Design and analysis of Algorithms

CSC 314

BScCSIT 5th

Instructor:

Shiv Raj Pant

About the Course

- The course introduces basic concepts of Algorithm design and analysis techniques
- Basic knowledge of Algorithms and Data structures is required

Class plan:

- The course is rather lengthy.
- We have lost about two months already. That is a significant amount of time !
- To compensate for the lost time we need to fully utilize the remaining time.
- There will be no Saturdays or other public holidays !
- The classes will run everyday. Although, class time can be altered for holidays.
- The effectiveness of the class will be determined by student’s active participation and involvement.
-

Text book:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *"Introduction to algorithms"*.

Reference book:

2. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekiaran, *"Computer Algorithms"*

   (some topics will be covered from this book)

Supplementary materials:

1. Instructor slides
2. Old notes from Samujwal Bhandari (will be provided by instructor)
   - These notes are very old and do not fully match with your syllabus. Nevertheless, I will copy most of my slide content from these notes.
3. Syllabus and old question papers available online at : https://collegenote.pythonanywhere.com/ (These online materials are not from the instructor(me). The instructor is not responsible for the content of the website. Please use the information (at your own risk) if you find it useful

Syllabus:

Unit 1: Foundation of Algorithm Analysis

Unit 2: Iterative Algorithms

Unit 3: Divide and Conquer Algorithms

Unit 4: Greedy Algorithms

Unit 5: Dynamic Programming

Unit 6: Backtracking algorithms

Unit 7: Number Theoretic Algorithms

Unit 8: NP Completeness

Unit 1: Foundation of Algorithm Analysis

1.1. Algorithm and its properties, RAM model, Time and Space Complexity

1.2. Asymptotic Notations: Big-O, Big-Ω and Big-θ Notations their Geometrical Interpretation and Examples.

1.3. Recurrences: Recursive Algorithms and Recurrence Relations, Solving Recurrences (Recursion Tree Method, Substitution Method, Application of Masters Theorem)

Algorithms

- An algorithm is a finite set of computational instructions, each instruction can be executed in finite time, to perform computation or problem solving by giving some value, or set of values as input to produce some value, or set of values as output.
- Algorithms are not dependent on a particular machine, programming language or compilers.

Algorithms Properties

- *Input(s)/output(s)*: There must be some inputs from the standard set of inputs and an algorithm's execution must produce outputs(s).
- *Definiteness*: Each step must be clear and unambiguous.
- *Finiteness*: Algorithms must terminate after finite time or steps.
- *Correctness*: Correct set of output values must be produced from the each set of inputs.
- *Effectiveness*: Each step must be carried out in finite time.

Expressing Algorithms

- There are many ways of expressing algorithms.
- e.g., Natural language, pseudo code and real programming language syntax.
- The natural language is most easy to express and understand.
- Pseudo code is also easy to understand. It combines naturalness of natural language and eliminates the compiler-dependency of real programming language syntax.
- Real programming language syntax is rarely used in expressing algorithms as the algorithms should not depend on machine or compiler.

Analyzing algorithms

- We need algorithms to understand the basic concepts of the Computer Science and programming, where the computations are done
- To understand the input output relation of the problem, we must be able to understand the steps involved in getting output(s) from the given input(s).
- The analysis of the algorithms gives a good insight of the algorithms under study.
- Analysis of algorithms tries to answer few questions like:
  - *is the algorithm correct? i.e. the algorithm generates the required result or not?*
  - *Does the algorithm terminate for all the inputs under problem domain?.*
  - *Is the algorithm efficient?*
  - *Does the algorithm give optimal solution? etc.*

- So knowing the different aspects of different algorithms on the similar problem domain, we can choose the better algorithm for our need.
- This can be done by knowing the *resources needed* for the algorithm for its execution.
- *Two most important resources are the time and the space*.

