1. Algorithms

An algorithm is a type of effective method in which a definite list of well-defined instructions for completing a task; that given an initial state, will proceed through a well-defined series of successive states, eventually terminating in an end-state. The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as finding the common divisor of two numbers or multiplying two numbers.

Algorithms are named for the 9th century Persian mathematician Al-Khowarizmi. He wrote a treatise in Arabic in 825 AD, *On Calculation with Hindu Numerals.* It was translated into Latin in the 12th century as *Algoritmi de numero Indorum*, which title was likely intended to mean "[Book by] Algoritmus on the numbers of the Indians", where "Algoritmi" was the translator's rendition of the author's name in the genitive case; but people misunderstanding the title treated *Algoritmi* as a Latin plural and this led to the word "algorithm" (Latin *algorismus*) coming to mean "calculation method".

1.1 Algorithm Specification

The criteria for any set of instruction for an algorithm is as follows:

- Input : Zero of more quantities that are externally applied
- Output : At least one quantity is produced
- Definiteness : Each instruction should be clear and unambiguous
- Finiteness : Algorithm terminates after finite number of steps for all test cases.
- Effectiveness : Each instruction is basic enough for a person to carried out using a pen and paper. That means ensure not only definite but also check whether feasible or not.

1.2 Algorithm Classification

There are various ways to classify algorithms. They are as follows

1.2.1 Classification by implementation

Recursion or **iteration**: A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition

matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, towers of hanoi is well understood in recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.

Logical: An algorithm may be viewed as controlled logical deduction. This notion may be expressed as:

Algorithm = logic + control.

The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the logic programming paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the axioms has a well defined change in the algorithm.

Serial or **parallel** or **distributed**: Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms or distributed algorithms. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilise multiple machines connected with a network. Parallel or distributed algorithms divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together. The resource consumption in such algorithms is not only processor cycles on each processor but also the communication overhead between the processors. Sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.

Deterministic or **non-deterministic**: Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithm solves problems via guessing although typical guesses are made more accurate through the use of heuristics.

Exact or **approximate**: While many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

1.2.2 Classification by Design Paradigm

Divide and conquer. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively), until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in conquer phase by merging them. A simpler variant of divide and conquer is called decrease and conquer algorithm, that solves an identical sub problem and uses the solution of this sub problem to solve the bigger problem. Divide and conquer divides the problem into multiple sub problems and so conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is binary search algorithm.

Dynamic programming. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to sub problems, and overlapping sub problems, meaning the same sub problems are used to solve many different problem instances, a quicker approach called *dynamic programming* avoids recomputing solutions that have already been computed. For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that sub problems are more or less independent in divide and conquer, whereas sub problems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When sub problems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex problems. By using memoization or maintaining a table of sub problems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

The greedy method. A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the sub problems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. The greedy method extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best decision (not all possible decisions) made in previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Kruskal.

Linear programming. When solving a problem using linear programming, specific inequalities involving the inputs are found and then an attempt is made to maximize (or minimize) some linear function of the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm.

Reduction. This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as *transform and conquer*.

Search and enumeration. Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

The probabilistic and heuristic paradigm. Algorithms belonging to this class fit the definition of an algorithm more loosely.

Probabilistic algorithms are those that make some choices randomly (or pseudo-randomly); for some problems, it can in fact be proven that the fastest solutions must involve some randomness.

Genetic algorithms attempt to find solutions to problems by mimicking biological evolutionary processes, with a cycle of random mutations yielding successive generations of "solutions". Thus, they emulate reproduction and "survival of the fittest". In genetic programming, this approach is extended to algorithms, by regarding the algorithm itself as a "solution" to a problem.

Heuristic algorithms, whose general purpose is not to find an optimal solution, but an approximate solution where the time or resources are limited. They are not practical to find perfect solutions. An example of this would be local search, or simulated annealing.

1.3 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. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.
Recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

For example consider factorial of a number, n
n! = n*(n-1)*(n-2)*...*2*1, and that 0! = 1.

In other words,

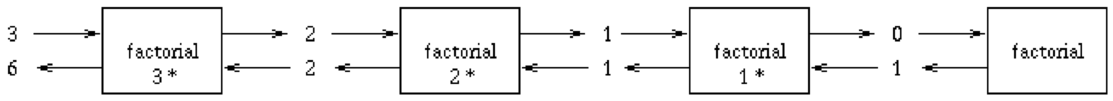
$$\text{For } n \geq 0,\ n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

Function to calculate the factorial can be written as

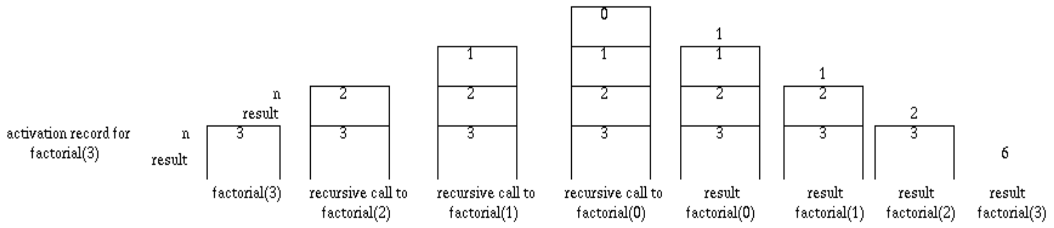
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return (n * factorial(n-1));
}
```

```
factorial(0)=> 1

factorial(3)
3 * factorial(2)
3 * 2 * factorial(1)
3 * 2 * 1 * factorial(0)
3 * 2 * 1 * 1
=> 6
```



This corresponds very closely to what actually happens on the execution stack in the computer's memory.


activation record for factorial(3)
n
result
factorial(3)
recursive call to factorial(2)
recursive call to factorial(1)
recursive call to factorial(0)
result factorial(0)
result factorial(1)
result factorial(2)
result factorial(3)
6

1.4 Space and Time Complexity

Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity.

1.4.1 Space Complexity

Space complexity of an algorithm is the amount to memory needed by the program for its completion. Space needed by a program has the following components:

1. **Instruction Space**
   Space needed to store the compiled version of program. It depends on
   i. Compiler used
   ii. Options specified at the time of compilation
   e.g., whether optimization specified, Is there any overlay option etc.
   iii. Target computer
   e.g., For performing floating point arithmetic, if hardware present or not.

2. **Data Space**
   Space needed to store constant and variable values. It has two components:
   i. Space for constants:
   e.g., value '3' in program 1.1
   Space for simple variables:
   e.g., variables a,b,c in program 1.1

Program 1.1

