

Chapter 1

Principles of Analyzing algorithms and Problems

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 i.e. algorithms run in same manner everywhere. So the algorithm is a mathematical object where the algorithms are assumed to be run under machine with unlimited capacity.

Examples of problems

- You are given two numbers, how do you find the Greatest Common Divisor.
- Given an array of numbers, how do you sort them?

We need algorithms to understand the basic concepts of the Computer Science, programming. Where the computations are done and 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).

You need designing concepts of the algorithms because if you only study the algorithms then you are bound to those algorithms and selection among the available algorithms. However if you have knowledge about design then you can attempt to improve the performance using different design principles.

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? The other issues of analysis are efficiency, optimality, 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. Both of the resources are measures in terms of complexity for time instead of absolute time we consider growth

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.

Here we deal with correctness and finiteness.

Expressing Algorithms

There are many ways of expressing algorithms; the order of ease of expression is natural language, pseudo code and real programming language syntax. In this course I inter mix the natural language and pseudo code convention.

Random Access Machine Model

This RAM model is the base model for our study of design and analysis of algorithms to have design and analysis in machine independent scenario. In this model each basic operations (+, -) takes 1 step, loops and subroutines are not basic operations. Each memory reference is 1 step. We measure run time of algorithm by counting the steps.

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 of the input(s).

In our study we concentrate on worst case complexity only.

Example 1: Fibonacci Numbers

Input: n

Output: n^{th} Fibonacci number.

Algorithm: assume a as first(previous) and b as second(current) numbers

```
fib(n)
{
    a = 0, b = 1, f = 1 ;
    for(i = 2 ; i <= n ; i++)
    {
        f = a+b ;
        a=b ;
        b=f ;
    }
    return f ;
}
```

Efficiency

Time Complexity: The algorithm above iterates up to $n-2$ times, so time complexity is $O(n)$.

Space Complexity: The space complexity is constant i.e. $O(1)$.

Example 2: Greatest Common Divisor

Inputs: Two numbers a and b

Output: G.C.D of a and b .

Algorithm: assume (for simplicity) $a > b \geq 0$

```
gcd(a,b)
{
    While(b != 0)
    {
        d = a/b ;
        temp = b ;
        b = a - b * d ;
        a = temp ;
    }
    return a ;
}
```

Efficiency

Running Time: if the given numbers a and b are of n -bits then loop executes for n time and the division and multiplication can be done in $O(n^2)$ time. So the total running time becomes $O(n^3)$.

Another way of analyzing:

For Simplicity Let us assume that

$$b = 2^n$$

$$\Rightarrow n = \log b$$

$$\Rightarrow \text{Loop executes } \log b \text{ times in worst case}$$

$$\Rightarrow \text{Time Complexity} = o(\log b)$$

Space Complexity: The only allocated spaces are for variables so space complexity is constant i.e. $O(1)$.

Mathematical Foundation

Since mathematics can provide clear view of an algorithm. Understanding the concepts of mathematics is aid in the design and analysis of good algorithms. Here we present some of the mathematical concepts that are helpful in our study.

Exponents

Some of the formulas that are helpful are:

$$x^a x^b = x^{a+b}$$

$$x^a / x^b = x^{a-b}$$

$$(x^a)^b = x^{ab}$$

$$x^n + x^n = 2x^n$$

$$2^n + 2^n = 2^{n+1}$$

Logarithmes

Some of the formulas that are helpful are:

$$1. \log_a b = \log_c b / \log_c a ; c > 0$$

$$2. \log ab = \log a + \log b$$

$$3. \log a/b = \log a - \log b$$

$$4. \log (a^b) = b \log a$$

$$5. \log x < x \text{ for all } x > 0$$

$$6. \log 1 = 0, \log 2 = 1, \log 1024 = 10.$$

$$7. {}_a \log b^n = n {}_a \log b^a$$

Series

$$1. \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

$$2. \sum_{i=0}^n a^i \leq 1 / 1-a ; \text{ if } 0 < a < 1$$

$$= a^{n+1} - 1 / a-1 ; \text{ else}$$

3. $\sum_{i=1}^n i = n(n+1) / 2$
4. $\sum_{i=0}^n i^2 = n(n+1)(2n+1) / 6$
5. $\sum_{i=0}^n i^k \approx n^{k+1} / (k+1) ; k \neq -1$
6. $\sum_{i=1}^n 1/i \approx \log_e n$

Asymptotic Notation

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. The figure below gives upper and lower bound concept.