Time complexity and space complexity

- *Time complexity*
  - It is the measure of how long an algorithm would take to solve a problem
  - We can not measure running time of an algorithm in terms of real (watch) time unit i.e. hour/minute/second.(*why ?*)
  - The running time is measured in terms of "How many steps it would take to solve a problem (input)" (counted as the number of instructions executed).
  - The running time is a function of problem(input) size.
  - The time complexity shows how the number of steps increase as the input size increases.

    e.g The complexity of Bubble sort is $\Theta(n^2)$. It means the number of steps taken to sort given set of elements increases quadratically.

  - Space complexity is the measure of how much memory will be required by an algorithm during execution (problem solving).
  - Again, we can not measure required memory in terms of real units, i.e., bytes
  - The space complexity is measured in terms of "how many units of memory are required for given problems size" (usually counted as the number of variables or objects required to be stored in memeory).

Best, Worst and Average case

- *Best case complexity* gives lower bound on the running time of the algorithm for any instance of input(s). This indicates that the algorithm can never have lower running time than best case for particular class of problems.

- *Worst case complexity* gives upper bound on the running time of the algorithm for all the instances of the input(s). This insures that no input can overcome the running time limit posed by worst case complexity.

- *Average case complexity* gives average number of steps required on any instance(s) of the input(s).

Random Access Machine (RAM) Model

- RAM model of computation is the generic one-processor model used to analyze the complexity of algorithms.
- In RAM model, we assume the instructions are executed one after another (no concurrent execution).
- The RAM (Random Access Machine) model of computation measures the run time of an algorithm by summing up the number of steps needed to execute the algorithm on a set of data.
- The RAM model operates by the following principles:
  I. Basic logical or arithmetic operations (+, -, *, =, if, call) are considered to be simple operations that take one time step.
  II. Loops and subroutines are complex operations composed of multiple time steps.
  III. All memory access takes exactly one time step.
- In this way, we measure run time of algorithm by counting the steps.
- This model encapsulates the core functionality of computers but does not mimic them completely. For example, an addition operation and a multiplication operation are both worth a single time step, however, in reality it will take a machine more operations to compute a product versus a sum.

The *Basic steps* in the RAM model(the instructions that are assumed basic)

- load/store: assignment, use of a variable
- arithmetic operations: addition, multiplication, division, etc.
- branch operations: conditional branch, jump
- subroutine call

A basic step in the RAM model takes a constant time

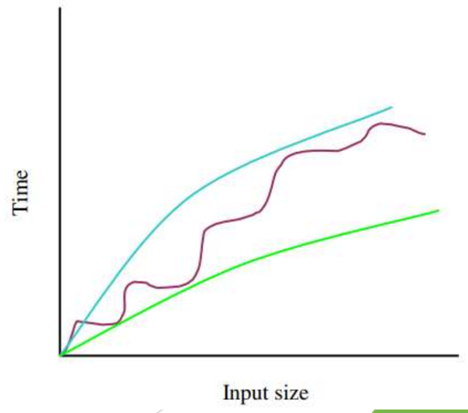
Example

```
//Input: n, a positive integer
//Output: Factorial of n.
factorial(n)
{
  fact = 1
  i = 1
  while(i<=n) do
     fact = fact * i
     i = i + 1
  end while
  return fact
}
```

Can you analyze the complexity of above algorithm using RAM model ?

Asymptotic Notations

- Complexity analysis of an algorithm is very hard if we try to analyze exact.
- We know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input.
- So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier.
- For this purpose we need the concept of asymptotic notations.
- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value
- i.e. it gives an idea about how the running time of the algorithm increases with input size.
- Three types of asymptotic notations:
  - *Big-oh* (O) *notation*
  - *Big-omega* (Ω) *notation*
  - *Big-theta* (Θ) *notation*



Big-Oh (O) notation

- Big-oh notation gives asymptotic upper bound.

