Introduction to Algorithm

1. **Algorithm**
   - A finite set of instructions which if followed accomplish a particular task.
   - In addition, every algorithm must satisfy following criteria:
     1. Input: zero or more quantities externally supplied
     2. Output: at least one quantity is produced
     3. Definiteness: Each instruction must be clear and unambiguous.
     4. Finiteness: In all cases algorithm must terminate after finite number of steps.
     5. Effectiveness: each instruction must be sufficiently basic.

2. **Approaches towards algorithm:**

(a) **Divide and conquer:** these techniques divide a given problem into smaller instances of the same problem, solve the smaller problems and combine solutions to solve the given problem.

(b) **Greedy algorithm:** An algorithm which always takes the best immediate or local solution while finding the answer. Greedy algorithms will find the overall or globally optimal solution for some optimization problems but may find less than optimal (suboptimal solutions) for some. These algorithms are very easy to design.

(c) **Dynamic programming Algorithm*:*** It solves sub problems just once and save the solutions in a table. The solution will be retrieve when the same problem is encountered later on.

(d) **Back tracking algorithm*:*** An algorithmic technique by trying one of the several possible choices. If the choice proves incorrect, computation backtracks or restarts at the point of choice and tries another choice.

(e) **Branch and bound** it is similar to backtracking. In backtracking we try to find one or all configurations modeled as n- tuples which satisfy certain properties. Branch and bound algorithms are oriented more toward optimization. Here a bounding function is developed to prove incorrect choice.

(f) **Approximate algorithm*:*** An algorithm to solve an optimization problem that runs in polynomial time in the length of input and outputs the solution that is guaranteed to be close to the optimal solution.

Time Complexity of algorithm:

- The time complexity of an algorithm is the amount of computer (CPU) time it needs to run to completion.

- The time T(P) taken by a program P is the sum of the compile time and the run (execution) time .

(1) ***Performance analysis of an algorithm***

There are many performance criteria to analyze/judge an algorithm:

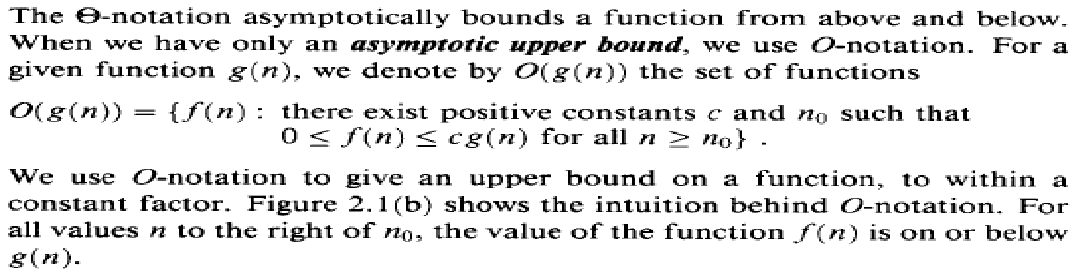
- Does it do what we want it to do?
- Does it work correctly according to the original specifications of the task?
- Is there documentation that describes how to use it and how it works?
- Are procedures created in such a way that they perform logical sub-functions?
- Is the code readable?

**Performance evaluation can be loosely divided into two major phases (1) a priori estimates and (2) a posteriori testing. We refer to these as performance analysis and performance measurement, respectively.

(2) ***Growth of functions***

- **Asymptotic notation:**
  - ✓ We are trying to study the asymptotic efficiency of algorithms that is we are concerned with how the running time of an algorithm increases with the size of the input in the limit as the size of the input increases without bound this is called **growth of function.**
  - ✓ This topic gives several standard methods for simplifying the asymptotic analysis of algorithms.
  - ✓ Why asymptotic analysis of algorithms is important?
  - ✓ A way to predict the growth of function or in other way it is useful to determine space and time complexity /the order of growth of the running time of an algorithm.
  - ✓ **Asymptotic function**: if any function either increase or decrease after a particular range. or if a curve tends to meet at infinity with a line it is said to asymptotic function.
  - ✓ **Order of growth**: It is the leading term in a function/expression for example the order of $2n^2 + 3n + 1$ will be n^2 .