```
int add (int a, int b, int c)
{
        return (a+b+c)/3;
}
```

ii. Space for component variables like arrays, structures, dynamically allocated memory.
e.g., variables a in program 1.2

Program 1.2

```
int Radd (int a[], int n)
{
    If (n>0)
        return Radd (a, n-1) + a[n-1];
    else
        return 0;
}
```

3. **Environment stack space**
   Environment stack is used to store information to resume execution of partially completed functions. When a function is invoked, following data are stored in Environment stack.
   i. Return address.
   ii. Value of local and formal variables.
   iii. Binding of all reference and constant reference parameters.

   Space needed by the program can be divided into two parts.
   i. Fixed part independent of instance characteristics. E.g., code space, simple variables, fixed size component variables etc.
   ii. Variable part. Space for component variables with space depends on particular instance. Value of local and formal variables.

   Hence we can write the space complexity as
   $S(P) = c + S_p$ (instance characteristics)

Example 1.1
Refer Program 1.1
One word for variables a,b,c. No instance characteristics. Hence $S_p(TC) = 0$

Example 1.2
Program 1.3

```
int Aadd (int *a, int n)
{
    int s=0;
    for (i=0; i<n; i++)
        s+ = a[i];
    return s;
}
```

One word for variables n and i. Space for a[] is address of a[0]. Hence it requires one word. No instance characteristics. Hence $S_p(TC) = 0$

Example 1.3
Refer Program 1.2

Instance characteristics depend on values of n. Recursive stack space includes space for formal parameters, local variables and return address. So one word each for a[],n, return address and return variables. Hence for each pass it needs 4 words. Total recursive stack space needed is 4(n).
Hence $S_p(TC) = 4(n)$.

1.4.2 Time Complexity

Time complexity of an algorithm is the amount of time needed by the program for its completion. Time taken is the sum of the compile time and the execution time. Compile time does not depend on instantaneous characteristics. Hence we can ignore it.

Program step: A program step is syntactically or semantically meaningful segment of a program whose execution time is independent of instantaneous characteristics. We can calculate complexity in terms of

1. Comments:
   No executables, hence step count = 0

2. Declarative Statements:
   Define or characterize variables and constants like (*int* , *long*, *enum*, …)
   Statement enabling data types (*class, struct, union, template*)
   Determine access statements (*public, private, protected, friend*)
   Character functions (*void, virtual*)
   All the above are non executables, hence step count = 0

3. Expressions and Assignment Statements:
   Simple expressions : Step count = 1. But if expressions contain function call, step count is the cost of the invoking functions. This will be large if parameters are passed as call by value, because value of the actual parameters must assigned to formal parameters.

   Assignment statements : General form is <variable> = <expr>. Step count = expr, unless size of <variable> is a function of instance characteristics. eg., a = b, where a and b are structures. In that case, Step count = size of <variable> + size of < expr >

4. Iterative Statements:

   While <expr> **do**
   Do .. While <expr>
   Step count = Number of step count assignable to <expr>

   For (<init-stmt>; <expr1>; <expr2>)
   Step count = 1, unless the <init-stmt>, <expr1>,<expr2> are function of instance characteristics. If so, first execution of control part has step count as sum of count of <init-stmt> and <expr1>. For remaining executions, control part has step count as sum of count of <expr1> and <expr2>.

5. Switch Statements:
   Switch (<expr>) {
   Case cond1 : <statement1>
   Case cond2 : <statement2>
   .
   .
   Default : <statement>
   }

   Switch (<expr>) has step count = cost of <expr>
   Cost of **Cond** statements is its cost plus cost of all preceding statements.

6. If-else Statements:
   If (<expr>) <statement1>;
   Else <statement2>;
   Step count of **If** and **Else** is the cost of <expr>.

7. Function invocation:
   All function invocation has Step count = 1, unless it has parameters passed as called by value which depend s on instance characteristics. If so, Step count is the sum of the size of these values.
   If function being invoked is recursive, consider the local variables also.

8. Memory management Statements:
   new object, **delete** object, **sizeof**(object), Step count =1.

9. Function Statements:
   Step count = 0, cost is already assigned to invoking statements.

10. Jump Statements:
    continue, break, goto has Step count =1
    return <expr>: Step count =1, if no *expr* which is a function of instance characteristics. If there is, consider its cost also.

Example 1.4
Refer Program 1.2
Introducing a counter for each executable line we can rewrite the program as

```
int Radd (int a[], int n)
{
    count++ // if
    If (n>0)
    {
        count++ // return
        return Radd (a, n-1) + a[n-1];
      }
     else
     {
        count++ // return
        return 0;
    }
}
```

Case 1: n=0

$t_{Radd} = 2$

Case 2: n>0

$2 + t_{Radd}(n-1)$

$= 2 + 2 + t_{Radd}(n-2)$

$= 2 * 2 + t_{Radd}(n-2)$

.

.

$= 2n + t_{Radd}(0)$

$= 2n + 2$

Example 1.5
Program 1.4

```
int Madd (int a[][], int b[][], int c[][], int n)
{
    For (int i=0; i<m; i++)
        For (int j=0; j<n; j++)
                c[i][j] = a[i][j] + b[i][j];
}
```

Introducing a counter for each executable line we can rewrite the program as