Big Oh (O) notation

When we have only asymptotic upper bound then we use O notation. A function $f(x)=O(g(x))$ (read as f(x) is big oh of g(x)) iff there exists two positive constants c and x_0 such that for all $x \geq x_0$, $0 \leq f(x) \leq c \cdot g(x)$

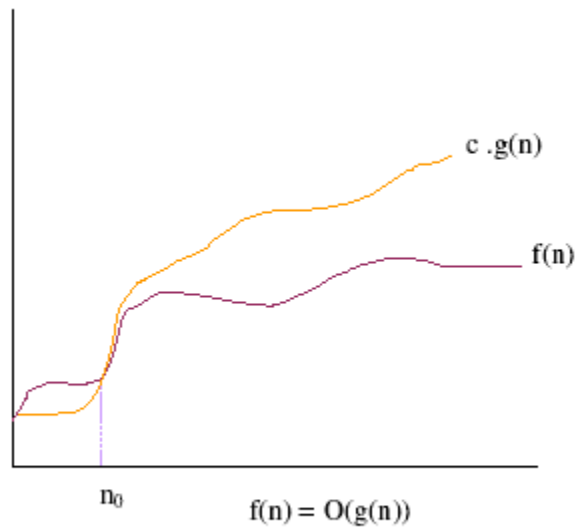
The above relation says that $g(x)$ is an upper bound of $f(x)$

Some properties:

Transitivity: $f(x) = O(g(x))$ & $g(x) = O(h(x)) \Rightarrow f(x) = O(h(x))$

Reflexivity: $f(x) = O(f(x))$

$O(1)$ is used to denote constants.



For all values of $n \geq n_0$, plot shows clearly that $f(n)$ lies below or on the curve of $c \cdot g(n)$

Examples

1. $f(n) = 3n^2 + 4n + 7$
 $g(n) = n^2$, then prove that $f(n) = O(g(n))$.

Proof: let us choose c and n_0 values as 14 and 1 respectively then we can have

$$f(n) \leq c \cdot g(n), n \geq n_0 \text{ as}$$

$$3n^2 + 4n + 7 \leq 14n^2 \text{ for all } n \geq 1$$

The above inequality is trivially true

$$\text{Hence } f(n) = O(g(n))$$

2. Prove that $n \log(n^3)$ is $O(\sqrt{n^3})$.

Proof: we have $n \log(n^3) = 3n \log n$

$$\text{Again, } \sqrt{n^3} = n \sqrt{n},$$

If we can prove $\log n = O(\sqrt{n})$ then problem is solved

Because $n \log n = n O(\sqrt{n})$ that gives the question again.

We can remember the fact that $\log^a n$ is $O(n^b)$ for all $a, b > 0$.

In our problem $a = 1$ and $b = 1/2$,

hence $\log n = O(\sqrt{n})$.

So by knowing $\log n = O(\sqrt{n})$ we proved that

$$n \log(n^3) = O(\sqrt{n^3}).$$

3. Is $2^{n+1} = O(2^n)$?

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

Big Omega (Ω) notation

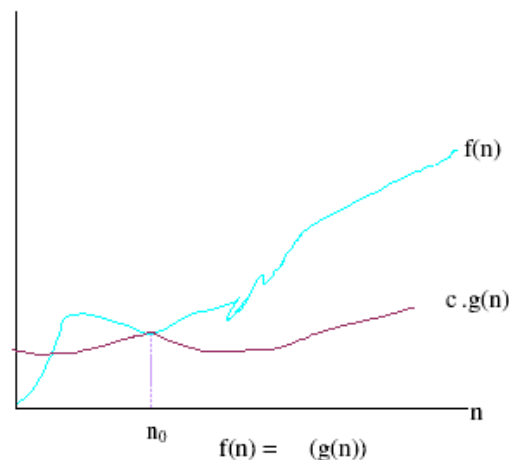
Big omega notation gives asymptotic lower bound. A function $f(x) = \Omega(g(x))$ (read as $g(x)$ is big omega of $f(x)$) iff there exists two positive constants c and x_0 such that for all $x \geq x_0$, $0 \leq c \cdot g(x) \leq f(x)$.

