1 Asymptotic sufficiently Analysis

Introduction

- An Algorithm is a combination of a sequence of finite steps to solve a problem. All steps should be finite and compulsory (If you are removing any one step, then the algorithm will be affected).
- Which algorithm is best suitable for a given application depends on the input size, the architecture of the computer, and the kind of storage to be used: disks, main memory, or even tapes.
- It should terminate after a finite time and produce one or more outputs by taking zero or more inputs.

Need For Analysis

- Problem may have many candidate solution, the majority of which do not solve the problem in hand. Finding solution that does, or one that is 'best', can be quite a challenge.
- While running a program, how much time it will take and how much space it will occupy is known as analysis of program or analysis of Algorithm.
- An algorithm analysis mostly depends on time and space complexity.

  Generally, we perform following types of analysis
  - **Worst case:**
    The maximum number of steps or resources required to solve a problem.
  - **Best case:**
    The minimum number of steps or resources required to solve a problem.
  - **Average case:**
    The average number of steps or resources required to solve a problem.
  - **Amortized:**
    A sequence of operations applied on the input of size n averaged over time.
- The order of growth of running time curve that varies with size of input helps us to know the efficiency of an algorithm and we can compare the relative performance of an alternative algorithm.
- We study asymptotic efficiency of an algorithm to know how the running time of an algorithm increases with the size of input.

Asymptotic Notations

- We use asymptotic notation primarily to describe the running times of algorithms.
- With the help of Asymptotic notation, we compare two positive functions in terms of running time.

θ-Notation

Let f(n) and g(n) be two positive functions

$f(n) = \theta(g(n))$

if and only if

$f(n) \le c_1 \cdot g(n)$

and

$f(n) \ge c_2 \cdot g(n)$

$\forall\, n \ge n_0$

such that there exists three positive constant $c_1 > 0$, $c_2 > 0$ and $n_0 \ge 1$

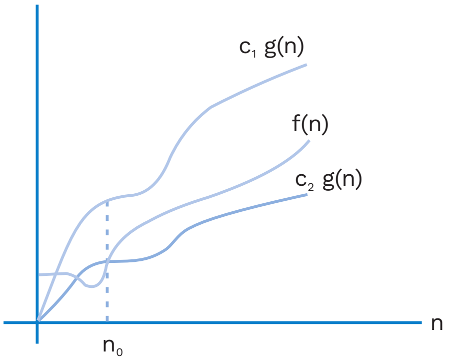


Fig. 1.2 (A)

- $f(n) \le c_1 \cdot g(n)$ and $f(n) \ge c_2 \cdot g(n)$, $\forall n$, $n \ge n_0$

O-Notation [Pronounced "big-oh"]

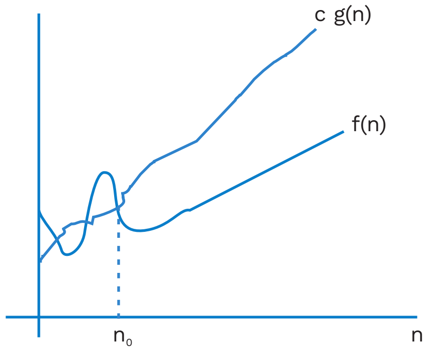
Let f(n) and g(n) be two positive functions

$f(n) = O(g(n))$

if and only if

$f(n) \leq c \cdot g(n), \forall n, \geq n_0$

such that $\exists$ two positive constants $c > 0$, $n_0 \geq 1$.



$f(n) = O(g(n))$

Fig. 1.2 (B)

e.g.,

$f(n) = 3n + 2$, $g(n) = n$

$f(n) = O(g(n))$

$f(n) \leq c\, g(n)$

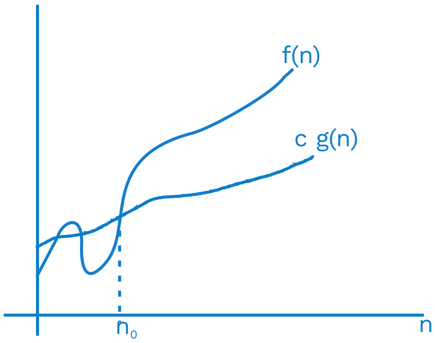
$3n + 2 \leq cn$

$c = 4$

$n_0 \geq 2$

Ω-Notation: [Pronounced "big-omega"]

- Ω notation provides an asymptotic lower bound for a given function g(n), denoted by Ω(g(n)). The set of functions
  $f(n) = \Omega(g(n))$
  if and only if
  $f(n) \geq c \cdot g(n), \forall\, n \geq n_0$
  such that $\exists$ two positive constants $c > 0$, $n_0 \geq 1$.