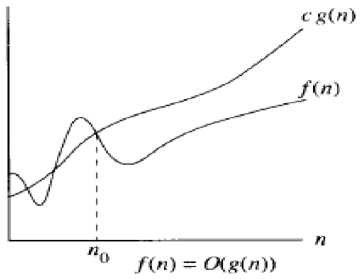
(a) O- **Notation:** Pronunciation: big oh



The Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

We use O-notation to give an upper bound on a function, to within a constant factor. Figure 2.1(b) shows the intuition behind O-notation. For all values n to the right of n_0, the value of the function $f(n)$ is on or below $g(n)$.

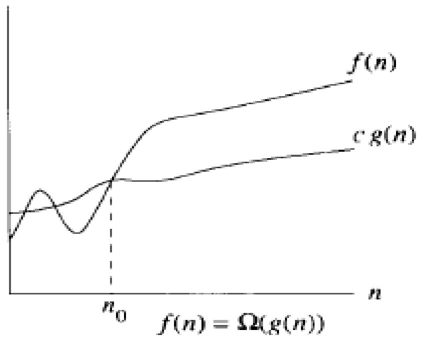


Example 1: Find the O Notation of the following: f (n) =3n+3
Solution: Order of growth /leading term of given function is ➔g(n)=n
F(n)<=c.g(n)
3n+3<=c.n
For n=1,c=4 ➔ 6<=4 Wrong
For n=2,c=4 ➔9<=8 Wrong
For n=3,c=4 ➔12<=12 Right
Hence 3n+3= O(n) is true if c=4 and n=3.

(**c) Ω Notation:** Pronunciation: big omega

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.
Examples of functions in $\Omega(n^2)$:
n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$

Example 1: Find the Ω Notation of the following: f (n) =3n+3
Solution: Order of growth /leading term of given function is ➔g(n)=n
F(n)>=c.g(n)
3n+3>=c.n
For n=1,c=1 ➔ 6>=1
For n=2,c=1 ➔9>=2
For n=3,c=1 ➔18>=5
Hence 3n+3= Ω(n) is true if c=1 and n=1, 2, 3…

(a) Θ- Notation: Pronunciation: big theta

For a given g (n), we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

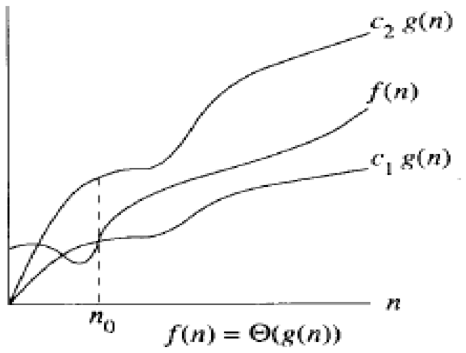


Figure 2.1(a) gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n) = \Theta(g(n))$. For all values of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \ge n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Example 1: Find the Θ- Notation of the following: f (n) =3n+3

Solution: as f (n) follow both upper and lower bound it will certainly follow Θ- notation. Prove yourself.

<u>Some facts</u>

- If we hold a constant, then the expression is strictly decreasing as b increases.