  A function f(x) is said to be big-oh of another function g(x), written as f(x)=O(g(x)), iff there exist two positive constants c and x_0 such that for all x >= x_0,

  $$0 \le f(x) \le c*g(x)$$

- The above relation says that g(x) is an upper bound of f(x)

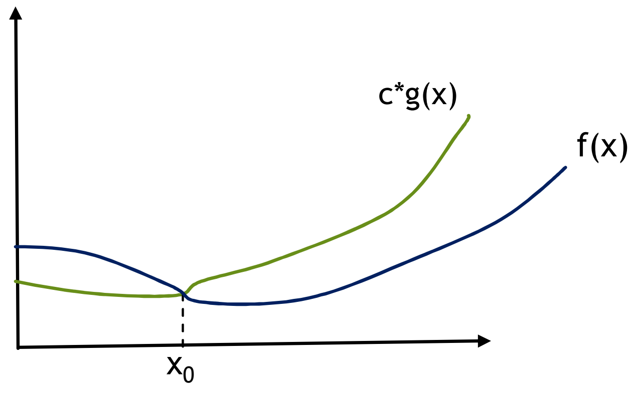


Fig: Geometrical interpretation of f(x)=O(g(x))

What f(x) = O(g(x)) means?

The growth rate of g(x) is greater than the growth rate of f(x).
i.e. when we increase the value x, the corresponding increase in function value is greater in g than in f.

some properties of O-notation:

- Transitivity : if f(x) = O(g(x)) and g(x) = O(h(x)) then f(x) = O(h(x))
- Reflexivity: f(x) = O(f(x))
- f(x) = O(g(x)) does not imply g(x) = O(f(x))
- O(f(x)) + O(g(x)) = O(max(f(x),g(x)))
- O(f(x)) * O(g(x)) = O(f(x)*g(x))
- O(1) is used to denote constants.

Did you know?

If f(n) = O(n) then f(x) = O(n^2)
= O(^{3})
= O(n^4)
.. and so on

Example of what O-notation actually means.*(this is just for understanding the concept)*

- Consider the functions n , $\log_2 n$, n^2, 2^n
- Let us compare the growth rate of these functions:

n	$\log_2 n$	n^2	2^n

Example of what O-notation actually means.*(this is just for understanding the concept)*

- Consider the functions n , $\log_2 n$, n^2, 2^n
- Let us compare the growth rate of these functions:

n	$\log_2 n$	n^2	2^n
1	0	1	2

Example of what O-notation actually means.*(this is just for understanding the concept)*

- Consider the functions n , $\log_2 n$, n^2, 2^n
- Let us compare the growth rate of these functions:

n	$\log_2 n$	n^2	2^n
1	0	1	2
2	1	4	4

Example of what O-notation actually means.*(this is just for understanding the concept)*

- Consider the functions n , $\log_2 n$, n^2, 2^n
- Let us compare the growth rate of these functions:

n	$\log_2 n$	n^2	2^n
1	0	1	2
2	1	4	4
4	2	16	16

Example of what O-notation actually means.*(this is just for understanding the concept)*

- Consider the functions n , $\log_2 n$, n^2, 2^n
- Let us compare the growth rate of these functions:

n	$\log_2 n$	n^2	2^n
1	0	1	2
2	1	4	4
4	2	16	16
8	3	64	256

Observe the growth rate from here

Example of what O-notation actually means.*(this is just for understanding the concept)*

- Consider the functions n , $\log_2 n$, n^2, 2^n
- Let us compare the growth rate of these functions:

n	$\log_2 n$	n^2	2^n
1	0	1	2
2	1	4	4
4	2	16	16
8	3	64	256
16	4	256	65536

- We see that 2^n grows too fast compared to other functions.
- ***Recall the complexity (number of steps required) in TOH problem !***

Example:

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$

Show that $f(n) = O(g(n))$.

Sol^n:

To prove above relation, we need to find two positive constants c and n_0 such that $f(n) \le c*g(n)$, for all $n \ge n_0$

let us choose $c=14$ and $n_0 = 1$

Then,

$3n^2 + 4n + 7 \le 14*n^2$, for all $n \ge 1$

Proved.

Some well-known facts about O-notation

- For a polynomial function, $f(x) = a_0 + a_1x + \ldots\ldots + a_{n-1}x^{m-1} + a_mx^m$,
  $f(x) = O(x^m)$, where m is the highest power.

- If $f(x) = O(x)$ then $f(x) = O(x^2)$
  $= O(x^3)$
  $= O(x^4)$
  .. and so on

Note: In design and analysis of algorithm, we generally use variable n because the complexity is the function of number of steps (or number of inputs) in an algorithm.