$f(n) = \Omega\ (g(n))$

Fig. 1.2 (C)

Theorem to Remember

Two functions, f(n) and g(n) are equivalent if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

o-Notation [Pronounced "little-oh"]

For a function f(n), we denote o(g(n)) as the set of functions

$f(n) = o(g(n))$

iff

$f(n) < c \cdot g(n), \forall\, c > 0$ whenever $n \geq n_0$

$n_0 \geq 1$.

ω-notation: [Pronounced 'little-omega"]

- By analogy, ω-notation is to Ω-notation as o-notation is to O-notation.
  For a function f(n), we denote ω(g(n)) as the set of functions.
  $f(n) = \omega(g(n))$
  if and only if
  $f(n) > c \cdot g(n), \forall\, c > 0$ whenever $n \geq n_0$
  $n_0 \geq 1$.

Some of the important properties of asymptotic notation are:

Transitivity:

- If $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$ then $f(n) = \theta(h(n))$
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$
- If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then $f(n) = \Omega(h(n))$
- If $f(n) = o(g(n))$ and $g(n) = o(h(n))$ then $f(n) = o(h(n))$
- If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ then $f(n) = \omega(h(n))$

Reflexivity:

- $f\ (n) = \theta(f(n))$
- $f\ (n) = O(f(n))$
- $f\ (n) = \Omega(f(n))$

Symmetry:

$f(n) = \theta(g(n))$ if and only if $g(n) = \theta(f(n))$

Transpose symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

$f\ (n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Rack Your Brain

Consider the following c program

```
unacademy(int n)
{
   int n = 2^2k
   for(i=1, i< = n, i++)
   {
      j = 2;
      while(j < = n)
      {
         j = j^2;
         printf("prepladder");
      }
   }
}
```

Express number of times prepladder is printed using O notation.

Comparison Function

1. Which is asymptotically larger: log(log*n) or log* (logn)

Note:
log*n (read "log star" of n) to denote the iterated logarithm

- The growth of iterated logarithm function is very slow.
  log*2 = 1,

  log*4 = 2,

  log*16 = 3,

  log*65536 = 4,

  log* (2^{65536}) = 5

$$\Rightarrow \lim_{n\to\infty} \frac{\log(\log^* n)}{\log^* (\log n)}$$

$$= \lim_{n\to\infty} \frac{\log(\log^* 2^n)}{\log^* (\log 2^n)}$$

[we have log*2^n = 1 + log*n]

$$= \lim_{n\to\infty} \frac{\log(1+\log^* n)}{\log^* n}$$

$$= \lim_{n\to\infty} \frac{\log(1+n)}{n}$$

$$= \lim_{n\to\infty} \frac{1}{1+n}$$

[∴ We have log*(logn) = 0 is asymptotically larger]

2. Table below, gives whether A is O, o, Ω, ω or θ of B. Assume k ≥ 1, ε > 0 and C > 1 are constants.

A	B	O	o	Ω	ω	θ
$\log^k n$	n^ε	w	Yes	No	No	No
n^k	C^n	Yes	Yes	No	No	No
$\sqrt{n}$	$n^{\sin n}$	No	No	No	No	No
2^n	$2^{n/2}$	No	No	Yes	Yes	No
$n^{\log c}$	$C^{\log n}$	Yes	No	Yes	No	Yes
log(n!)	$\log(n^n)$	Yes	No	Yes	No	Yes

Note:

- log(n!) = θ(nlogn) (by string's approximation)
- Fibonacci numbers are co-related to the golden ratio ϕ and to its conjugate $\hat{\phi}$, which are two roots of the equation $x^2 = x + 1$
  and are given by the following formulas

$$\phi = \frac{1+\sqrt{5}}{2} = 1.6183 \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2} = -.61803$$

Specifically, we have

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

Where

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$

in Fibonacci series.