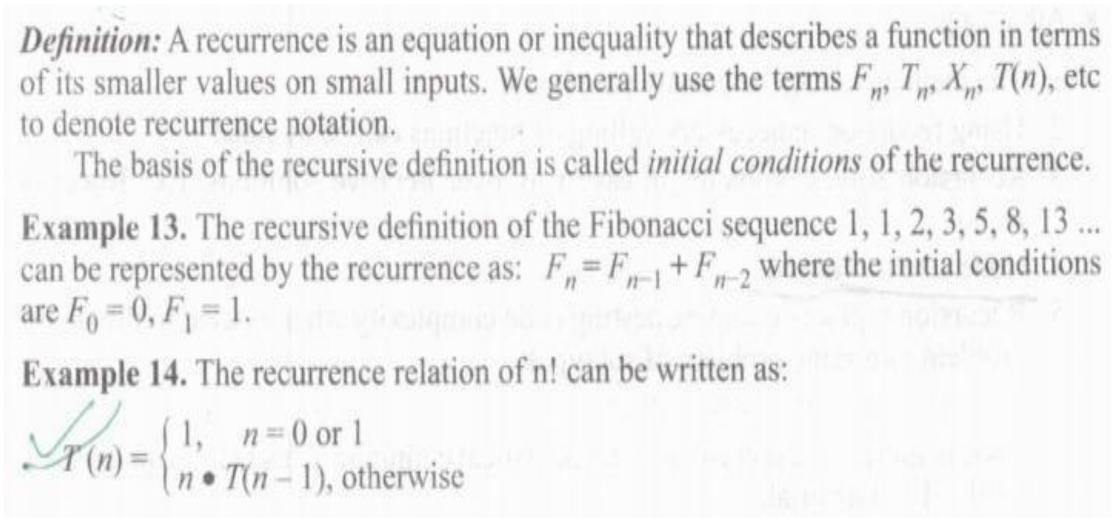
Useful identities for all real $a > 0$, $b > 0$, $c > 0$, and n, and where logarithm bases are not 1:

$$\begin{aligned}
a &= b^{\log_b a} , \\
\log_c(ab) &= \log_c a + \log_c b , \\
\log_b a^n &= n \log_b a , \\
\log_b a &= \frac{\log_c a}{\log_c b} , \\
\log_b(1/a) &= -\log_b a , \\
\log_b a &= \frac{1}{\log_a b} , \\
a^{\log_b c} &= c^{\log_b a} .
\end{aligned}$$

E

Recurrences:

1. Recurrences are useful to analyses algorithms.
2. A recurrence ia an equation or inequality that describes a function in terms of its value on smaller inputs.
3. Motive is to solve recurrences for obtaining O, Θ bounds on the solution.
In other words.



Definition: A recurrence is an equation or inequality that describes a function in terms of its smaller values on small inputs. We generally use the terms F_n, T_n, X_n, $T(n)$, etc to denote recurrence notation.

The basis of the recursive definition is called *initial conditions* of the recurrence.

Example 13. The recursive definition of the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13 ... can be represented by the recurrence as: $F_n = F_{n-1} + F_{n-2}$ where the initial conditions are $F_0 = 0, F_1 = 1$.

Example 14. The recurrence relation of n! can be written as:

$$T(n) = \begin{cases} 1, & n = 0 \text{ or } 1 \\ n \bullet T(n-1), & \text{otherwise} \end{cases}$$

Advantages:

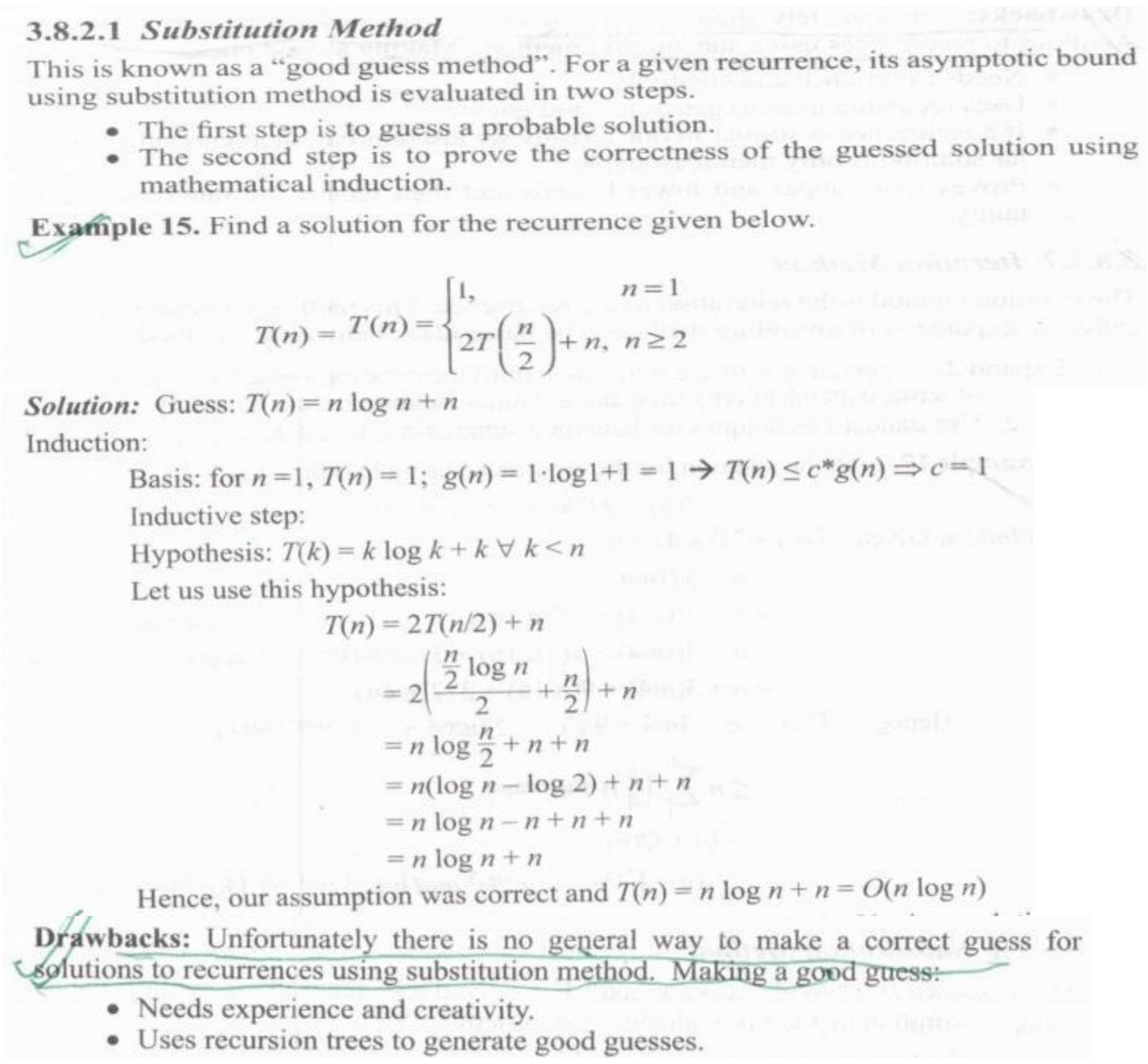
1. Unnecessary calling of functions can be avoided.
2. Solves problem in easy way over iterative solutions.
3. Code size ie reduced.
4. Complexity of nesting code can be avoided by using recursion.

Disadvantage:

1. Algo may require large amount of memory if the number of recursive call is very large.
2. High recursion may create confusion in cade.
3. Debugging is difficult at times.

Methods to solve recurrences:

1. Substitution method.
2. Iteration methos.
3. Recursion tree method.
4. Master method.



3.8.2.1 *Substitution Method*

This is known as a "good guess method". For a given recurrence, its asymptotic bound using substitution method is evaluated in two steps.

- The first step is to guess a probable solution.
- The second step is to prove the correctness of the guessed solution using mathematical induction.

Example 15. Find a solution for the recurrence given below.

$$T(n) = T(n) = \begin{cases} 1, & n = 1 \\ 2T\left(\frac{n}{2}\right) + n, & n \geq 2 \end{cases}$$

Solution: Guess: $T(n) = n \log n + n$

Induction:

Basis: for $n = 1$, $T(n) = 1$; $g(n) = 1 \cdot \log 1 + 1 = 1 \rightarrow T(n) \leq c^* g(n) \Rightarrow c = 1$

Inductive step:

Hypothesis: $T(k) = k \log k + k \; \forall \; k < n$

Let us use this hypothesis:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\left(\frac{\frac{n}{2}\log n}{2} + \frac{n}{2}\right) + n \\ &= n \log \frac{n}{2} + n + n \\ &= n(\log n - \log 2) + n + n \\ &= n \log n - n + n + n \\ &= n \log n + n \end{aligned}$$