```
int Madd (int a[][], int b[][], int c[][], int n)
{
    For (int i=0; i<m; i++)
     {
        count++ //for i
        For (int j=0; j<n; j++)
        {
                count++ //for j
                c[i][j] = a[i][j] + b[i][j];
                count++ //for assignment
        }
        count++ //for last j
    }
    count++ //for last i
}
```

Step count is 2mn + 2m +1.

Step count does not reflect the complexity of statement. It is reflected in step per execution (s/e).

Refer Program 1.2

Line	s/e	Frequency	Total Steps

		n=0	n>0	n=0	n>0
1	0	1	1	0	0
2	1	1	1	1	1
3	1 + t_{Radd} (n-1)	0	1	0	1 + t_{Radd} (n-1)
4	0	1	0	0	0
5	1	1	0	1	0
Total no. of steps				**2**	**2 + t_{Radd} (n-1)**

Refer Program 1.3

Line	s/e	Frequency	Total Steps
1	0	1	0
2	1	1	1
3	1	n+1	n+1
4	1	n	n
5	1	1	1
6	0	1	0
Total no. of steps			**2n + 3**

Refer Program 1.4

Line	s/e	Frequency	Total Steps
1	0	1	0
2	1	m+1	m+1
3	1	m(n+1)	m(n+1)
4	1	mn	mn
5	0	1	0
Total no. of steps			**2mn + 2m + 1**

1.5 **Asymptotic Notations**

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Since the values are not exact quantities we need only comparative statements like $c_1n^2 \leq t_p(n) \leq c_2n^2$.

For example, consider two programs with complexities $c_1n^2 + c_2n$ and c_3n respectively. For small values of n, complexity depend upon values of c_1, c_2 and c_3. But there will also be an n beyond which complexity of c_3n is better than that of $c_1n^2 + c_2n$.This value of n is called break-even point. If this point is zero, c_3n is always faster (or at least as fast). Common asymptotic functions are given below.

Function	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	n log n
n^2	Quadratic

n^3	Cubic
2^n	Exponential
n!	Factorial

1.5.1 Big 'Oh' Notation (O)

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

It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as

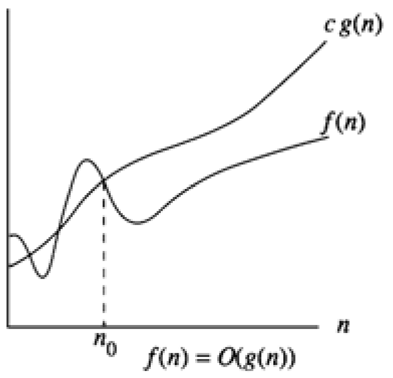


Fig 1.1

Find the Big 'Oh' for the following functions:

<u>Linear Functions</u>

Example 1.6

f(n) = 3n + 2

General form is $f(n) \le cg(n)$

When $n \ge 2$, $3n + 2 \le 3n + n = 4n$

Hence f(n) = O(n), here c = 4 and $n_0 = 2$

When $n \ge 1$, $3n + 2 \le 3n + 2n = 5n$

Hence f(n) = O(n), here c = 5 and $n_0 = 1$

Hence we can have different c,n_0 pairs satisfying for a given function.