Big-Omega (Ω) notation

- Big-omega notation gives asymptotic lower bound.

  A function f(x) is said to be big-omega of another function g(x), written as f(x)= Ω(g(x)), iff there exist two positive constants c and x_0 such that for all x >= x_0,

  $$0 \leq c*g(x) \leq f(x)$$

- The above relation says that g(x) is a lower bound of f(x)

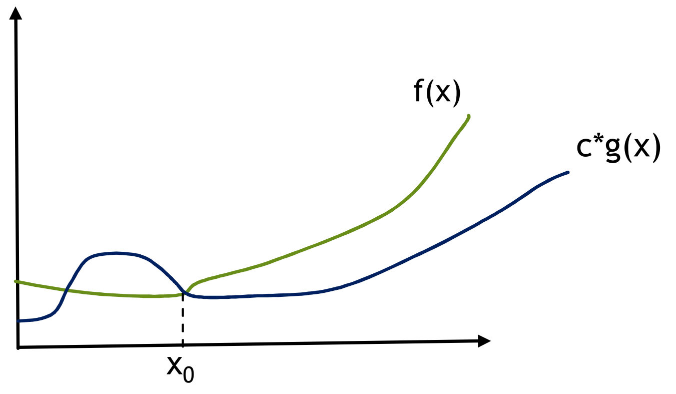


Fig: Geometrical interpretation of f(x)= Ω(g(x))

Example:

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$

Show that $f(n) = \Omega(g(n))$.

Sol^n:

To prove above relation, we need to find two positive constants c and n_0 such that $c*g(n) <= f(n)$, for all $n>=n_0$

let us choose c=1 and n_0 =1

Then,

$1*n^2 <= 3n^2 + 4n + 7$, for all $n >= 1$

Proved.

some properties of Ω-notation:

- Transitivity : if $f(x) = \Omega(g(x))$ and $g(x) = \Omega(h(x))$ then $f(x) = \Omega(h(x))$
- Reflexivity: $f(x) = \Omega(f(x))$
- $f(x) = \Omega(g(x))$ does not imply $g(x) = \Omega(f(x))$
- $\Omega(f(x)) + \Omega(g(x)) = \Omega(\min(f(x),g(x)))$
- $\Omega(f(x)) * \Omega(g(x)) = \Omega(f(x)*g(x))$
- $\Omega(1)$ is used to denote constants.

Big-theta (Θ) notation

- Big-theta notation gives asymptotic tight(exact) bound.

  A function f(x) is said to be big-theta of another function g(x), written as f(x)= Θ(g(x)), iff there exist three positive constants c_1, c_2 and x_0 such that for all x >= x_0,

  $$0 <= c_1*g(x) \leq f(x) \leq c_2*g(x)$$

- The above relation says that f(x) is order of g(x)

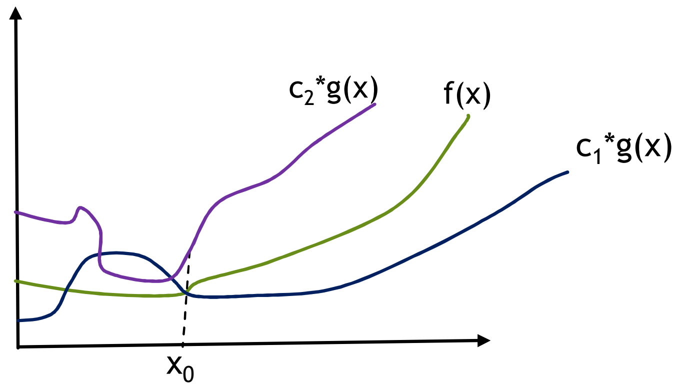


Fig: Geometrical interpretation of f(x)= Θ(g(x))

Example:

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$

Show that $f(n) = \Theta(g(n))$.