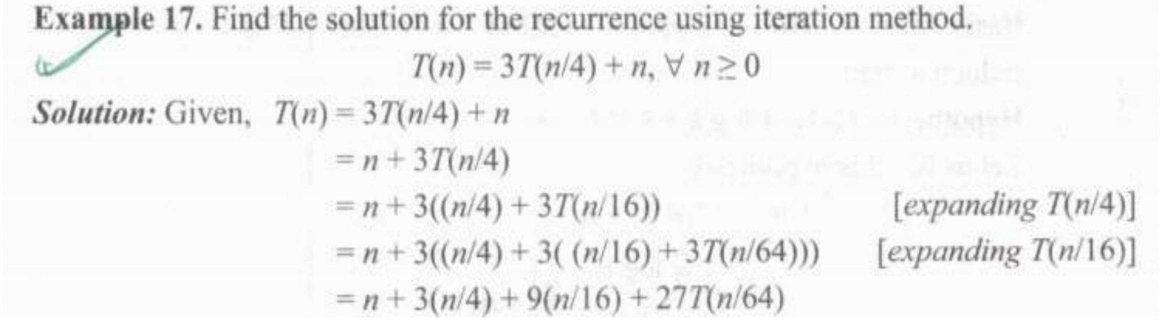
Hence, our assumption was correct and $T(n) = n \log n + n = O(n \log n)$

Drawbacks: Unfortunately there is no general way to make a correct guess for solutions to recurrences using substitution method. Making a good guess:

- Needs experience and creativity.
- Uses recursion trees to generate good guesses.

Iteration Method:

1. Expand the recurrence formula.
2. Apply summation techniques to get exact boundary and find out asymptotic bounds.

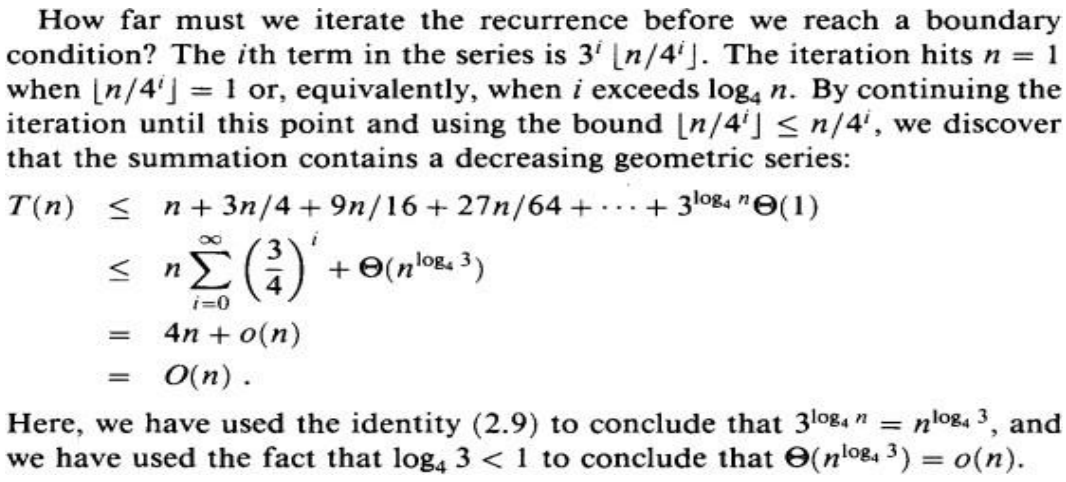


Example 17. Find the solution for the recurrence using iteration method.

$$T(n) = 3T(n/4) + n, \; \forall \; n \geq 0$$

Solution: Given,

$$\begin{aligned} T(n) &= 3T(n/4) + n \\ &= n + 3T(n/4) \\ &= n + 3((n/4) + 3T(n/16)) && [\textit{expanding } T(n/4)] \\ &= n + 3((n/4) + 3(\,(n/16) + 3T(n/64))) && [\textit{expanding } T(n/16)] \\ &= n + 3(n/4) + 9(n/16) + 27T(n/64) \end{aligned}$$