Note:
In exam do not apply logarithm blindly, first cancel the common terms then apply logarithm.

Example 1:

Determine which is faster growing function between 2^n and n^2

Solution:
Apply log
$\log 2^n$, $\log n^2$
nlog2, 2logn

Substitute large value in n(eg, $n = 2^{100}$)
2^{100} log2, $2\log 2^{100}$
2^{100} , 100*2
$\therefore$ 2^n grows faster than n^2

Example 2:

Determine which is faster growing function between $n^{\sqrt{n}}$ and $n^{\log n}$

Solution:

Apply log

$\sqrt{n}$ logn, logn logn

Substitute $n = 2^{16}$

$\sqrt{2^{16}}\ \log 2^{16}, \log 2^{16}\ \log 2^{16}$

$2^8 * 16$, $16 * 16$

$\therefore$ $n^{\sqrt{n}}$ grows faster than $n^{\log n}$

4. Following are some of the functions in increasing order of asymptotic (big-O) growth.
$4\log\log n < 4\log n < n^{1/2}\log^4(n) < 5n < n^4 < (\log n)^{5\log n} < n^{\log n} < n^{n^{1/5}} < 5^n < 5^{5n} < (n/4)^{n/4} < n^{n/4} < 4^{n^4} < 4^{4^n} < 5^{5^n}$

Verify the above given order by applying logarithm and substituting large value in ‘n’.

Rack Your Brain

Arrange the following functions in asymptotic decreasing order.

- $f_1 = 2^{\log 2^n}$
- $f_2 = n^{2024/16}$
- $f_3 = n\log n$
- $f_4 = n^{\log}$

Recurrences

Recurrences go hand in hand with the divide and conquer paradigm, because they give us a natural way to differentiate the running times of divide and conquer algorithms.

Definitions

Same Recursive program written in the form of a mathematical relation is called Recurrence relation.

Example: $T(n) = T(2n/3) + T(n/3) + \theta(n)$

There are three methods of solving a recurrence relation:

1. Substitution Method
2. Recursive - Tree Method
3. Master’s theorem

a) Substitution method

Guess a bound and use mathematical induction method to prove if the guess is correct.

b) Recursion (Recursive) - tree method

When more than 1 recursive call is present in given recurrence relation then we go for the recursive tree method.

c) Master theorem:

If any recurrence relation is there in the form of:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where a and b are positive constant and $a \geq 1$ and $b > 1$

The substitution method for solving recurrences.

The substitution method to solve recurrences comprises two steps:

1. Write (Guess) the asymptotic (upper or lower) bound on the Recursive Program.
2. Writing a recurrence relation is nothing but writing the Recursive program in the form of Mathematical relation .

For Example:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Let’s guess that the solution is T(n) = O(nlogn)

↳ The substitution method requires us to prove

$T(n) \leq c\ n\log n$

Let’s start by assuming that this bound holds for all positive m < n, in particular for $m = \lfloor n/2 \rfloor$,

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$

Substituting into the recurrence yields

$$T(n) \le 2\left(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)\right) + n$$

$$\le cn\log(n/2) + n$$

$= cnlogn - cnlog2 + n$

$\le$ cnlogn, step holds as long as $c \ge 1$.

Examples:

1. $T(n) = \begin{cases} T(n-1) + n & \text{for } n > 1 \\ 1 \quad , & \text{if } n = 1 \end{cases}$

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

$[T(n-k) \text{ becomes } 1 \text{ for } k = n-1]$

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

$O(n^2)$

2. function ( )

```
{
    if(n>1)
  return(function(n-1))
}
```

Above function can be written as equation form as

$T(n) = \begin{cases} 1 + T(n-1); & n > 1 \\ 1 \quad ; & n = 1 \end{cases}$

$$\begin{aligned} T(n) &= 1 + 1 + T(n-2) \\ &= 2 + T(n-2) \\ &= 3 + T(n-3) \\ &= 4 + T(n-4) \\ &\ \vdots \\ &\ \vdots \\ &= K + T(n-k) \end{aligned}$$

T(n-k) will eventually become 1 when $k = n - 1$,

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

$\therefore \quad T(n) = O(n)$

Note:
Sometimes, a small algebraic manipulation can make an unknown recurrence similar to a familiar one.

