CSC2204 ANALYSIS OF ALGORITHMS

2022/2023

Course Lectures
Dr. Muhammad Yusuf
Baffa Sani
Farouk Yushau
Bilal

Contents

1 Introduction to Design and analysis of algorithms

The Basic objective of solving problem with multiple constraints such as problem size performance and cost in terms of space and time. The goal is to design fast, efficient and effective solution to a problem domain. Some problems are easy to solve and some are hard. Quite cleverness is required to design solution with fast and better approach. Designing new system need a new technology and background of the new technology is the enhancement of existing algorithm. The study of algorithm is to design efficient algorithm not only limited in reducing cost and time but to enhance scalability, reliability and availability.

What is an algorithm?

- Algorithm is a set of steps to complete a task.

Example of algorithm

Task: ATM withdrawal system.

Algorithm:

1. Insert the card
2. Enter the PIN
3. Enter the amount
4. Withdraw amount
5. Check balance
6. Cancel/clear

Task: To make a cup of tea..

Algorithm:

1. Add water to the kettle
2. Boil it
3. Add tea leaves
4. Add Milk
5. Add sugar

What is Computer algorithm?

"a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it".

Described precisely: very difficult for a machine to know how much water, milk to be added etc. in the above tea making algorithm.

These algorithms run on computers or computational devices.For example, GPS in our smartphones, Google hangouts.

GPS uses shortest path algorithm. Online shopping uses cryptography which uses RSA algorithm.

Characteristics of an algorithm

- Must take an input.
- Must give some output (true/false, value etc.)
- Definiteness - each instruction is clear and unambiguous.
- Finiteness - algorithm terminates after a finite number of steps.
- Effectiveness - every instruction must be basic i.e. simple instruction.

Expectation from an algorithm

- Correctness:-
  - Correct: Algorithms must produce correct result.
  - Produces an incorrect answer:Even if it fails to give correct results all the time still there is a control on how often it gives wrong result. Eg. Rabin-Miller *Primality Test* (Used in RSA algorithm): It doesn't give correct answer all the time. 1 out of 2^{50} times it gives incorrect result.
  - Approximation algorithm: Exact solution is not found, but near optimal solutions can be obtained. (Applied to optimization problem.)
- Less resource usage:
  - Algorithms should use less resources (time and space).

Resource usage:

Here, the time is considered to be the primary measure of efficiency. We are also concerned with how much the respective algorithm involves the computer memory. But mostly time is the resource that is dealt with. And the actual running time depends on a variety of backgrounds: like the speed of a Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.

So, mainly the resource usage can be divided into:

1. Memory (space)
2. Time

Time taken by an algorithm

performance measurement or Apostoriori Analysis: Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.

Performance Evaluation or Apriori Analysis. Before implementing the algorithm in a system. This is done as follows

1. How long the algorithm takes :- will be represented as a function of the size of the input.
   $f(n) \rightarrow$ how long it takes if n is the size of input.
2. How fast the function that characterizes the running time grows with the input size.
   "Running time growth rate".
   The algorithm with less running time growth rate is considered better.

How algorithm is a technology?

Algorithms are just like a technology. We all use latest and greatest processors but we need to run implementations of good algorithms on that computer in order to properly take benefits of our money that we spent to have the latest processor. Let's make this example more concrete by pitting a faster computer (computer A) running a sorting algorithm whose running time on n values grows like n^2 against a slower computer (computer B) running a sorting algorithm whose running time grows like $n \log n$. They each must sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more

dramatic, suppose that the world's craftiest programmer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer writes for computer B, using a high level language with an inefficient compiler, with the resulting code taking $50n \log n$ instructions.

Computer A (Faster)	**Computer B(Slower)**
Running time grows like n^2.	Grows in $n \log n$.
10 billion instructions per sec.	10 million instruction per sec
$2n^2$ instruction.	50 $n \log n$ instruction.

Computer A takes $= \dfrac{2 \times (10^7)^2}{10^{10}} = 20{,}000$ seconds (more than 5:5 hours)

Computer B takes $= \dfrac{50 \times \log 10^7}{10^7} \approx 1163$ seconds (under 20 minutes)

So choosing a good algorithm (algorithm with slower rate of growth) as used by computer B affects a lot.