Example 1.7

f(n) = 3n + 3

When $n \ge 3$, $3n + 3 \le 3n + n = 4n$

Hence f(n) = O(n), here c = 4 and $n_0 = 3$

Example 1.8

$f(n) = 100n + 6$
When $n \geq 6$, $100n + 6 \leq 100n + n = 101n$
Hence $f(n) = O(n)$, here $c = 101$ and $n_0 = 6$

__Quadratic Functions__
Example 1.9
$f(n) = 10n^2 + 4n + 2$
When $n \geq 2$, $10n^2 + 4n + 2 \leq 10n^2 + 5n$
When $n \geq 5$, $5n \leq n^2$, $10n^2 + 4n + 2 \leq 10n^2 + n^2 = 11n^2$
Hence $f(n) = O(n^2)$, here $c = 11$ and $n_0 = 5$

Example 1.10
$f(n) = 1000n^2 + 100n - 6$
$f(n) \leq 1000n^2 + 100n$ for all values of n.
When $n \geq 100$, $5n \leq n^2$, $f(n) \leq 1000n^2 + n^2 = 1001n^2$
Hence $f(n) = O(n^2)$, here $c = 1001$ and $n_0 = 100$

__Exponential Functions__
Example 1.11
$f(n) = 6*2^n + n^2$
When $n \geq 4$, $n^2 \leq 2^n$
So $f(n) \leq 6*2^n + 2^n = 7*2^n$
Hence $f(n) = O(2^n)$, here $c = 7$ and $n_0 = 4$

__Constant Functions__
Example 1.12
$f(n) = 10$
$f(n) = O(1)$, because $f(n) \leq 10*1$

1.5.2 Omega Notation (Ω)

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$
It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as

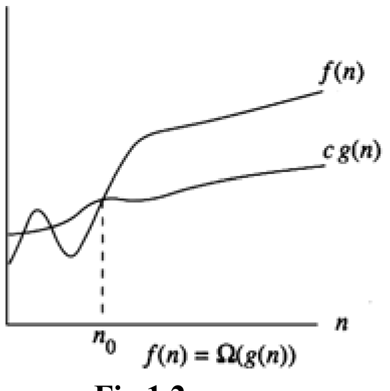


Fig 1.2

Example 1.13
$f(n) = 3n + 2$
$3n + 2 > 3n$ for all n.
Hence $f(n) = \Omega(n)$
Similarly we can solve all the examples specified under Big 'Oh'.

1.5.3 Theta Notation (Θ)

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$
If $f(n) = \Theta(g(n))$, all values of n right to n_0 f(n) lies on or above $c_1 g(n)$ and on or below $c_2 g(n)$. Hence it is asymptotic tight bound for f(n).

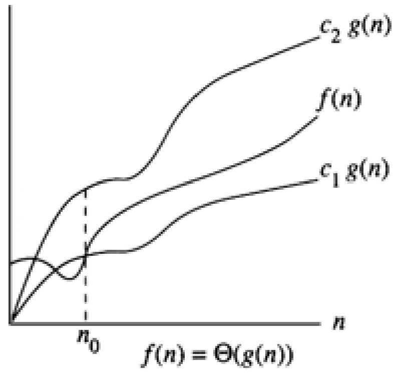


Fig 1.3

Example 1.14
$f(n) = 3n + 2$
$f(n) = \Theta(n)$ because $f(n) = O(n)$, $n \ge 2$.

Similarly we can solve all examples specified under Big'Oh'.

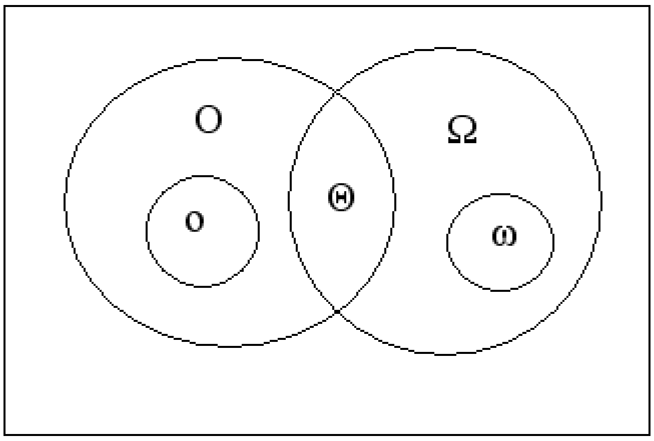
1.5.4 Little 'Oh' Notation (o)

$o(g(n)) = \{ f(n)$: for any positive constants $c > 0$, there exists $n_0>0$, such that $0 \le f(n) < cg(n)$ for all $n \ge n_0 \}$

It defines the asymptotic *tight* upper bound. Main difference with Big Oh is that Big Oh defines for some constants c by Little Oh defines for all constants.

1.5.5 Little Omega (ω)

$\omega(g(n)) = \{ f(n)$: for any positive constants $c>0$ and $n_0>0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0 \}$