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

Some properties:

Transitivity: $f(x) = O(g(x))$ & $g(x) = O(h(x)) \Rightarrow f(x) = O(h(x))$

Reflexivity: $f(x) = O(f(x))$



For all values of $n \geq n_0$, plot shows clearly that $f(n)$ lies above or on the curve of $c \cdot g(n)$.

Examples

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

$g(n) = n^2$, then prove that $f(n) = \Omega(g(n))$.

Proof: let us choose c and n_0 values as 1 and 1, respectively then we can have

$$f(n) \geq c \cdot g(n), n \geq n_0 \text{ as}$$

$$3n^2 + 4n + 7 \geq 1 \cdot n^2 \text{ for all } n \geq 1$$

The above inequality is trivially true

$$\text{Hence } f(n) = \Omega(g(n))$$

Big Theta (Θ) notation

When we need asymptotically tight bound then we use notation. A function $f(x) = \Theta(g(x))$ (read as $f(x)$ is big theta of $g(x)$) iff there exists three positive constants c_1 , c_2 and x_0 such that for all $x \geq x_0$, $c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x)$

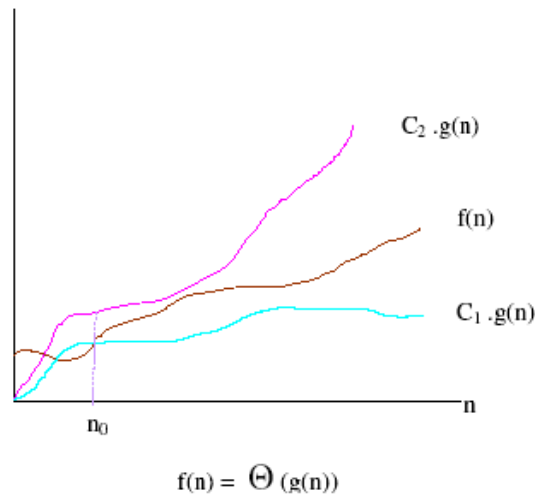
The above relation says that $f(x)$ is order of $g(x)$

Some properties:

Transitivity: $f(x) = \Theta(g(x))$ & $g(x) = \Theta(h(x)) \Rightarrow f(x) = \Theta(h(x))$

Reflexivity: $f(x) = \Theta(f(x))$

Symmetry: $f(x) = \Theta(g(x))$ iff $g(x) = \Theta(f(x))$



For all values of $n \geq n_0$, plot shows clearly that $f(n)$ lies between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$.

Examples

1. $f(n) = 3n^2 + 4n + 7$
 $g(n) = n^2$, then prove that $f(n) = \Theta(g(n))$.

Proof: let us choose c_1 , c_2 and n_0 values as 14, 1 and 1 respectively then we can have,

$$f(n) \leq c_1 \cdot g(n), n \geq n_0 \text{ as } 3n^2 + 4n + 7 \leq 14 \cdot n^2, \text{ and}$$

$$f(n) \geq c_2 * g(n), n \geq n_0 \text{ as } 3n^2 + 4n + 7 \geq 1 * n^2$$

for all $n \geq 1$ (in both cases).

So $c_2 * g(n) \leq f(n) \leq c_1 * g(n)$ is trivial.

Hence $f(n) = \Theta(g(n))$.

2. Show $(n + a)^b = \Theta(n^b)$, for any real constants a and b , where $b > 0$.

Here, using Binomial theorem for expanding $(n + a)^b$, we get ,

$$C(b,0)n^b + C(b,1)n^{b-1}a + \dots + C(b,b-1)na^{b-1} + C(b,b)a^b$$

We can obtain some constants such that $(n + a)^b \leq c_1 * (n^b)$, for all $n \geq n_0$

And $(n + a)^b \geq c_2 * (n^b)$, for all $n \geq n_0$

Here we may take $c_1 = 2^b$ $c_2 = 1$ $n_0 = |a|$,

Since $1 * (n^b) \leq (n + a)^b \leq 2^b * (n^b)$.

Hence the problem is solved.

Exercises

1. Show that 2^n is $O(n!)$.

2. $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$, where a_0, a_2, \dots, a_n are real numbers with $a_n \neq 0$.

Then show that $f(x)$ is $O(x^n)$, $f(x)$ is $\Omega(x^n)$ and then show $f(x)$ is order of x^n .

