Part I: Time Complexity, Recurrence Relations

Course: Design and Analysis of Algorithms
by Dr. Partha Basuchowdhuri

Department of Computer Science and Engineering, Heritage Institute of Technology, Kolkata, India
Phone:(+91)9163883328, E-mail: parthabasu.chowdhuri@heritageit.edu

March 4, 2017

Outline for Part I

1. Time Complexity
   - Introduction
   - Asymptotics
   - Efficiency Classes
2. Recurrence Relations
   - Introduction
   - Solving Recurrences using Substitution
   - Solving Recurrences using Recurrence Tree
3. Master Theorem
   - Introduction
   - Drawbacks
   - Examples

Understanding Algorithms

Understanding Algorithms

A program can be divided into two parts that can contribute to its efficiency (or inefficiency) -

Understanding Algorithms

A program can be divided into two parts that can contribute to its efficiency (or inefficiency) -

- **Data Structure:** Storage and manipulation of the data inside the program should be efficient.

Understanding Algorithms

A program can be divided into two parts that can contribute to its efficiency (or inefficiency) -

- **Data Structure:** Storage and manipulation of the data inside the program should be efficient.
- **Algorithm:** There can be many solutions to a problem. Naive solutions are less efficient than more sophisticated solutions.

Understanding Algorithms

A program can be divided into two parts that can contribute to its efficiency (or inefficiency) -

- **Data Structure:** Storage and manipulation of the data inside the program should be efficient.
- **Algorithm:** There can be many solutions to a problem. Naive solutions are less efficient than more sophisticated solutions.

Understanding Algorithms

A program can be divided into two parts that can contribute to its efficiency (or inefficiency) -

- **Data Structure:** Storage and manipulation of the data inside the program should be efficient.
- **Algorithm:** There can be many solutions to a problem. Naive solutions are less efficient than more sophisticated solutions.

The effect is ANDed, i.e., if either of those parts is inefficient, then the algorithm becomes inefficient.

Understanding Algorithms

A program can be divided into two parts that can contribute to its efficiency (or inefficiency) -

- **Data Structure:** Storage and manipulation of the data inside the program should be efficient.
- **Algorithm:** There can be many solutions to a problem. Naive solutions are less efficient than more sophisticated solutions.

The effect is ANDed, i.e., if either of those parts is inefficient, then the algorithm becomes inefficient.

Goal of the Course: Given a problem, how to select a class of solutions and data structures best suited for solving the class of problems.
NB: Evidently, it needs a deep understanding of the nature of the problem.

How to estimate running time of your program/algorithm?

A program is divided into several modules. The cost function of a program or an algorithm can be determined by line-by-line accumulation of the time taken to perform the operations, expressed in terms of the input size n.

How to estimate running time of your program/algorithm?

A program is divided into several modules. The cost function of a program or an algorithm can be determined by line-by-line accumulation of the time taken to perform the operations, expressed in terms of the input size n.

For example -

- A for loop that calculates sum of first n +ve integers, leads to a component that has a running time of n.

How to estimate running time of your program/algorithm?

A program is divided into several modules. The cost function of a program or an algorithm can be determined by line-by-line accumulation of the time taken to perform the operations, expressed in terms of the input size n.

For example -

- A for loop that calculates sum of first n +ve integers, leads to a component that has a running time of n.
- A subroutine takes a set of n numbers and checks every number with the numbers higher in index in the list to find occurrences of repeated numbers. This subroutine may take up to $\frac{n(n-1)}{2}+1$ comparisons.

How to estimate running time of your program/algorithm?

A program is divided into several modules. The cost function of a program or an algorithm can be determined by line-by-line accumulation of the time taken to perform the operations, expressed in terms of the input size *n*.

For example -

- A for loop that calculates sum of first *n* +ve integers, leads to a component that has a running time of *n*.
- A subroutine takes a set of *n* numbers and checks every number with the numbers higher in index in the list to find occurrences of repeated numbers. This subroutine may take up to $\frac{n(n-1)}{2}+1$ comparisons.
- To convert a decimal number to binary (with division by 2) iterative halving can be used, which will lead to a program component of running time $\log_2 n$.

How to estimate running time of your program/algorithm?

A program is divided into several modules. The cost function of a program or an algorithm can be determined by line-by-line accumulation of the time taken to perform the operations, expressed in terms of the input size n.