It defines the asymptotic *tight* lower bound. Main difference with Ω is that, ω defines for some constants c by ω defines for all constants.



1.6 Recurrence Relations

Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases
e.g., recurrence for Merge-Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if n } = 1 \\ 2T(n/2) + \Theta(n) & \text{if n } > 1 \end{cases}$$

- Useful for analyzing recurrent algorithms
- Make it easier to compare the complexity of two algorithms
- Methods for solving recurrences
  - Substitution method
  - Recursion tree method
  - Master method
  - Iteration method

1.6.1 Substitution Method

- Use mathematical induction to derive an answer
- Derive a function of n (or other variables used to express the size of the problem) that is not a recurrence so we can establish an upper and/or lower bound on the recurrence
- May get an exact solution or may just get upper or lower bounds on the solution

Steps

- Guess the form of the solution
- Use mathematical induction to find constants or show that they can be found and to prove that the answer is correct

Example 1.15
Find the upper bound for the recurrence relation

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

Guess the solution as T(n) = O(n.lg(n))
Then T(n) = c.n.lgn
Substituting in T(n), we get

$$\begin{aligned}T(n) &= 2(c.\lfloor n/2 \rfloor .lg(\lfloor n/2 \rfloor)) + n\\ &= c\,n\,lg(n/2) + n\\ &= cn\,lg(n) - cnlg(2) + n\\ &= cn\,lg(n) - cn + n\\ &= cn\,lg(n),\ \ c >= 1\end{aligned}$$

To prove using mathematical induction, we have to show that the solution holds for boundary condition also. We select boundary condition as n>=2 (Because for n = 1, T(1) = c.1.lg(1) = 0 which is false according to the definition of T(n))

Example 1.16
Find the worse case complexity of Binary Search
T(n) = c + T(n/2)

- Guess: T(n) = O(lgn)
  - Induction goal: $T(n) \le d\ lgn$, for some d and $n \ge n0$
  - Induction hypothesis: $T(n/2) \le d\ lg(n/2)$
- Proof of induction goal:
  $T(n) = T(n/2) + c \le d\ lg(n/2) + c$
  $= d\ lgn - d + c \le d\ lgn$
  if: $-d + c \le 0,\ d \ge c$

Example 1.17
T(n) = T(n-1) + n

- Guess: T(n) = O(n2)
  - Induction goal: $T(n) \le c\ n2$, for some c and $n \ge n0$
  - Induction hypothesis: $T(n-1) \le c(n-1)2$ for all $k < n$