Recurrences

- Recursive algorithms are described by using recurrence relations.
- A recurrence is an inequality that describes a problem in terms of itself.

For Example:

Recursive algorithm for finding factorial

$$T(n) = 1 \quad \text{when } n = 1$$

$$T(n) = T(n-1) + O(1) \quad \text{when } n > 1$$

Recursive algorithm for finding Nth Fibonacci number

$$\begin{array}{ll}
 T(1)=1 & \text{when } n=1 \\
 T(2)=1 & \text{when } n=2 \\
 T(n)=T(n-1) + T(n-2) + O(1) & \text{when } n>2
 \end{array}$$

Recursive algorithm for binary search

$$\begin{array}{ll}
 T(1)=1 & \text{when } n=1 \\
 T(n)=T(n/2) + O(1) & \text{when } n>1
 \end{array}$$

Technicalities

Consider the recurrence

$$\begin{array}{ll}
 T(n) = k & n=1 \\
 T(n) = 2T(n/2) + kn & n>1
 \end{array}$$

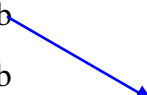
What is Odd about above? In next iteration n may not be integral.

More accurate is:

$$\begin{array}{ll}
 T(n) = k & n=1 \\
 T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + kn & n>1
 \end{array}$$

This difference rarely matters, so we usually ignore this detail

Again consider the recurrence

$$\begin{array}{ll}
 T(n) = \Theta(1) & n < b \\
 T(n) = a \cdot T(n/b) + f(n) & n \geq b
 \end{array}$$


For constant-sized problem, sizes can bound algorithm by some constant value.

This constant value is irrelevant for asymptote. Thus, we often skip writing the base case equation

Techniques for Solving Recurrences

We'll use four techniques:

- Iteration method
- Recursion Tree
- Substitution
- Master Method – for divide & conquer
- Characteristic Equation – for linear

Iteration method

- Expand the relation so that summation independent on n is obtained.
- Bound the summation

e.g.

$$\begin{aligned} T(n) &= 2T(n/2) + 1 && \text{when } n > 1 \\ T(n) &= 1 && \text{when } n = 1 \end{aligned}$$

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 2 \{ 2T(n/4) + 1 \} + 1 \\ &= 4T(n/4) + 2 + 1 \\ &= 4 \{ T(n/8) + 1 \} + 2 + 1 \\ &= 8T(n/8) + 4 + 2 + 1 \\ &\dots\dots\dots \\ &\dots\dots\dots \\ &= 2^k T(n/2^k) + 2^{k-1} T(n/2^{k-1}) + \dots\dots\dots + 4 + 2 + 1. \end{aligned}$$

For simplicity assume:

$$\begin{aligned} n &= 2^k \\ \Rightarrow k &= \log n \\ \Rightarrow T(n) &= 2^k + 2^{k-1} + \dots\dots\dots + 2^2 + 2^1 + 2^0 \\ \Rightarrow T(n) &= (2^{k+1} - 1) / (2 - 1) \\ \Rightarrow T(n) &= 2^{k+1} - 1 \end{aligned}$$

$$\Rightarrow T(n) = 2 \cdot 2^k - 1$$

$$\Rightarrow T(n) = 2n - 1$$

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

Second Example:

$$T(n) = T(n/3) + O(n) \quad \text{when } n > 1$$

$$T(n) = 1 \quad \text{when } n = 1$$

$$T(n) = T(n/3) + O(n)$$

$$\Rightarrow T(n) = T(n/3^2) + O(n/3) + O(n)$$

$$\Rightarrow T(n) = T(n/3^3) + O(n/3^2) + O(n/3) + O(n)$$

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

$$\Rightarrow T(n) = T(n/3^k) + O(n/3^{k-1}) + \dots + O(n/3) + O(n)$$

For Simplicity assume

$$n = 3^k$$

$$\Rightarrow k = \log_3 n$$

$$\Rightarrow T(n) \leq T(1) + c \cdot n/3^{k-1} + \dots + c \cdot n/3^2 + c \cdot n/3 + c \cdot n$$

$$\Rightarrow T(n) \leq 1 + \{ c \cdot n/3^{k-1} + \dots + c \cdot n/3^2 + c \cdot n/3 + c \cdot n \}$$

$$\Rightarrow T(n) \leq 1 + c \cdot n \{ 1/(1-1/3) \}$$

$$\Rightarrow T(n) \leq 1 + 3/2 \cdot c \cdot n$$

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

Substitution Method

Takes two steps:

1. Guess the form of the solution, using unknown constants.
2. Use induction to find the constants & verify the solution.

