations. It aims to find successive approximation in sequence to reach a solution.

  - Recursive Algorithms:

    A **recursive algorithm** is an **algorithm** which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.

Example: Factorial of a number, Iterative method.

```
factorial(n) {  i = 1
      factN = 1
      loop (i <= n)
            factN = factN * i
            i = i + 1
      end loop
      return factN  }
```

Example: Factorial of a number, Recursive method.

```
factorial(n) {
      if (n = 0)
            return 1
      else
            return n*factorial(n-1)
      end if
}
```

Time Complexity of Iterative Algorithms

Example: Factorial of a number, Iterative method.

```
factorial(n)                      ← Size of problem is n
{  i = 1                          ← # steps: One
   factN = 1                      ← # steps: One
   loop (i <= n)                  ← # steps: n
      factN = factN * i           ← # steps: n
      i = i + 1                   ← # steps: n
   end loop
   return factN                   ← # steps: One
}
```

Total # steps = 3n+3

If unit step is performed in c unit time,

Total running time= (3n+3)*c,

Hence the time complexity,

T(n)=O(cn)=O(n)

Time Complexity of Recursive Algorithms

- Example: Factorial of a number, Recursive method.

```
factorial(n) {
    if (n = 0)
        return 1
    else
        return n*factorial(n-1)
    end if
}
```

- Time complexity

- In order to find out the time complexity of Recursive algorithm first we need to obtain time complexity recursive function.
- In this case the recursive function is $T(n)=2+T(n-1)$

Time Complexity of Recursive Algorithms

- There are three methods to solve recursive function in order to find time complexity of the Algorithm
  - Back Substitution Method
  - Recursion tree Method
  - Masters Method

Substitution Method

- In this method we substitute the previous term in the recursive equation like we have in the case of factorial

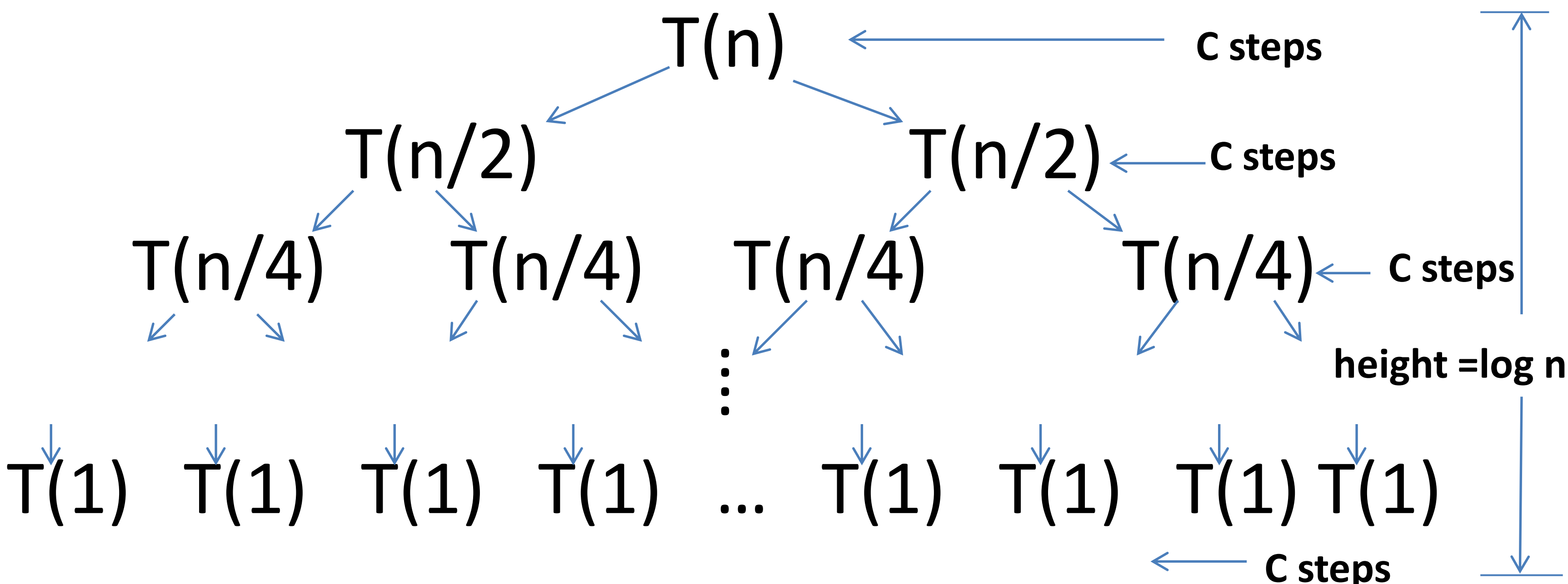
$$
\begin{aligned}
T(n) &= 2+T(n-1) \\
&= 2+2+T(n-2) \\
&= 2+2+2+T(n-1) \\
&\vdots \\
&= 2*n + T(0)
\end{aligned}
$$

$$\Rightarrow T(n)=O(2*n)=O(n)$$

Recursive tree Method

- If the recursive function is of the form

  T(n)=2T(n/2)+c, in this case we use recursive tree method to solve the recursion



Complexity T(n)=height*#steps at each level=C*log n

=> T(n)=O(C*log n)=O(log n)