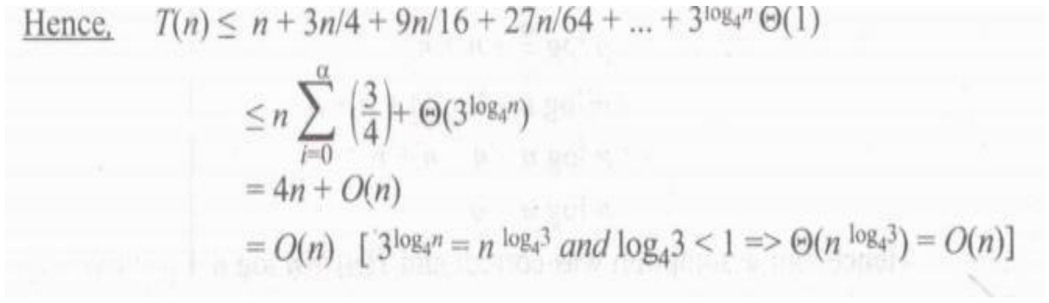


How far must we iterate the recurrence before we reach a boundary condition? The ith term in the series is $3^i \lfloor n/4^i \rfloor$. The iteration hits $n = 1$ when $\lfloor n/4^i \rfloor = 1$ or, equivalently, when i exceeds $\log_4 n$. By continuing the iteration until this point and using the bound $\lfloor n/4^i \rfloor \le n/4^i$, we discover that the summation contains a decreasing geometric series:

$$\begin{aligned} T(n) &\le n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^{\log_4 n}\Theta(1) \\ &\le n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n) . \end{aligned}$$

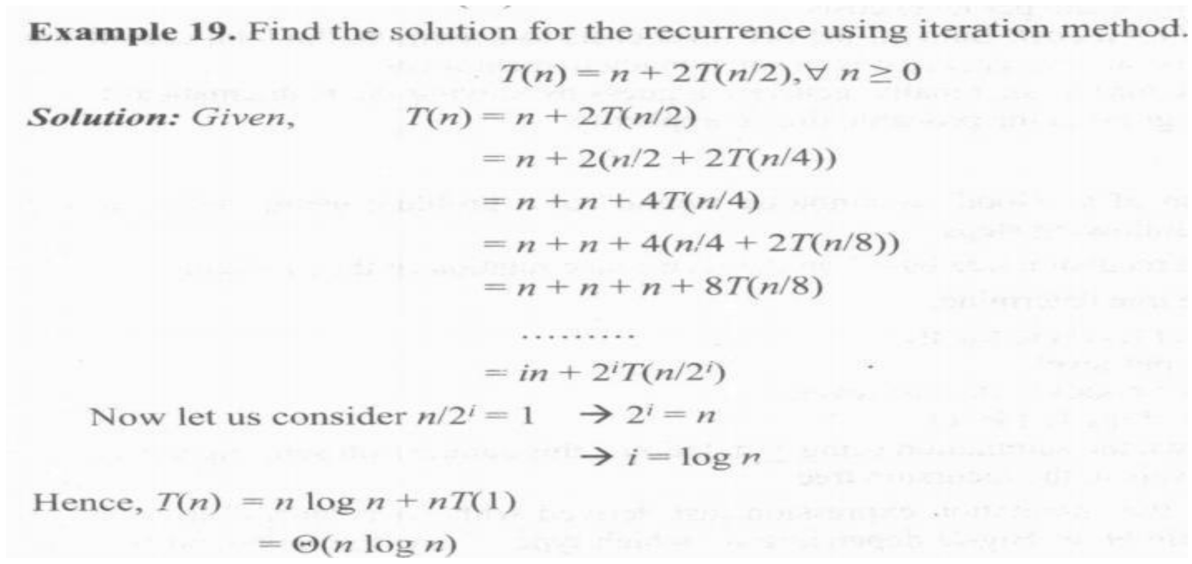
Here, we have used the identity (2.9) to conclude that $3^{\log_4 n} = n^{\log_4 3}$, and we have used the fact that $\log_4 3 < 1$ to conclude that $\Theta(n^{\log_4 3}) = o(n)$.