Completely dependent on making reasonable guesses

Consider the example:

$$T(n) = 1 \quad n=1$$

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

$$\text{Guess: } T(n) = O(n^3).$$

More specifically:

$$T(n) \leq cn^3, \text{ for all large enough } n.$$

Prove by strong induction on n .

Assume: $T(k) \leq ck^3$ for $\forall k < n$.

Show: $T(n) \leq cn^3$ for $\forall n > n_0$.

Base case,

For $n=1$:

$$T(n) = 1$$

Definition

$$1 \leq c$$

Choose large enough c for conclusion

Inductive case, $n > 1$:

$$T(n) = 4T(n/2) + n \quad \text{Definition.}$$

$$\leq 4c \cdot (n/2)^3 + n \quad \text{Induction.}$$

$$= c/2 \cdot n^3 + n \quad \text{Algebra.}$$

While this is $O(n^3)$, we're not done.

Need to show $c/2 \cdot n^3 + n \leq c \cdot n^3$.

Fortunately, the **constant factor** is shrinking, not growing.

$$T(n) \leq c/2 \cdot n^3 + n$$

From before.

$$= cn^3 - (c/2 \cdot n^3 - n)$$

Algebra.

$$\leq cn^3$$

Since $n > 0$, if $c \geq 2$

Proved:

$$T(n) \leq 2n^3 \text{ for } \forall n > 0$$

$$\text{Thus, } T(n) = O(n^3).$$

Second Example

$$T(n) = 1 \quad n=1$$

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

Guess: $T(n) = O(n^2)$.

Same recurrence, but now try tighter bound.

More specifically:

$$T(n) \leq cn^2 \text{ for } \forall n > n_0.$$

Assume $T(k) \leq ck^2$, for $\forall k < n$.

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c \cdot (n/2)^2 + n \\ &= cn^2 + n \end{aligned}$$

Not $\leq cn^2$!

Problem is that the constant isn't shrinking.

Solution: Use a tighter guess & inductive hypothesis.

Subtract a lower-order term – a common technique.

Guess:

$$T(n) \leq cn^2 - dn \text{ for } \forall n > 0$$

Assume $T(k) \leq ck^2 - dk$, for $\forall k < n$. Show $T(n) \leq cn^2 - dn$.

Base case, $n=1$

$T(n) = 1$ Definition.

$1 \leq c - d$ Choosing c, d appropriately.

Inductive case, $n > 1$:

$$\begin{aligned} T(n) &= 4T(n/2) + n && \text{Definition.} \\ &\leq 4(c(n/2)^2 - d(n/2)) + n && \text{Induction.} \\ &= cn^2 - 2dn + n && \text{Algebra.} \\ &= cn^2 - dn - (dn - n) && \text{Algebra.} \\ &\leq cn^2 - dn && \text{Choosing } d \geq 1. \\ T(n) &\leq 2n^2 - dn \text{ for } \forall n > 0 \\ \text{Thus, } T(n) &= O(n^2). \end{aligned}$$

Ability to guess effectively comes with experience.

Changing Variables:

Sometimes a little algebraic manipulation can make a unknown recurrence similar to one we have seen

Consider the example

$$T(n) = 2T(\lfloor n^{1/2} \rfloor) + \log n$$

Looks Difficult: Rearrange like

$$\text{Let } m = \log n \Rightarrow n = 2^m$$

Thus,

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

Again let

$$S(m) = T(2^m) \quad S(m) = 2S(m/2) + m$$

We can show that

$$S(m) = O(\log m)$$

$$\Rightarrow T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$$

Recursion Tree

Jus Simplification of Iteration method:

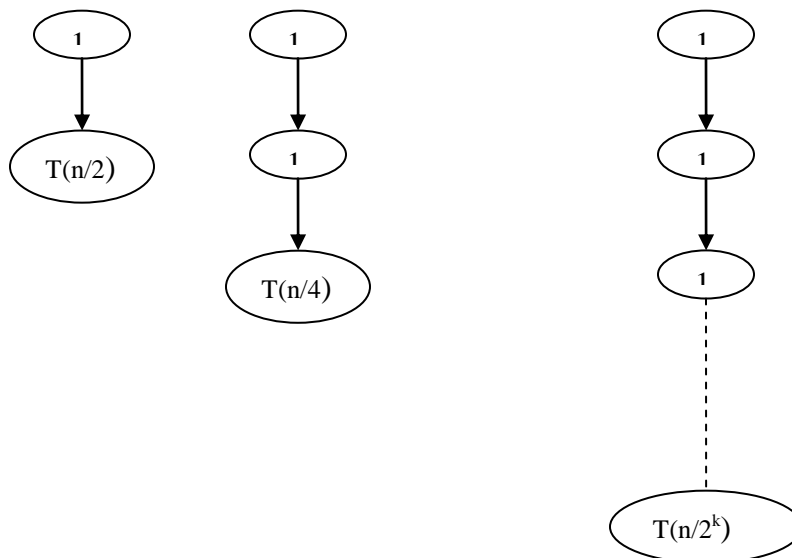
Consider the recurrence

$$T(1) = 1$$

when $n=1$

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

when $n > 1$



Summing the cost at each level,

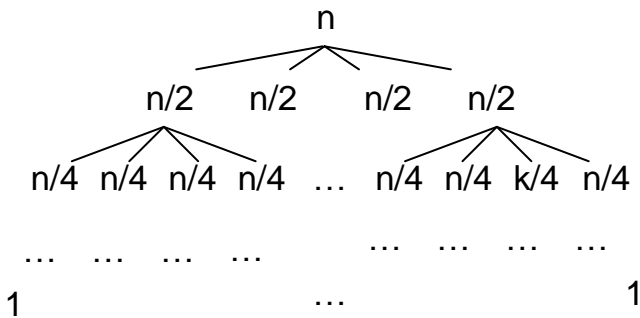
Total cost = $1 + 1 + 1 + \dots$ Up to $\log n$ terms

\Rightarrow complexity = $O(\log n)$

Second Example

$$T(n) = 1 \quad n=1$$

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

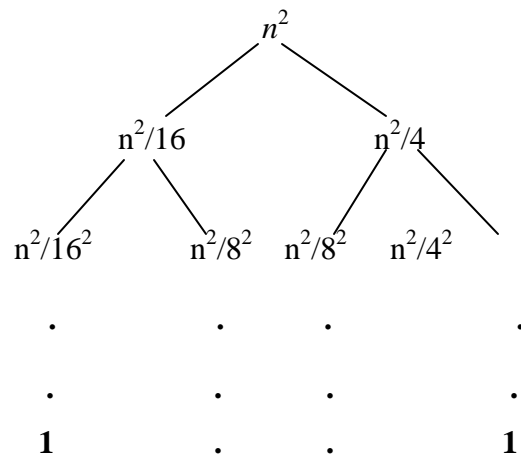
Third Example $T(n)$	Cost at this level
n	n
	$2n$
$n/4$ $n/4$ $n/4$ $n/4$... $n/4$ $n/4$ $n/4$ $n/4$	2^2n
...	...
1 ... 1	$2^k n$

Assume: $n = 2^k$

$\Rightarrow k = \log n$

$$\begin{aligned}
 T(n) &= n + 2n + 4n + \dots + 2^{k-1}n + 2^k n \\
 &= n(1 + 2 + 4 + \dots + 2^{k-1} + 2^k) \\
 &= n(2^{k+1} - 1)/(2 - 1) \\
 &= n(2^{k+1} - 1) \\
 &\leq n 2^{k+1} \\
 &= 2n 2^k \\
 &= 2n \cdot n \\
 &= O(n^2)
 \end{aligned}$$

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



$$\text{Total Cost} \leq n^2 + 5 n^2/16 + 5^2 n^2/16^2 + 5^3 n^2/16^3 + \dots + 5^k n^2/16^k$$