- Proof of induction goal:
  $T(n) = T(n-1) + n \le c\ (n-1)2 + n$
  $= cn2 - (2cn - c - n) \le cn2$
  if: $2cn - c - n \ge 0 \Leftrightarrow c \ge n/(2n-1) \Leftrightarrow c \ge 1/(2 - 1/n)$
  - For $n \ge 1 \Rightarrow 2 - 1/n \ge 1 \Rightarrow$ any $c \ge 1$ will work

Example 1.17
T(n) = 2T(n/2) + n
- Guess: T(n) = O(nlgn)
  - Induction goal: $T(n) \le cn\ lgn$, for some c and $n \ge n0$
  - Induction hypothesis: $T(n/2) \le cn/2\ lg(n/2)$
- Proof of induction goal:
  $T(n) = 2T(n/2) + n \le 2c\ (n/2)lg(n/2) + n$
  $= cn\ lgn - cn + n \le cn\ lgn$
  if: $- cn + n \le 0 \Rightarrow c \ge 1$

1.6.2 **Recursion tree Method**

- Main disadvantage of Substitution method is that it is always difficult to come up with a good guess
- Recursion tree method allows you make a good guess for the substitution method
- Allows to visualize the process of iterating the recurrence

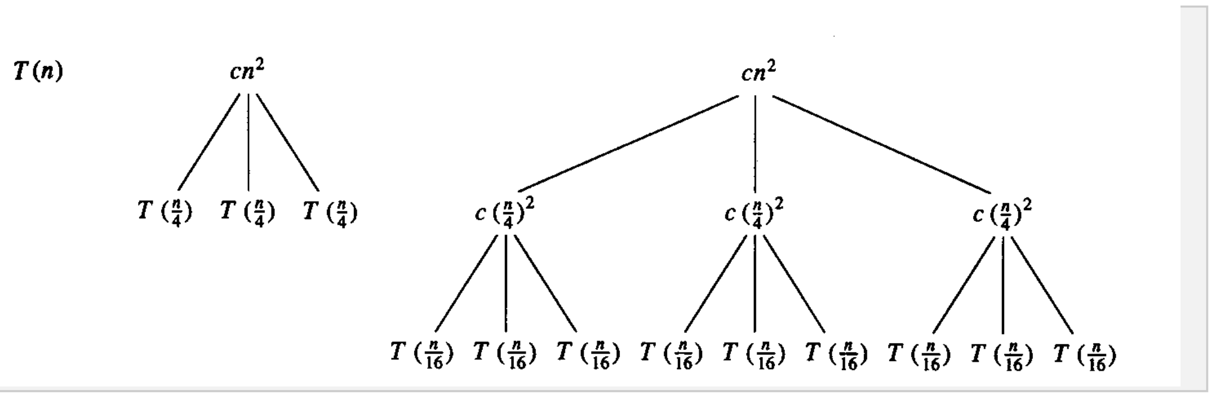
Steps

- Convert the recurrence into a tree.
- Each node represents the cost of a single sub problem somewhere in the set of recursive function invocations
- Sum the costs within each level of the tree to obtain a set of per-level costs
- Sum all the per-level costs to determine the total cost of all levels of the recursion

Example 1.18

$T(n) = 3T(n/4) + (n^2)$

$T(n) = 3T(n/4) + cn^2$


$T(n)$
cn^2
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
cn^2
$c(\frac{n}{4})^2$ $c(\frac{n}{4})^2$ $c(\frac{n}{4})^2$
$T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$

- The sub problem size for a node at depth i is n/4i
  - When the sub problem size is 1, n/4i = 1, $i=\log_4 n$
  - The tree has log4n+1 levels (0, 1, 2,.., $\log_4 n$)
- The cost at each level of the tree (0, 1, 2,.., $\log_4 n$-1)
  - Number of nodes at depth i is 3^i
  - Each node at depth i has a cost of $c(n/4i)^2$
  - The total cost over all nodes at depth i is $3^i\ c(n/4i)^2=(3/16)^i cn^2$
- The cost at depth $\log_4 n$
  - Number of nodes is $3^{\log_4 n} = n^{\log_4 3}$
  - Each contributing cost T(1)
  - The total cost $n^{\log_4 3}T(1) = \Theta(n^{\log_4 3})$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \ldots + (\frac{3}{16})^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3})$$

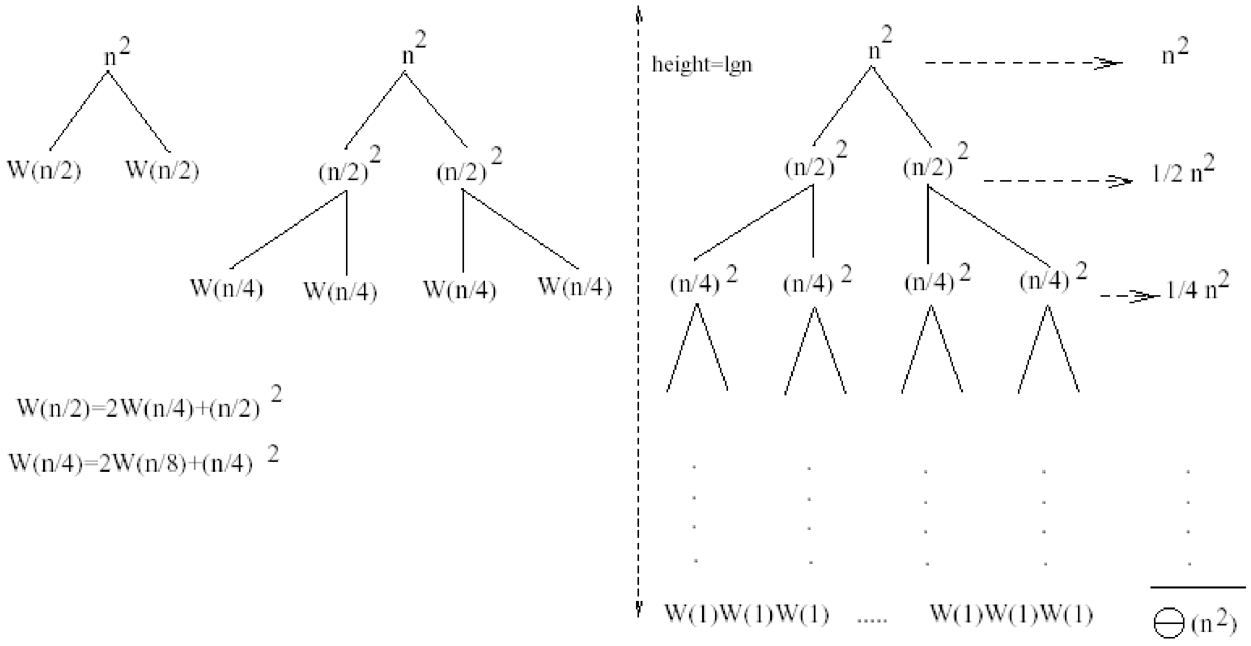