For example -

- A for loop that calculates sum of first n +ve integers, leads to a component that has a running time of n.
- A subroutine takes a set of n numbers and checks every number with the numbers higher in index in the list to find occurrences of repeated numbers. This subroutine may take up to $\frac{n(n-1)}{2}+1$ comparisons.
- To convert a decimal number to binary (with division by 2) iterative halving can be used, which will lead to a program component of running time $\log_2 n$.

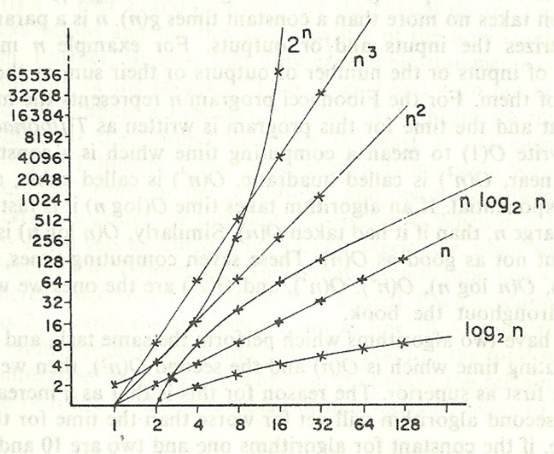
How to estimate running time of your program/algorithm?

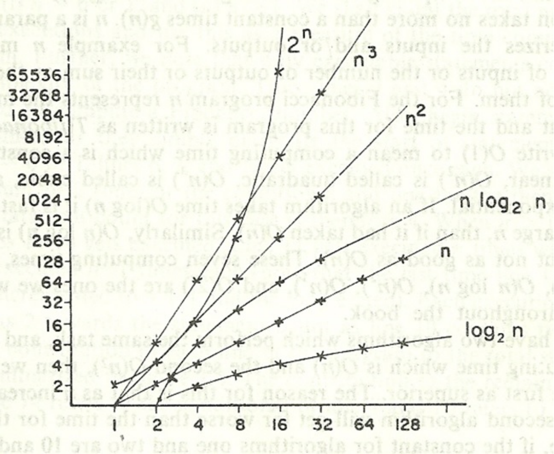
A program is divided into several modules. The cost function of a program or an algorithm can be determined by line-by-line accumulation of the time taken to perform the operations, expressed in terms of the input size n.

For example -

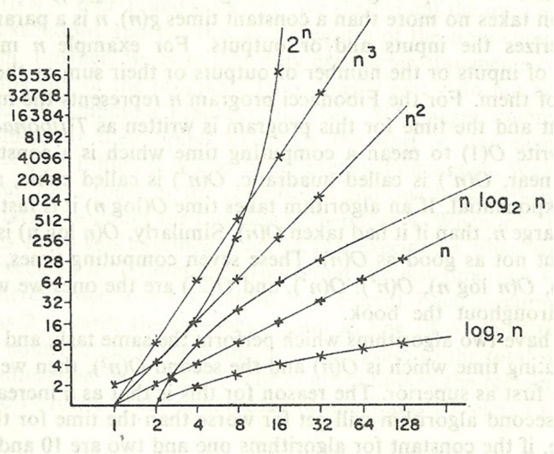
- A for loop that calculates sum of first n +ve integers, leads to a component that has a running time of n.
- A subroutine takes a set of n numbers and checks every number with the numbers higher in index in the list to find occurrences of repeated numbers. This subroutine may take up to $\frac{n(n-1)}{2}+1$ comparisons.
- To convert a decimal number to binary (with division by 2) iterative halving can be used, which will lead to a program component of running time $\log_2 n$.

It is important to understand that the time complexity of an algorithm is dependent on the number of comparisons performed throughout the algorithm.