Sol^n:

To prove above relation, we need to find three positive constants c_1 , c_2 and n_0 such that $c_1 * g(n) <= f(n) <= c_2 g(n)$, for all $n >= n_0$

let us choose $c_1=1$, $c_2 = 14$ and $n_0 = 1$

Then,

$1*n^2 <= 3n^2 + 4n + 7 <= 14*n^2$, for all $n >= 1$

Proved.

Some facts about Θ-notation:

- if f(x) = O(g(x)) and f(x) = Ω(g(x)) then f(x) = Θ(h(x))
- Reflexivity: f(x) = Θ(f(x))
- If f(x) = Θ(g(x)) then g(x) = Θ(f(x))
- Θ(1) is used to denote constants.

Comparison of growth rates of some well known functions:

- $\log n < n < n \log n < n^2 < n^3 < n^4 < \ldots$
- Growth of polynomial function is less than growth of exponential function

  e.g. growth of n^2 < growth of 2^n

Recurrences

- Recurrence relations (or recurrences), in the computer algorithms, can model the complexity (particularly for divide and conquer algorithms).
- Recursions are very useful to express the complexity and can be solved easily.
- Recurrence relation is an equation or an inequality that is defined in term of itself.
- There are many real-word problems that are easily described through recursion.

e.g.

A recurrence relation has two parts:
1. Base case (boundary condition)and
2. Recursive case

- The recurrence relation for nth natural number

$$a_n = a_{n-1} + 1$$
$$a_1 = 1$$

- Recurrence relation representing complexity of algorithm for finding nth fibonacci number

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

- Complexity of Mergesort algorithm can be represented by recurrence relation

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

Solving recurrences

- There are different types of recurrence relations.
- No general procedure for solving all kinds recurrence relations is known.
- Solving the recurrence relation is an art.
- However there are generally used methods that can solve problems of specific type and having specific characteristics.
- Some methods for solving recurrences:
  1. *Iteration method*
  2. *Recursion tree method*
  3. *Substitution method*
  4. *Master method*

Note 1: (prerequisite) Please revise the various concepts of recurrence relation you studied in "Discrete math"

Note 2: The RR solving methods that we study here are particularly useful in analyzing algorithms. So we are not studying all the methods.

Note 3: we do not take care of floors, ceilings and boundary conditions in solving the relations since we are solving for asymptotic value and those conditions are of little importance.

Iteration method

- Expand the recurrence relation until the base condition is reached.
- Bound the series.

Example: solve the recurrence

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

$T(1) = 1$

Sol^n:

We can expand the recurrence as follows:

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

$= 2[2T(n/4) + 1] + 1$ $\quad$ $(= 4T(n/4) + 2 + 1)$

$= 4[2T(n/8) + 1] + 2 + 1$ $\quad$ $(= 8T(n/8) + 4 + 2 + 1)$

.. .. ..

after k steps

$= 2^kT(n/2^k) + 2^{k-1} + 2^{k-2} + \ldots\ldots 4 + 2 + 1$

$= 2^kT(n/2^k) + (2^k - 1)$

The iteration stops when base condition is reached i.e. T(1).

This happens when $2^k = n$

Then the recurrence becomes,

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

$= n + n - 1$

$= 2n - 1$

$= \Theta(n)$

Example: solve the recurrence by iteration method

$T(n) = 2T(n/2) + n$

$T(1) = 1$

Example: solve the recurrence by iteration method

$T(n) = T(n/3) + \Theta(n)$

$T(1) = \Theta(1)$

Recursion tree method

This method is just the modification to the iteration method for pictorial representation.