Example: $T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \log n$

which looks difficult

- We can simplify this recurrence, by changing the variables.
- For convenience, renaming m = logn yields

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

  // Lets say, $T(2^m) = S(m)$ to produce the new recurrence

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

  // $s(m) = O(mlogm)$
- After back substitution from s(m) to T(n), we obtain
  $T(n) = T(2^m) = S(m) = O(mlogm)$
  $= O(logn\ log\ logn)$

Rack Your Brain

Consider the recurrence relation

$T(n) = \sqrt{n}\ T(\sqrt{n}) + 100n$

express T(n) in θ notation

The Recursive (recursion) tree method for solving recurrences:

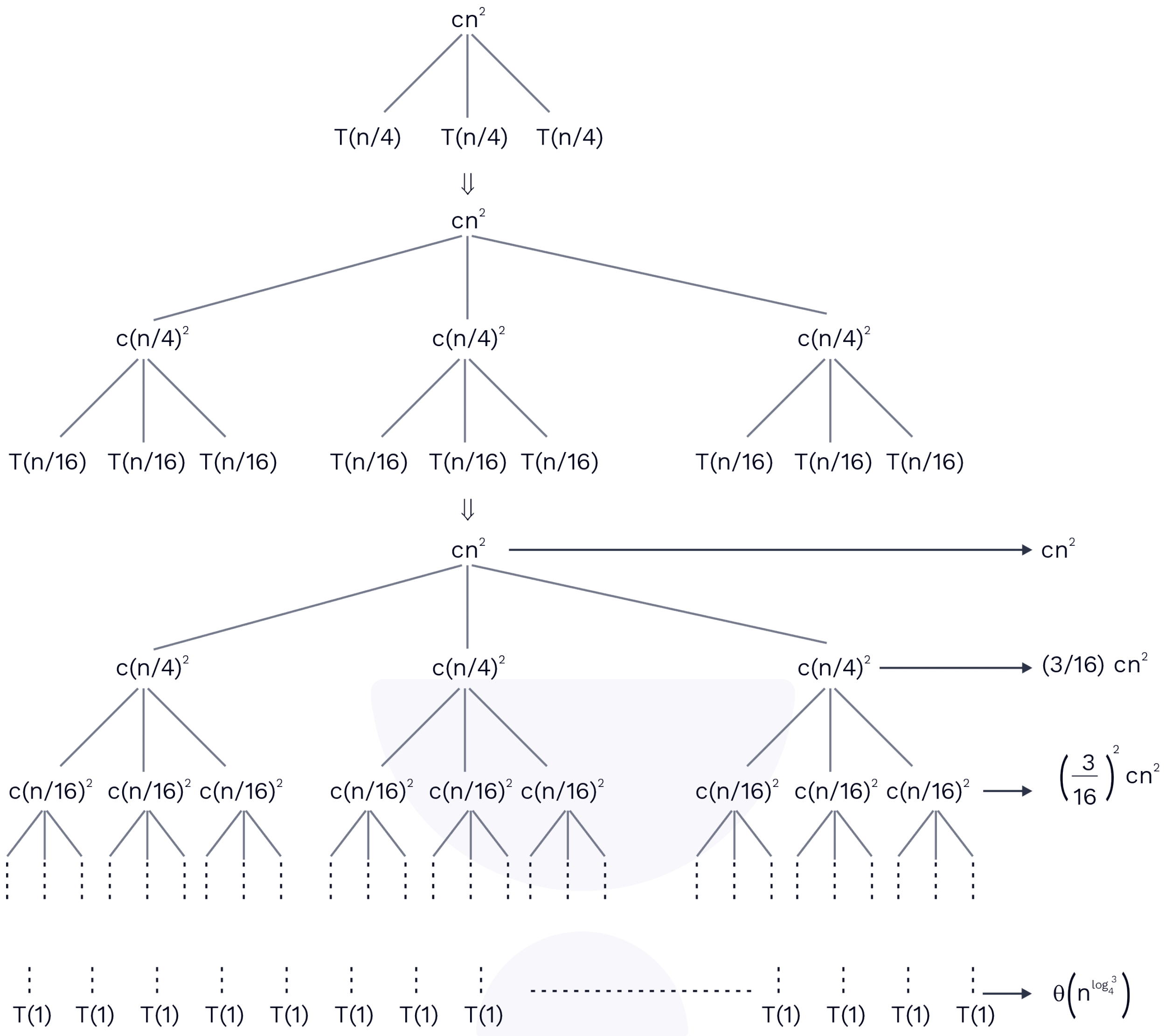
- Recursive tree method produces N-ary tree. (Where n is the recursive function call).
- Then we calculate recursive terms at each level of N-ary tree.