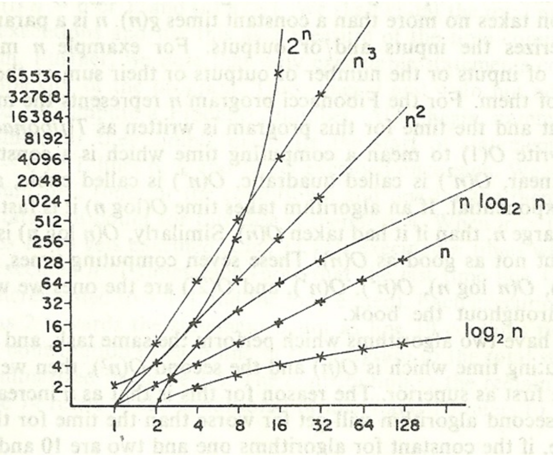

2^n
n^3
n^2
$n \log_2 n$
n
$\log_2 n$
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
4
8
16
32
64
128



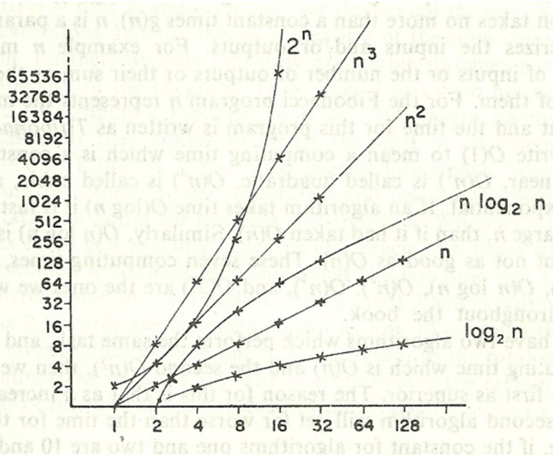
- We are really only interested in the **Order of Growth** of an algorithm's complexity.



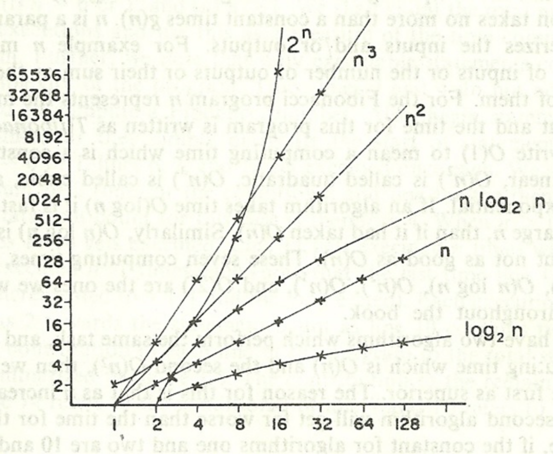
- We are really only interested in the **Order of Growth** of an algorithm's complexity.
- How well does the algorithm perform as the input size grows; $n \to \infty$.



- We are really only interested in the **Order of Growth** of an algorithm's complexity.
- How well does the algorithm perform as the input size grows; $n \to \infty$.
- We have seen how to mathematically evaluate the cost functions of algorithms with respect to their input size n and their elementary operation.



- We are really only interested in the **Order of Growth** of an algorithm's complexity.
- How well does the algorithm perform as the input size grows; $n \to \infty$.
- We have seen how to mathematically evaluate the cost functions of algorithms with respect to their input size n and their elementary operation.
- However, it suffices to simply measure a cost function's asymptotic behavior.



- We are really only interested in the **Order of Growth** of an algorithm's complexity.
- How well does the algorithm perform as the input size grows; $n \to \infty$.
- We have seen how to mathematically evaluate the cost functions of algorithms with respect to their input size n and their elementary operation.
- However, it suffices to simply measure a cost function's asymptotic behavior.
- Algorithms that have running times n^2 and $2000n^2$ are considered to be asymptotically equivalent.

Definitions

Big-Oh Notation (Asymptotic Upper Bound)

Given two non-negative functions $f(n)$ and $g(n)$ there exists an integer n_0 and a constant $c > 0$, such that $\forall$ integers $n \geq n_0$, $f(n) \leq cg(n)$, then $f(n) \in O(g(n))$.

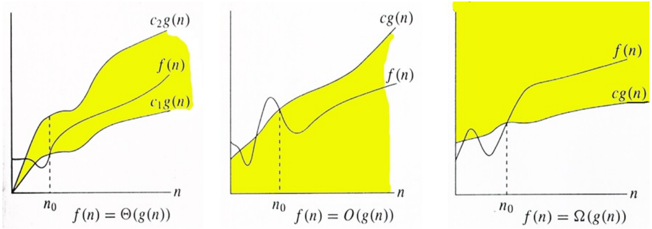
Big-Omega Notation (Asymptotic Lower Bound)

Given two non-negative functions $f(n)$ and $g(n)$ there exists an integer n_0 and a constant $c > 0$, such that $\forall$ integers $n \geq n_0$, $f(n) \geq cg(n)$, then $f(n) \in \Omega(g(n))$.