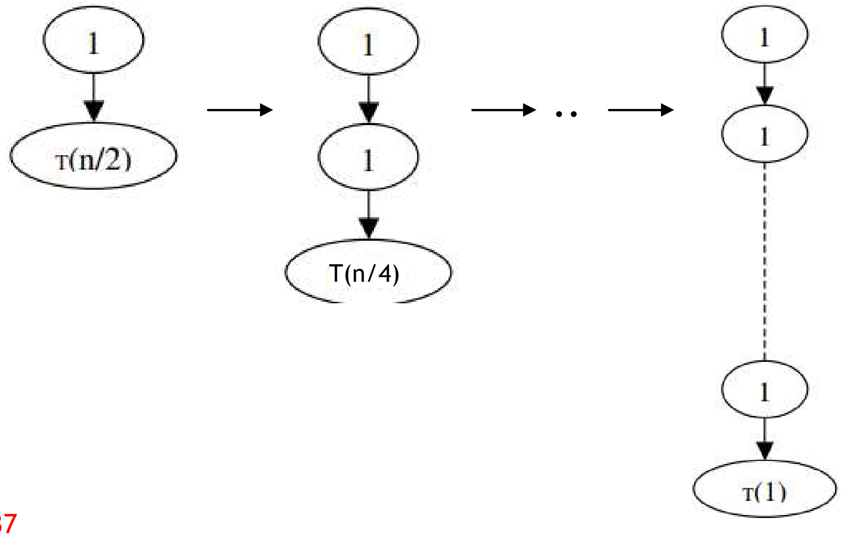
In this method, we construct a tree as follows.

- Make root node for the non-recursive term in the RR and make child node for each recursive terms.
- Expand the children nodes until base case is reached.

Example: Given a RR,

T(n) = 1

T(n) = T(n/2) + 1



Here, we add each node value and bound the summation.
Suppose the recursion stops after k steps
This happens when $n/2^k = 1$

$\Rightarrow 2^k = n$

$\Rightarrow k = \log n$

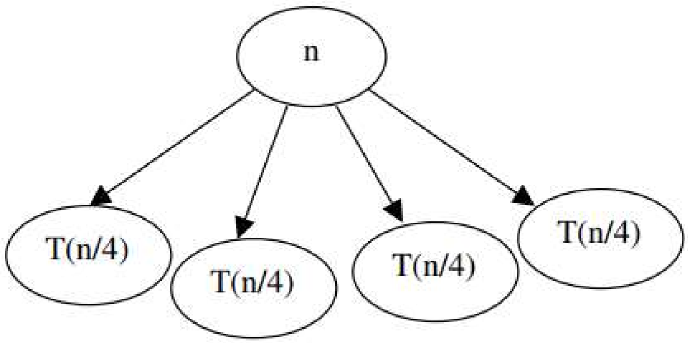
The value in each node =1
Total height of the tree = k

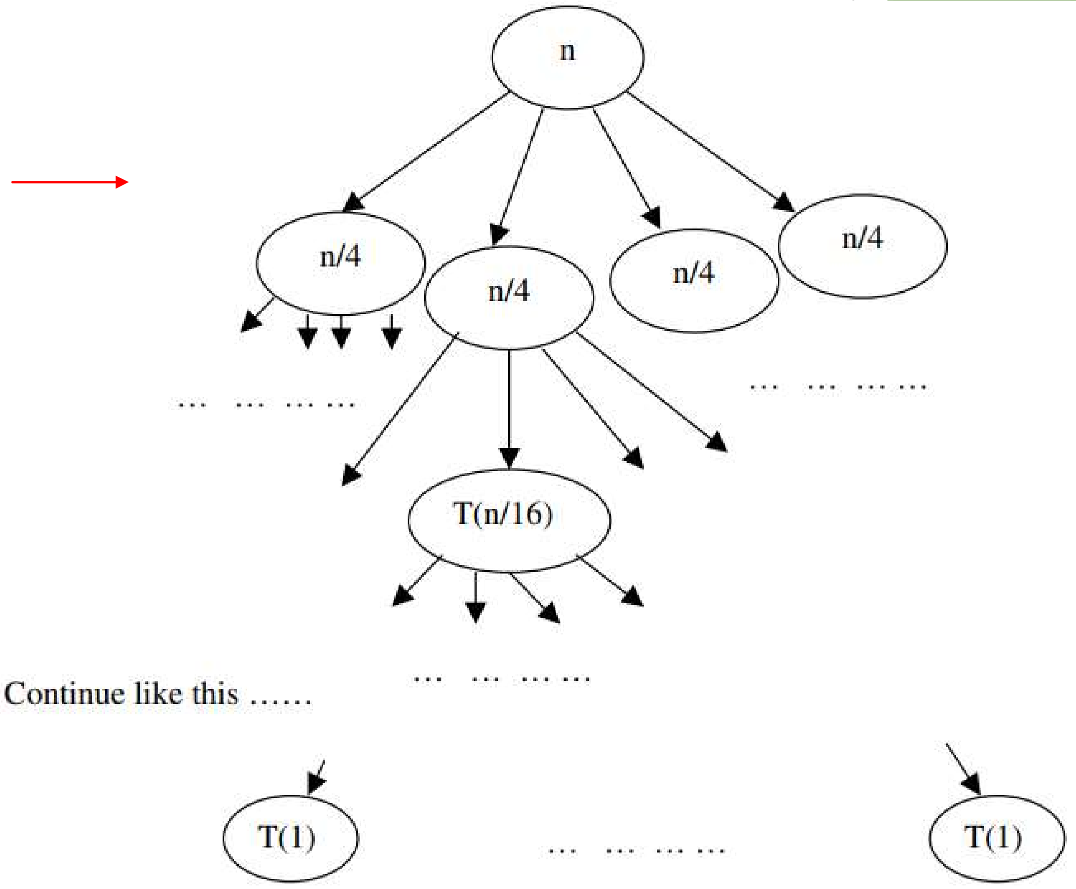
$$\begin{aligned} T(n) &= 1 + 1 + \ldots \ldots K \text{ terms} \\ &= k \\ &= \log n \\ &= \Theta(\log n) \end{aligned}$$