Examples:

i) $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

or

$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + cn^2$



- At every level, subproblem size decreases by factor 4, and we must go up to the boundary condition.
- The subproblem size for node at depth ‘i’ is (n/4^i).

  At every level, calculate the non recursive terms.
- At each level, there exists three function calls.
- At depth ‘i’ ,every node has a cost of

  $$c\left(\frac{n}{4^i}\right)^2.$$
- Multiplying, we can see that the total cost over all nodes at depth i, for i = 0, 1, 2, ..., $\log_4 n-1$, is $3^i c\left(\frac{n}{4^i}\right)^2 = \left(3/16\right)^i cn^2$
- The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost T(1), for a total cost of $n^{\log_4 3}T(1)$, which is $\theta\left(n^{\log_4^3}\right)$, since we assume that T(1) is a constant.
- To find the total complexity , we find the cost at each level and sum all the costs at each level.

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots$$

$$+\left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \theta\left(n^{\log_4 3}\right)$$

$$= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2 + \theta\left(n^{\log_4 3}\right)$$

$$< \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^{i} cn^{2} + \theta\left(n^{\log_4 3}\right)$$

$$= \frac{1}{1-\left(3/16\right)} cn^{2} + \theta\left(n^{\log_4 3}\right)$$

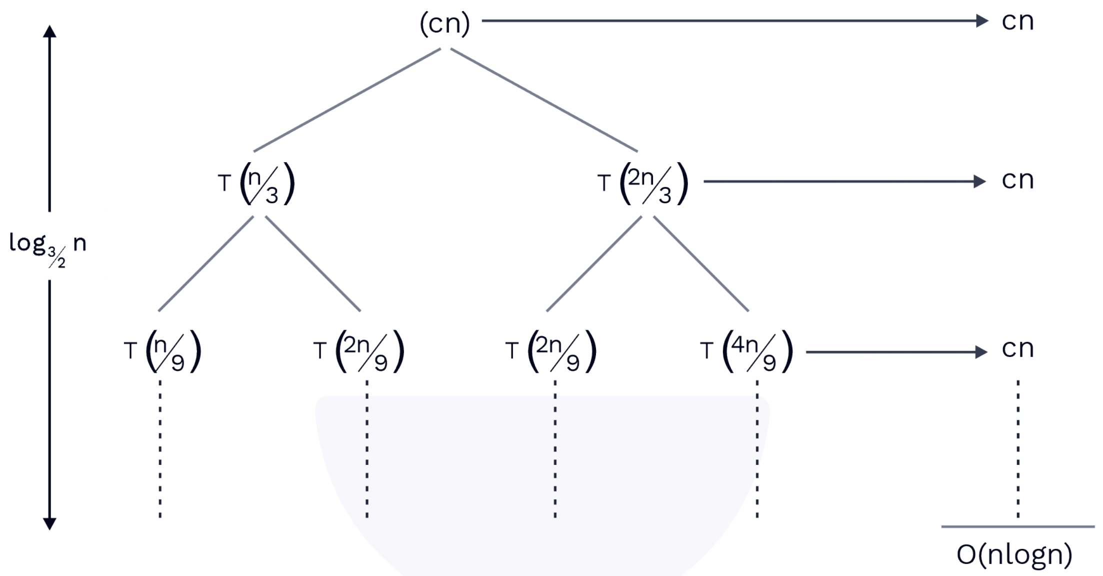
$$= \frac{16}{13} cn^{2} + \theta\left(n^{\log_4 3}\right)$$

$$= O(n^{2})$$

$\therefore$ $T(n) = O(n^2)$

Example:

ii) $T(n) = T\left(n/3\right) + T\left(2n/3\right) + O(n)$



- The largest simple path from the root to a leaf is

$$n \to \left(\frac{2}{3}\right)n \to \left(\frac{2}{3}\right)^{2} n \to \ldots\ldots \to 1$$

When $k = \log_{3/2} n$, the height of the tree is

$\log_{3/2} n$.