2 Growth of Functions (Asymptotic notations)

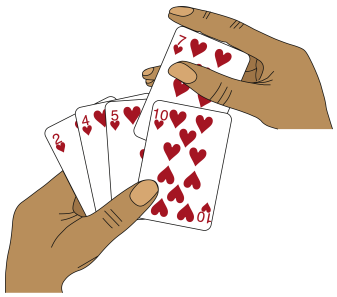
Before going for growth of functions and asymptotic notation let us see how to analyse an algorithm.

2.1 How to analyse an Algorithm

Insertion sort

A good algorithm for sorting a small number of elements.
It works the way you might sort a hand of playing cards:

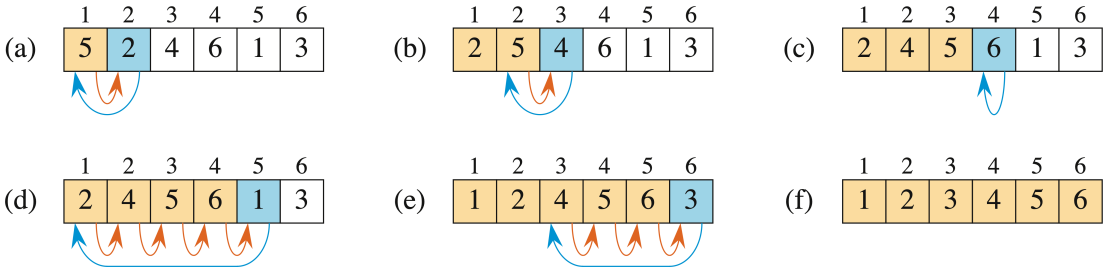
- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.



Let us form an algorithm for *Insertion sort* (which sort a sequence of numbers).

	INSERTION-SORT(A, n)	*cost*	*times*
1	**for** $i = 2$ **to** n	c_1	n
2	$key = A[i]$	c_2	$n-1$
3	// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4	$j = i-1$	c_4	$n-1$
5	**while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6	$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8	$A[j+1] = key$	c_8	$n-1$

Example



Analyzing algorithms

We want to predict the resources that the algorithm requires. Usually, running time.

Why analyze?

Why not just code up the algorithm, run the code, and time it?
Because that would tell you how long the code takes to run

- on your particular computer,
- on that particular input,
- with your particular implementation,
- using your particular compiler or interpreter,
- with the particular libraries linked in,
- with the particular background tasks running at the time.

You wouldn't be able to predict how long the code would take on a different computer, with a different input, if implemented in a different programming language, etc.
Instead, devise a formula that characterizes the running time.

Random-access machine (RAM) model

In order to predict resource requirements, we need a computational model.

- Instructions are executed one after another. No concurrent operations.
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll use instructions commonly found in real computers:

- Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling). Also, shift left/shift right/shift.
- Data movement: load, store, copy.
- Control: conditional/unconditional branch, subroutine call and return.

Each of these instructions takes a constant amount of time. Ignore memory hierarchy (cache and virtual memory).

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

Input size

Depends on the problem being studied.

- Usually, the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Running time

On a particular input, it is the number of primitive operations (steps) executed.

- Want to define steps to be machine-independent.
- Figure that each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line k takes the same amount of time C_k.

- This is assuming that the line consists only of primitive operations.
  - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
  - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: "sort the points by x-coordinate."

Analysis of insertion sort

- Assume that the k_{th} line takes time c_k, which is a constant. (Since the third line is a comment, it takes no time.)
- For $i = 1, 2, \ldots n$, let t_i be the number of times that the while loop test is executed for that value of i.
- Note that when a for or while loop exits in the usual way - due to the test in the loop header - the test is executed one time more than the loop body.

The running time of the algorithm is

$$\sum_{\text{all statements}} \text{(cost of statement).(number of times statement is executed)}$$

Let $T(n)$ = running time of INSERTION-SORT.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)$$

The running time depends on the values of t_i . These vary according to the input.