Example: solve the following RR by recursion tree method

$T(n) = 1$

$T(n) = 4T(n/4) + n$





T(n) = ??

Substitution method

- Guess the solution
- Prove that solution to be true by mathematical induction.

Example:

T(n) = 1, when n = 1
T(n) = T(n-1) + 1, when n > 1

Sol^n:

let us guess the solution to be

T(n) = O(n) 1

For relation 1 to be true, we must have

T(n) <= c*n for some +ve constant c

Then,

T(n/2) <= c*(n-1)

From the given recurrence relation,

T(n) <= c*(n-1) + 1
[= cn – (c - 1)] .. ..@

<=cn for any c>=1

Thus the solution is true for given recurrence.

Now we show the solution holds for boundary condition too.

Here,

T(1) <= c*1 – (c-1) (putting n=1 in eq^n @)

⇒ 1 <= 1

Thus the solution holds for boundary condition too.

Hence our guess is true.

T(n) = O(n)

Another example:

$T(1) = 1$

$T(n) = 2T(\frac{n}{2}) + n$ 1

Guess: $T(n) = O(n \log n)$

Then, we must have

$T(n) \leq c*n \log n$ for some c>0

$\Rightarrow T(\frac{n}{2}) \leq c*\frac{n}{2}\log(^n/_2)$

From 1,

$T(n) \leq 2c*\frac{n}{2}\log(^n/_2) + n$

$[= c*n\log(^n/_2) + n]$

$[= c*n(\log n - \log 2) + n]$

$[= c*n\log n - (cn - n)]$

$\leq c*n \log n$ for any c>0

Thus the solution holds for recurrence

Now we must show the solution also holds for boundary condition.

T(1) <= c*1 log1 -(c*1 – 1)

1 <= 1-c (false!! for any c>0)

OMG, is our guess incorrect ??

Since we are doing asymptotic analysis, we don't much care about specific boundary condition. Since we can have some $n>n_0$, we may choose boundary condition as n = 2,3, etc

T(2) = 2T(1) + 2 = 4

T(3) = 2T(1) + 3 = 5 (3/2 = 1 as integer division)

Now,

T(2) <= c*2log2 – (2c-2)

4 <= 4 (true)

T(3) <= c*3log3 -(3c-3)

5 <= 4.75*c -3c (true for any c>3)

Thus our guess was correct.
T(n) = O(nlogn)

How to guess ??

- Substitution method requires the solution be guessed in advance.
- This seems little bit awkward. But the method still works.
- Near accurate guessing comes from knowledge and experience.
- If we have analyzed a lot of algorithms based on recurrence. Then we can relate previously seen solutions to new problems.
- e.g.,

$$T(n) = 2T(\frac{n}{2}+3)+3+n$$

Have you seen the problem similar to this before ?