{ why \leq ? Why not $=$? }

$$= n^2 (1 + 5/16 + 5^2/16^2 + 5^3/16^3 + \dots + 5^k/16^k)$$

$$= n^2 + (1 - 5^{k+1}/16^{k+1})$$

$$= n^2 + \text{constant}$$

$$= O(n^2)$$

Master Method

Cookbook solution for some recurrences of the form

$$T(n) = a \cdot T(n/b) + f(n)$$

where

$a \geq 1$, $b > 1$, $f(n)$ asymptotically positive

Describe its cases

Master Method Case 1

$$T(n) = a \cdot T(n/b) + f(n)$$

$$f(n) = O(n^{\log_b a - \epsilon}) \text{ for some } \epsilon > 0 \rightarrow T(n) = \Theta(n^{\log_b a})$$

$$T(n) = 7T(n/2) + cn^2 \quad a=7, b=2$$

$$\text{Here } f(n) = cn^2 \quad n^{\log_b a} = n^{\log_2 7} = n^{2.8}$$

$$\Rightarrow cn^2 = O(n^{\log_b a - \epsilon}), \text{ for any } \epsilon \leq 0.8.$$

$$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8})$$

Master Method Case 2

$$T(n) = a \cdot T(n/b) + f(n)$$

$$f(n) = \Theta(n^{\log_b a}) \rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$$

$$T(n) = 2T(n/2) + cn \quad a=2, b=2$$

$$\text{Here } f(n) = cn \quad n^{\log_b a} = n$$

$$\Rightarrow f(n) = \Theta(n^{\log_b a})$$

$$T(n) = \Theta(n \lg n)$$

Master Method Case 3

$$T(n) = a \cdot T(n/b) + f(n)$$

$$f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ for some } \epsilon > 0 \quad \text{and} \\ a \cdot f(n/b) \leq c \cdot f(n) \text{ for some } c < 1 \text{ and all large enough } n \\ \rightarrow T(n) = \Theta(f(n))$$

I.e., is the constant factor shrinking?

$$T(n) = 4 \cdot T(n/2) + n^3 \quad a=4, b=2$$

$$n^3 = ? \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{\log_2 4 + \epsilon}) = \Omega(n^{2 + \epsilon}) \text{ for any } \epsilon \leq 1.$$

$$\text{Again, } 4(n/2)^3 = \frac{1}{2} \cdot n^3 \leq cn^3, \text{ for any } c \geq \frac{1}{2}.$$

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

Master Method Case 4

$T(n) = a \cdot T(n/b) + f(n)$
 None of previous apply. Master method doesn't help.

$$T(n) = 4T(n/2) + n^2/\lg n \quad a=4, b=2$$

$$\text{Case 1? } n^2/\lg n = O(n^{\log_b a - \epsilon}) = O(n^{\log_2 4 - \epsilon}) = O(n^{2 - \epsilon}) = O(n^2/n^\epsilon)$$

No, since $\lg n$ is asymptotically less than n^ϵ .
 Thus, $n^2/\lg n$ is asymptotically greater than n^2/n^ϵ .

$$\text{Case 2? } n^2/\lg n = ? \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$$

No.

$$\text{Case 3? } n^2/\lg n = ? \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{\log_2 4 + \epsilon}) = \Omega(n^{2 + \epsilon})$$

No, since $1/\lg n$ is asymptotically less than n^ϵ .

Exercises

- Show that the solution of $T(n) = 2T(n/2) + n$ is $\Omega(n \log n)$. Conclude that solution is $\Theta(n \log n)$.
- Show that the solution to $T(n) = 2T(n/2) + n$ is $O(n \log n)$.
- Write recursive Fibonacci number algorithm derive recurrence relation for it and solve by substitution method. {Guess 2^n }
- Argue that the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + n$ is $(n \log n)$ by appealing to a recursion tree.
- Use iteration to solve the recurrence $T(n) = T(n-a) + T(a) + n$, where $a \geq 1$ is a constant.
- Use the master method to give tight asymptotic bounds for the following recurrences.
 - ✓ $T(n) = 9T(n/3) + n$
 - ✓ $T(n) = 3T(n/4) + n \log n$

- ✓ $T(n) = 2T(2n/3) + 1$
 - ✓ $T(n) = 2T(n/2) + n \log n$
 - ✓ $T(n) = 2T(n/4) + \sqrt{n}$
 - ✓ $T(n) = T(\sqrt{n}) + 1$
- The running time of an algorithm A is described by the recurrence $T(n) = 7T(n/2) + n^2$. A competing algorithm A' has a running time of $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for 'a' such that A' is asymptotically faster than A?