Best case

The array is already sorted. All t_i are 1.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8) \\ &= an + b \text{ for constants } a \text{ and } b \text{ (that depend on the statement costs } c_k) \end{aligned}$$

Worst case

The array is in reverse sorted order. We have to compare key with all elements to the left of the i_{th} position $\implies$ compare with $i-1$ elements. Since the while loop exits because i reaches 0, there's

one additional test after the $i-1$ tests $\implies t_j = i$. Running time is

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8) \\
&= an^2 + bn + c \text{ for constants } a, b, c \text{ (that again depend on statement costs } c_k)
\end{aligned}$$

We usually concentrate on finding the worst-case running time: the longest running time for any input of size n.

Why we concentrate on worst-case running time?

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it's often about as bad as the worst case.

Order of growth
Another abstraction to ease analysis and focus on the important features.
Look only at the leading term of the formula for running time.

- Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

Example: For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2bn + c$.
Drop lower-order terms $\implies an^2$.
Ignore constant coefficient $\implies n^2$.
But we cannot say that the worst-case running time $T(n)$ *equals* n^2.
It *grows like* n^2. But it doesn't *equal* n^2.
We say that the running time is Θn^2 to capture the notion that the order of *growth* is n^2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

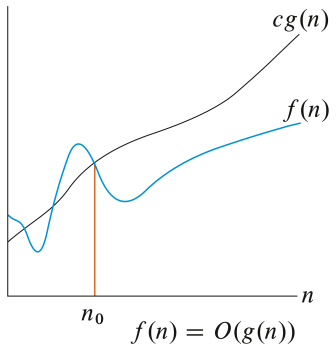
2.2 Asymptotic Notations

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:
  1. $O \approx \leq$
  2. $\Omega \approx \geq$
  3. $\Theta \approx =$
  4. $o \approx <$
  5. $\omega \approx >$

O-notation

O-notation characterizes an upper bound on the asymptotic behavior of a function: it says that a function grows no faster than a certain rate. This rate is based on the highest order term.

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



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.
If $f(n) \in Og(g(n))$, we write $f(n) = O(g(n))$

Example

$2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Example

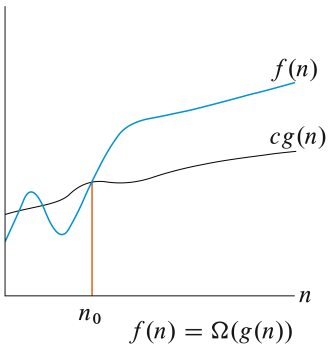
$f(n) = 7n^3 + 100n^2 - 20n + 6$ is $O(n^3)$, since the highest order term is $7n^3$, and therefore the function

grows no faster than n^3.

Ω-*notation*

Ω-notation characterizes a lower bound on the asymptotic behavior of a function: it says that a function grows at least as fast as a certain rate. This rate is again based on the highest-order term.

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



$g(n)$ is an **asymptotic lower bound** for $f(n)$

Example

$n^2 + n = \Omega(n^2)$

Example

$\sqrt{n} = \Omega(\log n)$, with $c = 1$ and $n_0 = 16$.

Example

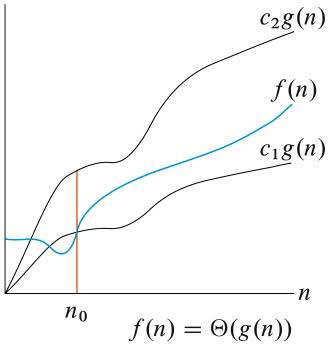
$f(n) = 7n^3 + 100n^2 - 20n + 6$ is $\Omega(n^3)$, since the highest order term is $7n^3$, and therefore the function grows at least as fast as n^3.

Θ-*notation*

Θ-notation characterizes a tight bound on the asymptotic behavior of a function: it says that a function grows precisely at a certain rate, again based on the highest order term.

If a function is both $O(f(n))$ and $\Omega(f(n))$, then a function is $\Theta(f(n))$.

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



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$

Example
$\frac{n^2}{2} - 2n = \Theta(n^2)$, with $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$, and $n_0 = 8$.