Masters Method

- Masters method are frequently used to solve recursions.

$$T(n) = aT\left(n/b\right) + \theta\left(n^k \log^p n\right)$$

$$where \;\; \log^p n = \overbrace{\log\log\ldots\log}^{p\ times}(n)$$

- Master theorem works on three rules

1. $If \;\; a > b^k,\; then\; T(n) = \theta\left(n \log_b a\right)$

2. $If \;\; a = b^k$

   (i) If p>-1, $then \;\; T(n) = \theta\left(n^{\log_b a} \log^{p+1} n\right)$

   (ii) If p=-1, $then \;\; T(n) = \theta\left(n^{\log_b a} \log\log n\right)$

   (iii) If p<-1, $then \;\; T(n) = \theta\left(n^{\log_b a}\right)$

3. $If \;\; a < b^k$

   (i) If $p \geq 0$, $then \;\; T(n) = \theta\left(n^k \log^p n\right)$

   (ii) If p<0, $then \;\; T(n) = \theta\left(n^k\right)$

Examples

I. $T(n)=3T(n/2)+n^2$

here $a=3$, $b=2$, $k=2$ and $p=0$ & $a<b^k$ so rule 3(i) will work

$\Rightarrow T(n)=\theta(n^2\log^0 n)=\theta(n^2)$

II. $T(n)=4T(n/2)+n^2$

Here $a=4$, $b=2$, $k=2$, $p=0$ and $a=b^k$ So rule 2(i) will work

$$\Rightarrow T(n)=\theta\left(n^{\log_2 4}\log^1 n\right)=\theta\left(n^2\log n\right)$$

Space Complexity of Recursive function

- Example if we have a recursive algorithm
- A(n) {

```
      if(n≥1)
   {
      A(n-1);
     Pf(n);
      } }
```

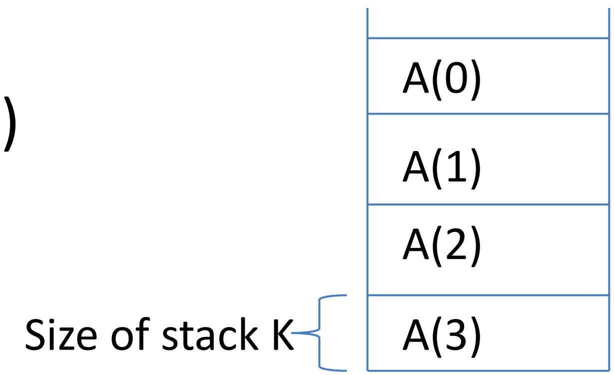
Let n=3 so recursive tree is

A(3)

A(2) Pf(3)

A(1) Pf(2)

A(0) Pf(1)



Output is

123

So when input is 3 there were 4 recursive call therefore when input is n there should be n+1 recursive calls.

Each stack is of size k.

Total space required is O((n+1)k).

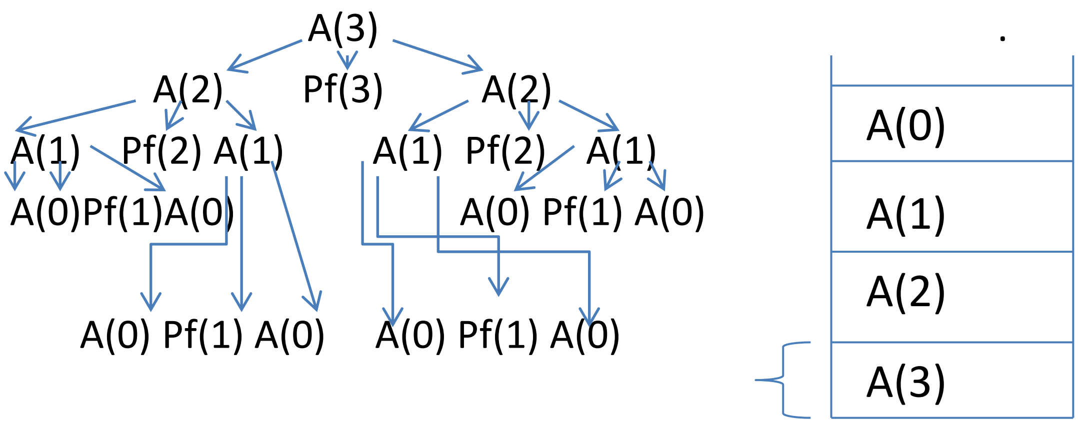
Hence the space complexity is O(nk)=O(n).

Space Complexity of Recursive function

- Example if we have a recursive algorithm
- A(n) {

```
    if(n≥1)
  {
    A(n-1);
   Pf(n);
   A(n-1);
    } }
```

Let n=3 so recursive tree is



Size of stack K

Output is 1213121

When input is 3 there were 15=2^{3+1}-1 recursive call, for n=2 there were 7=2^{2+1}-1 recursive call, and for n=1 there were 3=2^{1+1}-1 recursive call, therefore when input is n there should be 2^{n+1}-1 recursive calls.

Every time when the function is called it will push into the stack and when its job is done it will pop from the stack so here we required only stack with level n+1 only

Each stack is of size k. Total space required is O((n+1)k). Hence the space complexity is O(nk)=O(n).

Thank You