***** $(3^i/4^i)*n$ general term we can express it by log by the following way: $(3^i/4^i)*n \Rightarrow 3^i(n/4^i)$***
let $n/4^i=1$ then $n=4^i$ now taking log of both side $\log n = i \log 4$
$i=\log n/\log 4$ $i=\log_4{}^n$ as $\log_b{}^a=\log a/\log b$ iff log=log e now putting i in to main exp. $3^{\log_4^n}(n/4^i)$ or $3^{\log_4^n}(1) \Rightarrow 3^{\log_4^n}$



Hence, $T(n) \le n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4 n}\,\Theta(1)$

$$\begin{aligned} &\le n \sum_{i=0}^{\alpha} \left(\frac{3}{4}\right) + \Theta(3^{\log_4 n}) \\ &= 4n + O(n) \\ &= O(n) \quad [\, 3^{\log_4 n} = n^{\log_4 3} \text{ and } \log_4 3 < 1 \Rightarrow \Theta(n^{\log_4 3}) = O(n)] \end{aligned}$$



Example 19. Find the solution for the recurrence using iteration method.

$$T(n) = n + 2T(n/2), \forall\ n \ge 0$$

Solution: *Given,*

$$\begin{aligned} T(n) &= n + 2T(n/2) \\ &= n + 2(n/2 + 2T(n/4)) \\ &= n + n + 4T(n/4) \\ &= n + n + 4(n/4 + 2T(n/8)) \\ &= n + n + n + 8T(n/8) \\ &\ \ \ldots\ldots\ldots \\ &= in + 2^i T(n/2^i) \end{aligned}$$

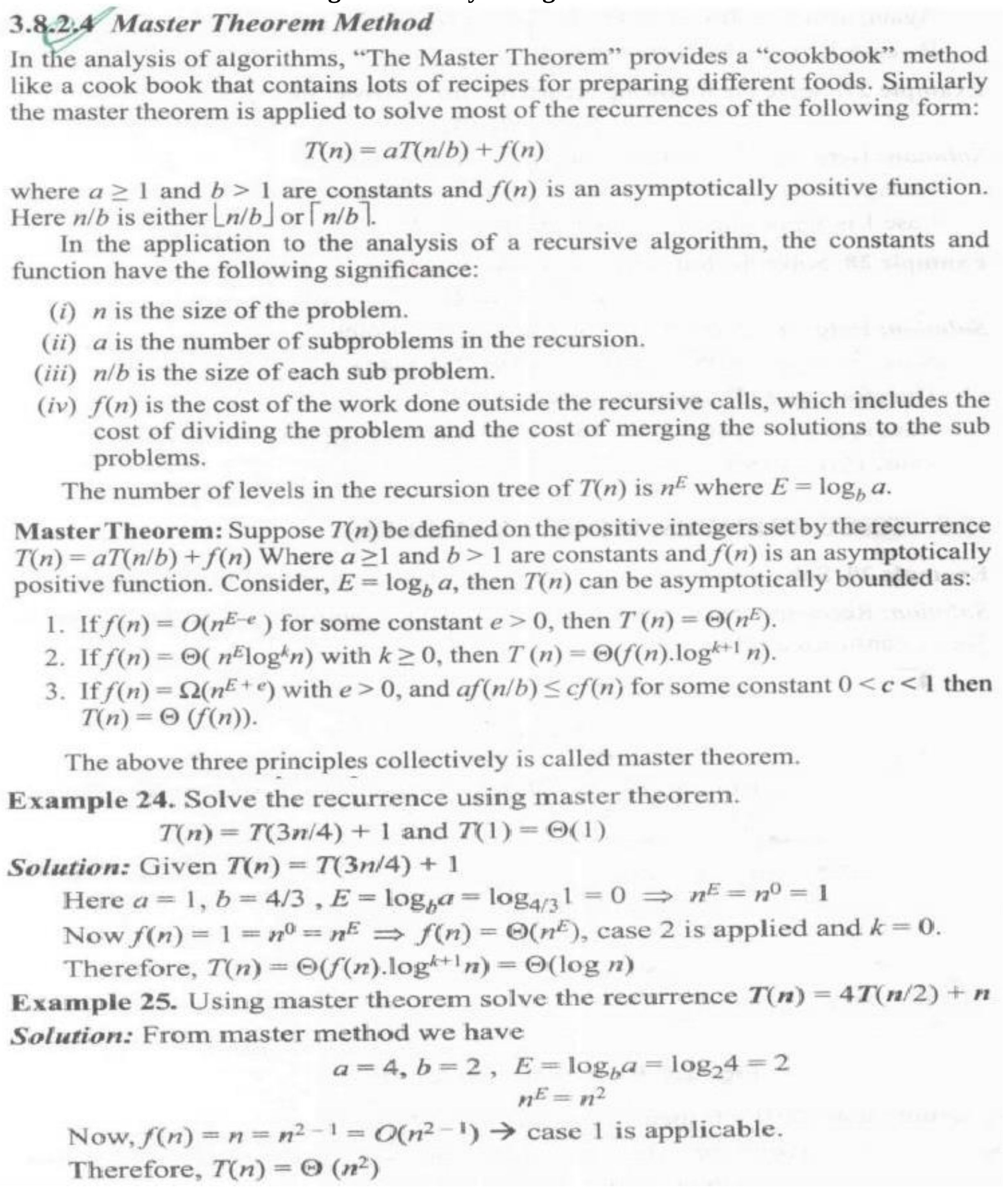
Now let us consider $n/2^i = 1 \rightarrow 2^i = n$