Theorem 2.1. $f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$

ø-notation
$o(g(n)) = \{f(n)$: for all constants $c > 0$, there exists a constant $n_{0>0}$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Another view, probably easier to use: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

Example
$n^{1.9999} = o(n^2)$

$\frac{n^2}{\log n} = o(n^2)$

$n^2 \neq o(n^2)$

$\frac{n^2}{1000} \neq o(n^2)$

ω-notation
$\omega = \{f(n)$: for all constants $c > 0$, there exists a constant $n_0 > n$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

A much easier definition to use: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.

Example
$n^{2.0001} = \omega(n^2)$

$n^2 \log n = \omega(n^2)$

$n^2 \neq \omega(n^2)$

Exercise 2.1. .

1. Suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64n \log n$ steps. For which values of n does insertion sort beat merge sort?

2. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

3 Recurrences, Solution of Recurrences by substitution, Recursion Tree and Master Method

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a **recurrence equation** which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.
For example, the worst case running time $T(n)$ of the merge sort procedure by recurrence can be expressed as

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

whose solution can be found as $T(n) = \Theta(n \log n)$

There are various techniques to solve recurrences.

1. **SUBSTITUTION METHOD:**
   The substitution method comprises of 3 steps

   i. Guess the form of the solution

   ii. Verify by induction

   iii. Solve for constants

   We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name "substitution method". This method is powerful, but we must be able to guess the form of the answer in order to apply it.

   Example
   Recurrence: $T(n) = 4T\left(\frac{n}{2}\right) + n$ and T

Step 1: Guess the solution

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ T(n) &= 4T\left(\frac{n}{2}\right) \\ \implies T(2n) &= 4T(n) \\ \implies T(n) &= n^2 \end{aligned}$$

So, $T(n)$ is order of n^2

We guess that the solution is $T(n) = O(n^3)$

Step 2: So we must prove that $T(n) \leq cn^3$ for some constant c.
Assume $T(k) \leq ck^3$

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ &\leq 4c\left(\frac{n}{2}\right)^3 + n \\ &\leq c\frac{n^3}{2} + n \\ &\leq cn^3 - \left(\frac{cn^3}{2} - n\right) \\ T(n) &\leq cn^3 \text{ as } \left(\frac{cn^3}{2} - n\right) \text{ is always positive} \end{aligned}$$

Therefore, we have shown that $T(n) \leq cn^3$, so $T(n) = O(n^3)$.

Step 3: To solve for constants

$$\begin{aligned} \frac{cn^3}{2} - n &\geq 0 \\ \implies n &\geq 1 \\ \implies c &\geq 2 \end{aligned}$$

Now suppose we guess that $T(n) = O(n^2)$ which is tight upper bound

Assume, $T(k) \leq ck^2$

so, we should prove that $T(n) \leq cn^2$

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ &= 4c\left(\frac{n}{2}\right)^2 + n \\ &= cn^2 + n \end{aligned}$$

So, $T(n)$ will never be less than cn^2. But if we will take the assumption of $T(k) = c_1k^2 - c_2k$, then we can find that $T(n) = O(n^2)$

2. **BY ITERATIVE METHOD:**
   Example
   Recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + n$

$$\begin{aligned} T(n) &= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n \\ &= 2^2T\left(\frac{n}{4}\right) + n + n \\ &= 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n \\ &= 2^3T\left(\frac{n}{2^3}\right) + 3n \end{aligned}$$