- By intuition, we expect the solution to the recurrence to be at most the number of levels multiplied with the cost of each level,

or $O(cn\log_{3/2} n) = O(n\log n)$

Not every level in the tree contributes to a cost of cn.

- Indeed, we can use the substitution method to verify that O(nlogn) is an upper bound for the solution to the recurrence. We have

$$T(n) \le T(n/3) + T(2n/3) + cn$$

$$\le d(n/3)\log(n/3) + d(2n/3)\log(2n/3) + cn$$

Where d is a suitable positive constant.

$$= \left(d\left(n/3\right)\log n - d\left(n/3\right)\log 3\right)$$

$$+\left(d\left(2n/3\right)\log n - d\left(2n/3\right)\log\left(3/2\right)\right) + cn$$

$$= dn\log n - d\left(\left(n/3\right)\log 3 + \left(2n/3\right)\log\left(3/2\right)\right) + cn$$

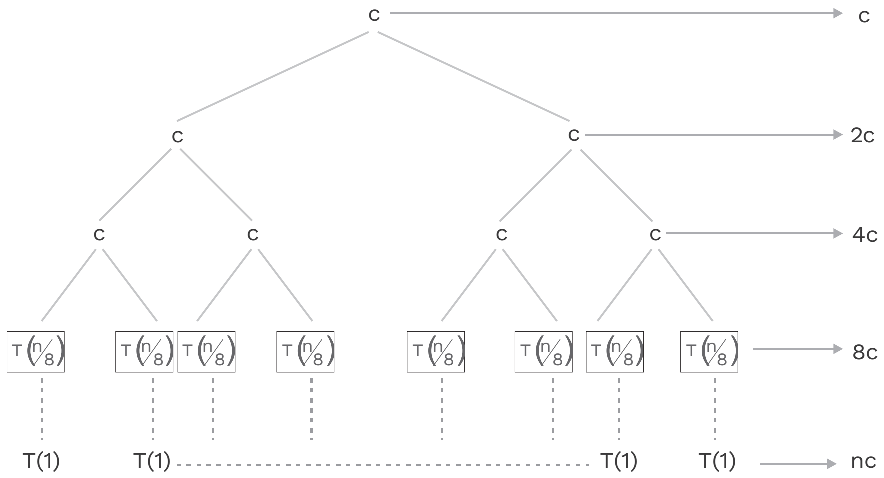
$$= dn\log n - d\begin{pmatrix}\left(n/3\right)\log 3 + \left(2n/3\right)\log 3 \\ -\left(2n/3\right)\log 2\end{pmatrix} + cn$$

$$= dn\log n - dn\left(\log 3 - 2/3\log 2\right) + cn$$

$$\le dn\log n, \text{ as long as } d \ge c/\left(\log 3 - \left(2/3\right)\log 2\right)$$

Example:

iii) $T(n) = \begin{cases} 2T\left(n/2\right) + c; & \text{if } n > 1 \\ c & ; \text{if } n = 1 \end{cases}$



- The total cost over all nodes at depth i for i = 0, 1, 2,..., logn-1, is $2^i c$

$$T(n) = \sum_{i=0}^{\log n-1} 2^i c$$

$$= c + 2c + 4c + \ldots nc$$

$$= c(1 + 2 + 4 + \ldots + n) \text{ assume } (n = 2^k)$$

$$= c(1 + 2 + 4 + \ldots + 2^k)$$

$$= c(1 + 2^1 + 2^2 + \ldots + 2^k)$$

$$= c\left(\frac{1(2^{k+1} - 1)}{2 - 1}\right)$$

$$= c(2^{k+1} - 1)$$

$$= c(2n - 1)$$

$$= O(n)$$

Rack Your Brain

Consider the recurrence relation
T(n) = T(n / 2) + 2T(n / 5) + T(n / 10) + 4n
express T(n) in 'big-oh' (O) notation

The master's theorem procedure for solving recurrences:

The master theorem is used to solve recurrence relation if and only if it is in the form of

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad \ldots(i)$$

Where $a \geq 1$ and $b > 1$ are positive constant and f(n) is a positive function.