$$= \sum^{\log_4 n-1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

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

Prove T(n)=O(n2) is an upper bound by the substitution method
T(n) ≤ dn2 for some constant d > 0

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2 \end{aligned}$$

Example 1.19
W(n) = 2W(n/2) + (n^2)



- Subproblem size at level i is: n/2i
- Subproblem size hits 1 when 1 = n/2i ⇒ i = lgn
- Cost of the problem at level i = (n/2i)2 No. of nodes at level i = 2i

- Total cost:

$$W(n)=\sum_{i=0}^{\lg n-1}\frac{n^2}{2^i}+2^{\lg n}W(1)=n^2\sum_{i=0}^{\lg n-1}\left(\frac{1}{2}\right)^i+n\le n^2\sum_{i=0}^{\infty}\left(\frac{1}{2}\right)^i+O(n)=n^2\frac{1}{1-\frac{1}{2}}+O(n)=2n^2$$

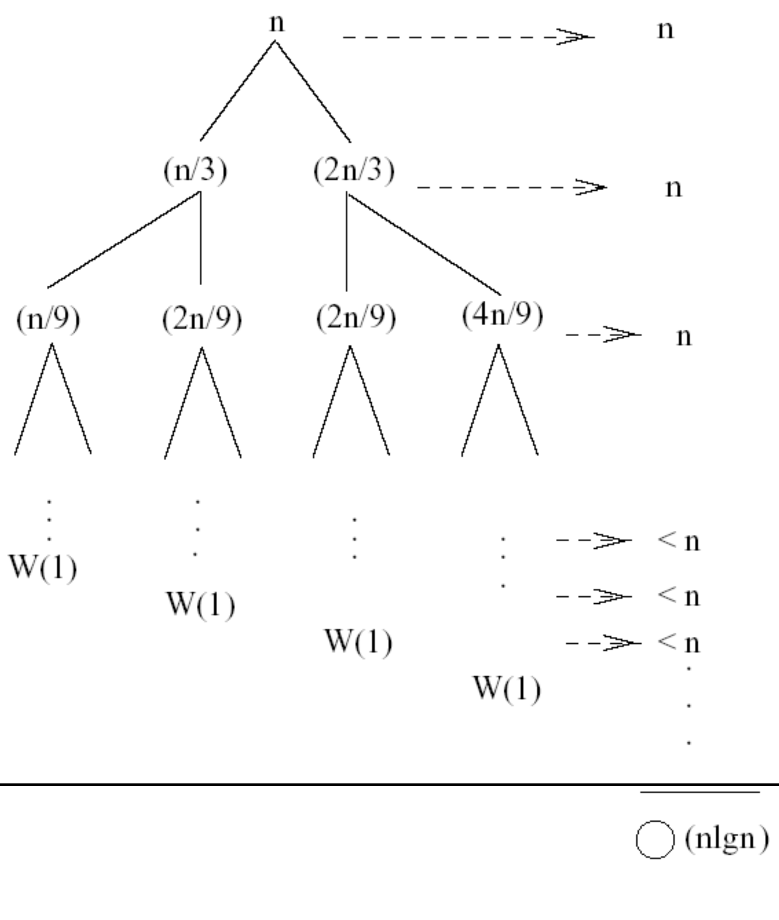
⇒ W(n) = O(n2)

Example 1.20
W(n) = W(n/3) + W(2n/3) + O(n)

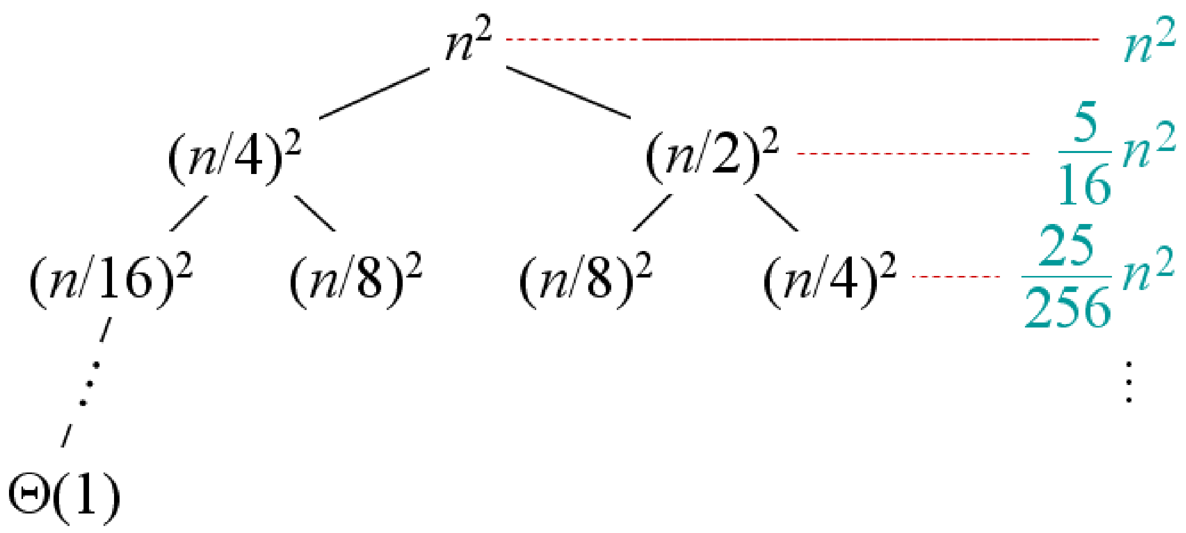
- The longest path from the root to a leaf is: n → (2/3)n → (2/3)2 n → … → 1
- Subproblem size hits 1 when 1 = (2/3)in ⇔ i=log3/2n
- Cost of the problem at level i = n
- Total cost:

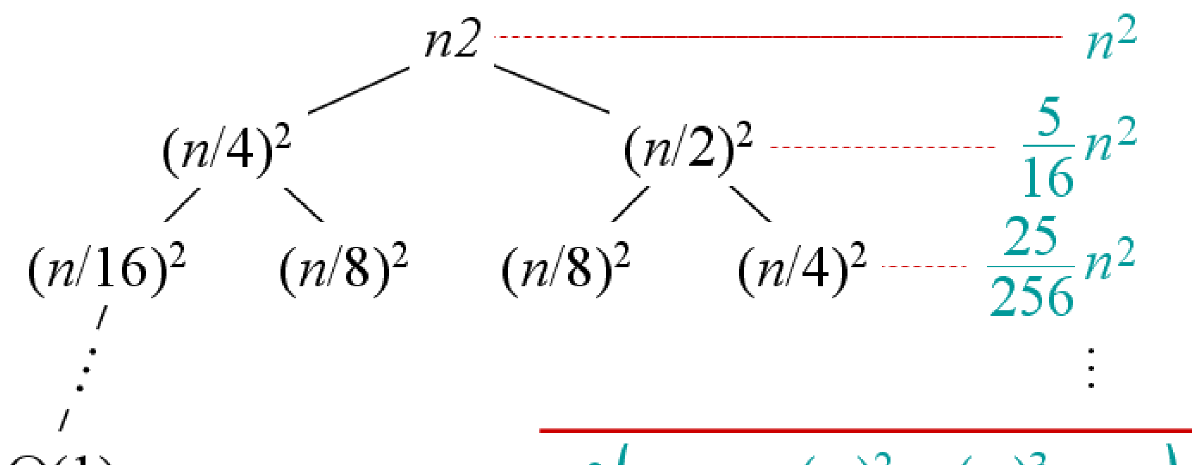
$$n+n+\ldots<\sum_{i=0}^{\log_{3/2}n}n=n\sum_{i=0}^{\log_{3/2}n}1=n\log_{3/2}n=n\frac{\lg n}{\lg 3/2}=\frac{1}{\lg 3/2}n\lg n$$

⇒ W(n) = O(nlgn)



Example 1.21
T(n) = T(n/4) + T(n/2) + n2

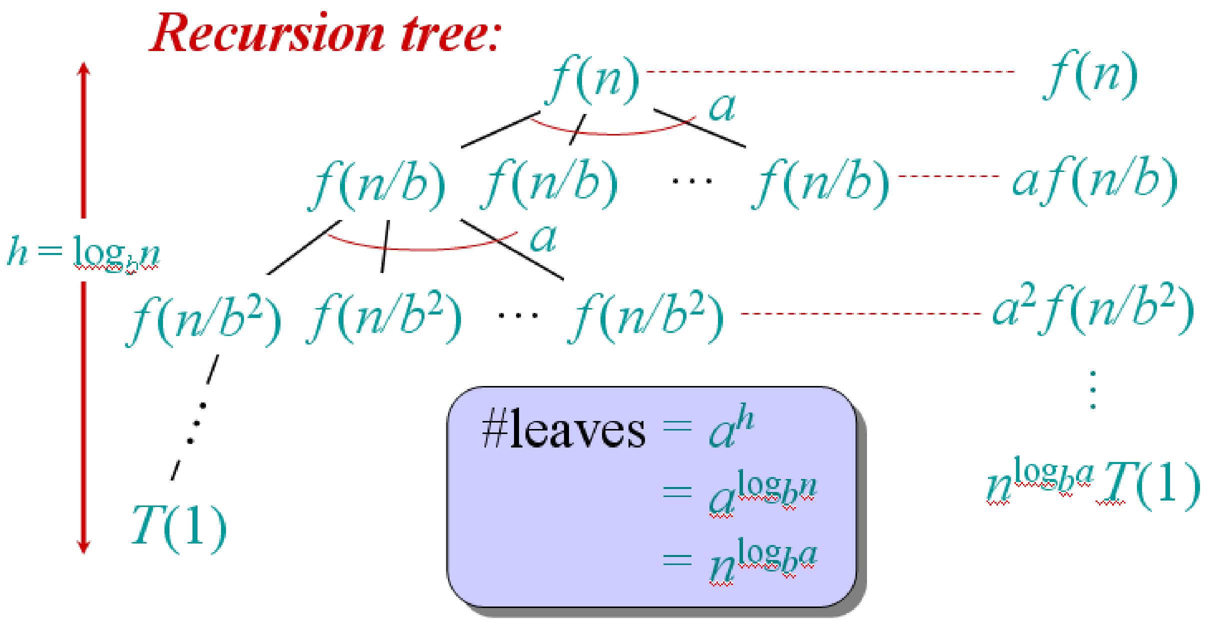




$$\text{Total} = n^2\left(1+\frac{5}{16}+\left(\frac{5}{16}\right)^2+\left(\frac{5}{16}\right)^3+\Lambda\right)$$
$$= \Theta(n^2)$$

1.6.3 **Master Method**

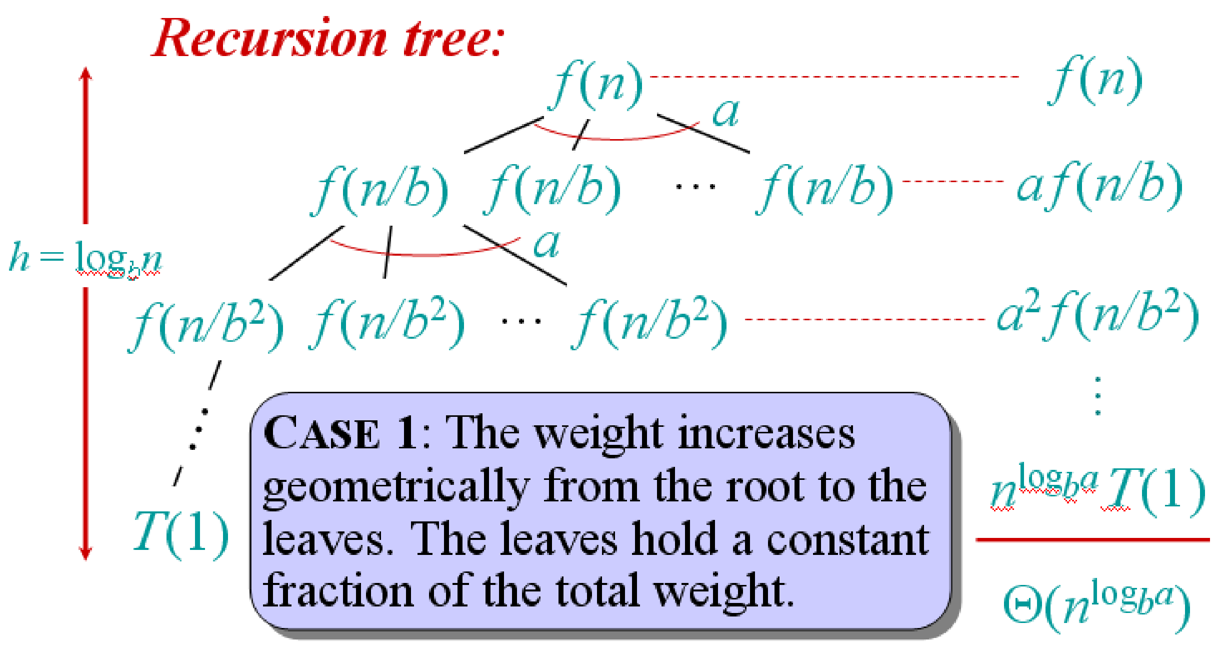
- The master method applies to recurrences of the form $T(n) = a\ T(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and f is asymptotically positive.
- Describe the running time of an algorithm that divides a problem of size n into a sub problems, each of size n/b



- There are three common cases

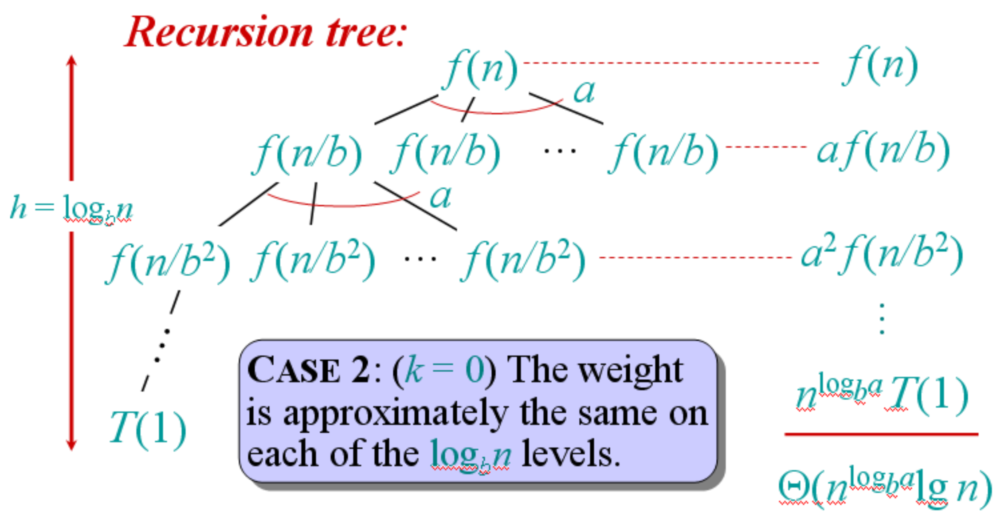
Case 1

- Compare $f(n)$ with $n^{\log ba}$
- If $f(n) = O(n^{\log ba - e})$ for some constant e > 0.
  i.e., $f(n)$ grows polynomially slower than $n^{\log ba}$
  i.e., *f(n)* is asymptotically smaller by an n^e factor.
  Then ***Solution:*** $T(n) = \Theta(n^{\log ba})$.


Recursion tree:
$f(n)$
a
$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$
$h = \log_b n$
$f(n/b^2)$ $f(n/b^2)$ $\cdots$ $f(n/b^2)$