$$\vdots$$

$$T(n) = 2^kT\left(\frac{n}{2^k}\right) + kn \text{ After k iterations.}$$

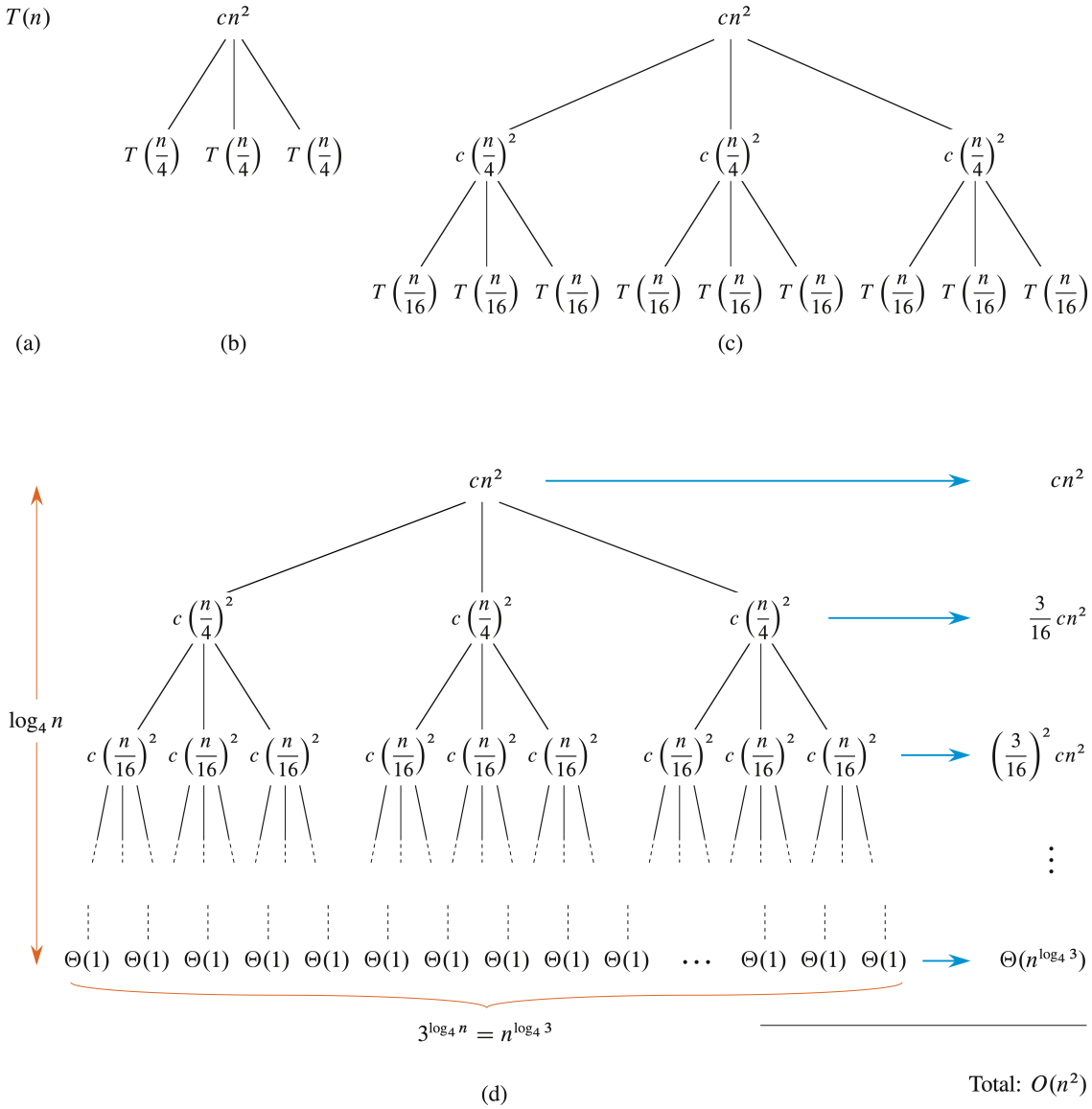
   Sub problem size is 1 after $\frac{n}{2^k} = 1 \implies k = \log n$
   So, after $k = \log n$ iterations, the sub-problem size will be 1.
   Substituting $k = \log n$ in the equation above, we obtained
   $T(n) = nT(1) + n\log n$
   $\implies = nc + n\log n$ where $c = T(1)$
   $\implies = O(n\log n)$

3. **BY RECURSION TREE METHOD:**
   In a recursion tree, each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations we sum the cost within each level of the tree to obtain a

set of per level cost, and then we sum all the per level cost to determine the total cost of all levels of recursion.

Example

Constructing a recursion tree for the recurrence $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$



4 Recursion tree method

4.1 Master Method

4.2 Worst case analysis of merge sort, quick sort and binary search