- The recurrence (1) describes the running time of an algorithm that divides into 'a' subproblems where each is restricted by size $\frac{n}{b}$.
- The 'a' subproblems are solved recursively, each in time $T\left(\frac{n}{b}\right)$.
- The function f(n) covers the cost of dividing the problem and combining the results of the subproblems.

Master's theorem:

Let f(n) is a positive function and T (n) is defined recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Where a >= 1 and b > 1 are two positive constants.

Then T(n) follows asymptotic bounds:

Case 1:

If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > o$

then $T(n) = \theta\left(n^{\log_b a}\right)$

Case 2:

If $f(n) = \theta\left(n^{\log_b a}\right)$, then $T(n) = \theta\left(n^{\log_b a} \log n\right)$

Case 3:

If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$,

and if $a\, f\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$

and all sufficiently large n, then

$T(n) = \theta(f(n))$

- In case 1, the function $n^{\log_b a}$ is larger, then the solution is given by
  $T(n) = \theta\left(n^{\log_b a}\right)$.
- In case 3, if function f(n) is larger, then the solution is $T(n) = \theta(f(n))$.
- In case 2, if two functions are the same size, then multiply by a logarithmic factor, and the solution is $T(n) = \theta\left(n^{\log_b a} \log n\right)$
  $= \theta\left(f(n) \log n\right)$

Note:
The all three cases of Master's Theorem does not cover all the possibilities of f(n):

- There is a gap between cases 1 and 2 when f(n) is smaller than nlogba but not polynomially smaller.
- Similarly, there is a gap between cases 2 and 3 when f(n) is larger than nlogba but not polynomially larger.
- If the function f(n) falls into one of these gaps, you cannot use the master method to solve the recurrence.

Examples:

1. $T(n) = 9T\left(\frac{n}{3}\right) + n$

   For this recursion, we have a = 9, b = 3, f(n) = n, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$.

   Since $f(n) = O\left(n^{\log_3 9 - \in}\right)$. Where $\in = 1$, we can apply case 1 of master theorem.

$\therefore$ $T(n) = \theta\ (n^2)$

2. $T(n) = T\left(\frac{2n}{3}\right) + 1$

   $a = 1,\ b = \frac{3}{2},\ f(n) = 1$

   and $n^{\log_b a} = n^{\log_{3/2} 1} = n^{o} = 1$
   Case 2 of master theorem applies,
   since $f(n) = \theta\left(n^{\log_b a}\right) = \theta(1)$

$\therefore$ $T(n) = \theta\ (\log n)$

3. $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

   we get a = 3, b = 4, f(n) = nlogn, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.

   Since $f(n) = \Omega\left(n^{\log_4 3 + \in}\right)$, where $\in \approx 0.2$.

   Case 3 master theorem applies,
   For sufficiently large n, we have that
   $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right) n \log n$
   $= cf(n)$ for $c = \frac{3}{4}$.

   Consequently, by case 3
   $T(n) = \theta\ (n\log n)$

4. $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$,

   Even though it appears to have proper form a = 2, b = 2, f(n) = nlogn, and $n^{\log_b a} = n$.
   This is not polynomial larger so this does not belong to case 3

   $f(n) / n^{\log_b a} = (n \log n) / n = \log n$

   is asymptotically less than $n^{\in}$ for any positive constant $\in$.
   Consequently, the recurrence falls into the gap between case 2 and case 3.

5. $T(n) = 8T\left(\frac{n}{2}\right) + \theta\left(n^2\right)$

   a = 8, b = 2, and $f(n) = \theta\ (n^2)$,

   $n^{\log_b a} = n^{\log_2 8} = n^3$
   since n^3 is polynomially larger than f(n)

   i.e $f(n) = O(n^{3-\in})$ for $\in = 1$

   case 1 applies

   $T(n) = \theta(n^3)$

6. $T(n) = 7T\left(\frac{n}{2}\right) + \theta\left(n^2\right)$

   $a = 7,\ b = 2,\ f(n) = \theta(n^2)$

   $n^{\log_b a} = n^{\log_2 7}$

   $f(n) = O(n^{\log 7 - \in})$ for $\in = 0.8$

   Case 1 applies

$\therefore$ $T(n) = \theta(n^{\log_2 7})$

Rack Your Brain

Give the tight asymptotic bound for the recurrence relation
$T(n) = 2T(n\ /\ 4) + \sqrt{n}$

Previous Years’ Question

Consider the following three functions.