$T(1)$
$f(n)$
$af(n/b)$
$a^2 f(n/b^2)$
$n^{\log_b a}T(1)$
$\Theta(n^{\log_b a})$
CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

Case 2

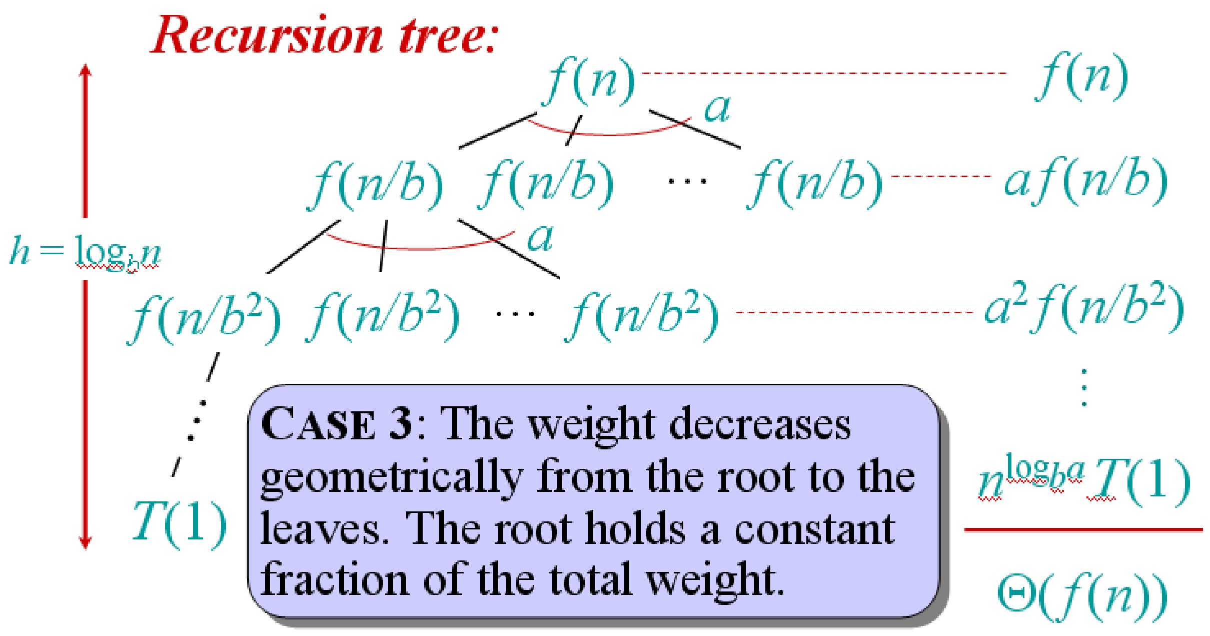
- Compare $f(n)$ with $n^{\log ba}$
- If $f(n) = \Theta(n^{\log ba})$
  i.e., $f(n)$ and $n^{\log ba}$ grow at similar rates.
  Then ***Solution:*** $T(n) = \Theta(n^{\log ba} \lg n)$.


Recursion tree:
$f(n)$
a
$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$
$h = \log_b n$
$f(n/b^2)$ $f(n/b^2)$ $\cdots$ $f(n/b^2)$
$T(1)$
$f(n)$
$af(n/b)$
$a^2 f(n/b^2)$
$n^{\log_b a}T(1)$
$\Theta(n^{\log_b a}\lg n)$
CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

Case 3

- Compare $f(n)$ with $n^{\log ba}$
- If $f(n) = \Omega(n^{\log ba+e})$ for some constant e > 0
  i.e., $f(n)$ grows polynomially faster than $n^{\log ba}$
  i.e., *f(n)* is asymptotically larger by an n^e factor.

Then ***Solution:*** $T(n) = \Theta(f(n))$.


Recursion tree:
f(n)
a
f(n/b) f(n/b) ··· f(n/b)
a
f(n/b²) f(n/b²) ··· f(n/b²)
T(1)
h = log_b n
f(n)
af(n/b)
a²f(n/b²)
n^{log_b a} T(1)
Θ(f(n))
CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

Example 1.22
T(n)=9T(n/3) + n
a=9, b=3, f(n) = n
CASE 1: $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2), f(n) = O(n^{\log_3 9-1})$
T(n) = θ(n2)

Example 1.23
$T(n) = 4T(n/2) + n$
$a = 4, b = 2 \Rightarrow nlogba = n2; f(n) = n.$
CASE 1: $f(n) = O(n^{2-e})$ for e = 1.
$\therefore\ T(n) = \Theta(n2).$

Example 1.24
T(n) = T(2n/3) + 1
a=1, b=3/2, f(n)=1
Case 2: $n^{\log_b a} = n^{\log_{3/2} 1} = 1, f(n) = 1 = \Theta(1)$
T(n) = θ(lg *n*)

Example 1.25
$T(n) = 4T(n/2) + n^2$
$a = 4, b = 2 \Rightarrow nlogba = n2; f(n) = n2.$
CASE 2: $f(n) = Q(n2lg0n)$, that is, $k = 0$.
$\therefore\ T(n) = \Theta(n^2 \lg n).$

Example 1.26
T(n) = 3T(n/4) + n lg n (Case 3)
a=3, b=4, f(n) = n lg n
Case3: $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$ $\quad f(n) = \Omega(n^{\log_4 3+\varepsilon})$ e = 0.2

For sufficiently large n, $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c=3/4$
$T(n) = \theta(n \lg n)$

Example 1.27
$T(n) = 4T(n/2) + n^3$
$a = 4, b = 2 \Rightarrow n^{\log ba} = n^2; f(n) = n^3.$
CASE 3: $f(n) = \Omega(n2 + e)$ for e = 1 *and* 4(*cn*/2)3 £ *cn*3 (reg. cond.) for $c = 1/2$.
$\therefore\ T(n) = \Theta(n^3)$.