- From our experience, we can also guess the upper bound (maximum) or the lower bound (minimum) for the complexity.

Master method

- Many of the recurrence relations we encounter in the analysis of algorithms are of the form.

  $T(n) = aT(n/b) + f(n)$

  where a >=1 and b >1 are constants and $f(n)$ is asymptotically positive function.

- The above recurrence represents complexity of algorithms where problem of size n is divided into a sub problems each of size n/b where a and b are positive constants. The cost of dividing and combining the problem is given by $f(n)$.

- Such type of recurrences can be easily solved by *Master method*

Master Theorm

let a >=1 and b >1 be constants, let $f(n)$ be a function, and let T(n) be defined on the nonnegative integers by the recurrence

T(n) = a T(n/b) + $f(n)$,   Where n/b is either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

Then T(n) can be bounded asymptotically as follows:

1. If $f(n)$ = O($n^{(log_b a)-\varepsilon}$) for some constant ε > 0, then T(n) = Θ($n^{log_b a}$).
2. If $f(n)$ = Θ($n^{log_b a}$), then T(n) = Θ($n^{log_b a}$ $logn$).
3. If $f(n)$ = Ω($n^{(log_b a)+\varepsilon}$) for some constant ε > 0, and if a$f(n/b)$ ≤ c*$f(n)$ for some constant c<1 and all sufficiently large n , then T(n) = Θ($f(n)$).

Examples

1.

T(n) = 9T(n/3) + n

Here, a=9, b=3, f(n) = n

So, $n^{log_b a}$ = $n^{(log_3 9)}$ = n^2

Now,

f(n) = n = O($n^{(logba)-\varepsilon}$), where ε = 1,

Here, Case 1 of master theroem holds.

So, T(n) = Θ(n^2)

2.

T(n) = 2T(n/2) + n

Here, a=2, b=2, f(n) = n

So, $n^{log_b a}$ = $n^{(log_2 2)}$ = n

f(n) = n = Θ($n^{log_b a}$), (Case 2)

So, T(n) = Θ($nlogn$)

3.

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Here, a=2, b=2, f(n) = n^2

So, $n^{log_b a}$ = $n^{(log_2 2)}$ = n

f(n) = n^2 = Ω($n^{1+\varepsilon}$) , for any $0 < \varepsilon < 1$

Hence, case 3 holds

Limitation of master method

The master theorem cannot be used if:

1. T(n) is not monotone. e.g. T(n) = sin n
2. 'a' is not a constant. e.g. a = 2n
3. a < 1
4. Master method is not applicable in some cases.

e.g.

T(n)=2T(n/2)+nlogn

Here, a = 2, b = 2, f(n) = nlogn

So, $n^{(logba)-\varepsilon}$ = $n^{(log_2 2)}$ = n

Now,

f(n) = $n$logn ≠ O($n^{1-\varepsilon}$), for any ε>0

Hence case 1 fails.

f(n) = $n$logn ≠ Θ(n) , case 2 fails

f(n) = $n$logn ≠ Ω($n^{1+\varepsilon}$), for any ε>0

(case 3 fails)

Hence master method can not be applied here.

Another example:

T(n)=3T(n/2)-n^2

Master method does not apply. f(n) is negative.

Sometimes applying master method is tricky!.

Example:

$T(n) = 2T(\sqrt{n}) + \log n$ 1

- This equation is not in the form $aT(\frac{n}{b}) + f(n)$.
- So we can not apply master theorem directly
- But we can do some tweak to convert this relation into suitable format.

Let, $\log n = m \Rightarrow n = 2^m$

Then equation 1 becomes,

$T(2^m) = 2T(2^{m/2}) + m$ 2

Again let,

$T(2^m) = S(m)$

Then 2 becomes,

$S(m) = 2S(m/2) + m$ 3

Now this relation (3) is in suitable format where we can apply master theorem.

Finally we substitute back the values.

End of unit 1

Thank You!