$f_1 = 10^n$ $f_2 = n^{\log n}$ $f_3 = n^{\sqrt{n}}$

Which one of the following options arranges the functions in the increasing order of asymptotic growth rate?

(A) f_3, f_2, f_1 (B) f_2, f_1, f_3

(C) f_1, f_2, f_3 (D) f_2, f_3, f_1

Solution: (D) **[GATE CS 2021 (Set-1)]**

Previous Years’ Question

There are n unsorted arrays: $A_1, A_2, \ldots, A_n$. Assume that n is odd. Each of $A_1, A_2, \ldots, A_n$ contains n distinct elements. There are no common elements between any two arrays. The worst-case time complexity of computing the median of the medians of $A_1, A_2, \ldots, A_n$ is **[GATE CS 2019]**

(A) $O(n)$ (B) $O(n\log n)$

(C) $O(n^2)$ (D) $O(n^2\log n)$

Solution: (C)

Previous Years’ Question

Consider the following C function

[GATE CSE - 2017 (Set-2)]

```
int fun (int n) {
   int i, j;
   for (i = 1; i < = n; i++) {
      for (j = 1 ; j < n ; j+=i) {
         printf (“%d %d”, i, j) ;
         }
     }
 }
```

Asymptotic notation of fun in terms of θ notation is

(A) $\theta(n\sqrt{n})$ (B) $\theta(n^2)$

(C) $\theta(n \log n)$ (D) $\theta(n^2 \log n)$

Solution:(C)

Previous Years’ Question

Consider the following functions from positive integers to real numbers:

$$10, \sqrt{n}, n, \log_2 n, \frac{100}{n}$$

The correct arrangement of the above functions in increasing order of asymptotic complexity is:

[GATE CSE 2017 (Set-1)]

(A) $\log_n n, \frac{100}{n}, 10\sqrt{n}, n$

(B) $\frac{100}{n}, 10, \log_2 n, \sqrt{n}, n$

(C) $10\frac{100}{n}, \sqrt{n}, \log_2\sqrt{n}, n$

(D) $\frac{100}{n}, \log_2 n, 10, \sqrt{n}, n$

Solution: (B)

Previous Years’ Question

In an adjacency list representation of an undirected simple graph G = (V, E), each edge (u, v) has two adjacency list entries: [v] in the adjacency list of u, and [u] in the adjacency list of v. These are called twins of each other. A twin pointer is a pointer from an adjacency list entry to its twin. If |E| = m and |V| = n, and the memory size is not a constraint, what is the time complexity of the most efficient algorithm to set the twin pointer in each entry in each adjacency list?

[GATE CSE 2016 (Set-2)]

(A) $\Theta(n2\ )$

(B) $\Theta(n + m)$

(C) $\Theta(m^2)$

(D) $\Theta(n^4)$

Solution: (B)

Chapter Summary

- **Types of analysis**
  - Worst case
  - Best case
  - Average case
  - Amortized.
- **Asymptotic notations**
  - **Big O notation:**
    f(n) = O(g(n)) if there exist constants n_0 and c such that $f(n) \leq c\ g(n)$ for all $n \geq n_0$
  - **Big omega notation:**
    $f(n) = \Omega(g(n))$ if there exist constants n_0 and c such that $f(n) \geq c\ g(n)$ for all $n \geq n_0$
  - **Theta (θ) notation:**
    If f(n) = O(g(n)) and g(n) = O(f(n)), then $f(n) = \theta(g(n))$
- **Master's theorem**
  T(n) = aT(n/b) + f(n) where a 1 and $b \geq 1$ be constant, f(n) be the function.
  - If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = (n^{\log_b a})$
  - If $f(n) = (n^{\log_b a})$, then $T(n) = (n^{\log_b a})$
  - If $f(n) = (n^{\log_b a+\epsilon})$ for constant $\epsilon > 0$, and $af(n/b) \leq c\ f(n)$ for some constant $c < 1$ for all n and all sufficiently large $T(n) = \theta(f(n)$