$\rightarrow i = \log n$

Hence, $T(n) = n \log n + nT(1)$

$= \Theta(n \log n)$

****Best Case analysis: Big omega notation**
(minimum number of steps/time taken in an algo to process inputs)
Worst Case analysis:Big oh notation
Average Case analysis:Big theta notation



3.8.2.4 *Master Theorem Method*

In the analysis of algorithms, "The Master Theorem" provides a "cookbook" method like a cook book that contains lots of recipes for preparing different foods. Similarly the master theorem is applied to solve most of the recurrences of the following form:

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. Here n/b is either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

In the application to the analysis of a recursive algorithm, the constants and function have the following significance:

(*i*) n is the size of the problem.
(*ii*) a is the number of subproblems in the recursion.
(*iii*) n/b is the size of each sub problem.
(*iv*) $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the sub problems.

The number of levels in the recursion tree of $T(n)$ is n^E where $E = \log_b a$.

Master Theorem: Suppose $T(n)$ be defined on the positive integers set by the recurrence $T(n) = aT(n/b) + f(n)$ Where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. Consider, $E = \log_b a$, then $T(n)$ can be asymptotically bounded as:

1. If $f(n) = O(n^{E-e})$ for some constant $e > 0$, then $T(n) = \Theta(n^E)$.
2. If $f(n) = \Theta(n^E \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(f(n).\log^{k+1} n)$.
3. If $f(n) = \Omega(n^{E+e})$ with $e > 0$, and $af(n/b) \leq cf(n)$ for some constant $0 < c < 1$ then $T(n) = \Theta(f(n))$.

The above three principles collectively is called master theorem.

Example 24. Solve the recurrence using master theorem.

$$T(n) = T(3n/4) + 1 \text{ and } T(1) = \Theta(1)$$

Solution: Given $T(n) = T(3n/4) + 1$

Here $a = 1$, $b = 4/3$, $E = \log_b a = \log_{4/3} 1 = 0 \Rightarrow n^E = n^0 = 1$

Now $f(n) = 1 = n^0 = n^E \Rightarrow f(n) = \Theta(n^E)$, case 2 is applied and $k = 0$.

Therefore, $T(n) = \Theta(f(n).\log^{k+1} n) = \Theta(\log n)$

Example 25. Using master theorem solve the recurrence $T(n) = 4T(n/2) + n$

Solution: From master method we have

$$a = 4,\ b = 2\ ,\ E = \log_b a = \log_2 4 = 2$$
$$n^E = n^2$$

Now, $f(n) = n = n^{2-1} = O(n^{2-1}) \rightarrow$ case 1 is applicable.

Therefore, $T(n) = \Theta(n^2)$