Big-Theta Notation (Asymptotic Equivalence)

Given two non-negative functions $f(n)$ and $g(n)$ there exists an integer n_0 and a constant c_1, $c_2 > 0$, such that $\forall$ integers $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$, then

$$f(n) \in \Theta(g(n)).$$



Little-Oh Notation (Loosely bounds from top)

Given two non-negative functions $f(n)$ and $g(n)$ there exists an integer n_0 for every positive constant c, such that $\forall$ integers $n \geq n_0$, $f(n) \leq cg(n)$, then $f(n) \in o(g(n))$.

Little-Omega Notation (Loosely bounds from bottom)

Given two non-negative functions $f(n)$ and $g(n)$ there exists an integer n_0 for every positive constant c, such that $\forall$ integers $n \geq n_0$, $f(n) \geq cg(n)$, then $f(n) \in \omega(g(n))$.

Revisiting the definitions

- $O(f)$: Function grows no faster than f. (Slower or equal)

Revisiting the definitions

- $O(f)$: Function grows no faster than f. (Slower or equal)
- $\Omega(f)$: Function grows no slower than f. (Faster or equal)

Revisiting the definitions

- $O(f)$: Function grows no faster than f. (Slower or equal)
- $\Omega(f)$: Function grows no slower than f. (Faster or equal)
- $\Theta(f)$: Functions grows at the same rate as f.

Revisiting the definitions

- $O(f)$: Function grows no faster than f. (Slower or equal)
- $\Omega(f)$: Function grows no slower than f. (Faster or equal)
- $\Theta(f)$: Functions grows at the same rate as f.
- $o(f) = O(f)$ - $\Theta(f)$ (Loosely bounds from top), function grows slower than f.

Revisiting the definitions

- $O(f)$: Function grows no faster than f. (Slower or equal)
- $\Omega(f)$: Function grows no slower than f. (Faster or equal)
- $\Theta(f)$: Functions grows at the same rate as f.
- $o(f) = O(f)$ - $\Theta(f)$ (Loosely bounds from top), function grows slower than f.
- $\omega(f) = \Omega(f)$ - $\Theta(f)$ (Loosely bounds from bottom), function grows faster than f.

Categorizing Algorithms with Examples

Categorizing Algorithms with Examples

- **Constant time** - $O(\mathbf{1})$: Accessing elements of an array (independent on input size).

Categorizing Algorithms with Examples

- **Constant time** - $O(\mathbf{1})$: Accessing elements of an array (independent on input size).

- **Linear time** – $O(n)$: Searching a number sequentially from a list (dependent linearly on the size of the input sequence).

Categorizing Algorithms with Examples

- **Constant time** - $O(\mathbf{1})$: Accessing elements of an array (independent on input size).
- **Logarithmic time** – $O(log_2 n)$: Loops with iterative halving (eg. Binary search).
- **Linear time** – $O(n)$: Searching a number sequentially from a list (dependent linearly on the size of the input sequence).

Categorizing Algorithms with Examples

- **Constant time** - $O(\mathbf{1})$: Accessing elements of an array (independent on input size).
- **Double logarithmic time** – $O(log_2 log_2 n)$: Searching in a dictionary (eg. interpolation search).
- **Logarithmic time** – $O(log_2 n)$: Loops with iterative halving (eg. Binary search).
- **Linear time** – $O(n)$: Searching a number sequentially from a list (dependent linearly on the size of the input sequence).

Categorizing Algorithms with Examples (continued)

Categorizing Algorithms with Examples (continued)

- **Loglinear time** - $O(nlog_2n)$: Two nested loops – one with linear iteration and the other with iterative halving. In which order? Does not matter (e.g., sorting methods such as merge sort, quick sort).

Categorizing Algorithms with Examples (continued)

- **Loglinear time** - $O(nlog_2n)$: Two nested loops – one with linear iteration and the other with iterative halving. In which order? Does not matter (e.g., sorting methods such as merge sort, quick sort).
- **Quadratic time** – $O(n^2)$: Two nested loops with linear iteration (e.g., naïve sorting methods such as bubble sort).

Categorizing Algorithms with Examples (continued)

- **Loglinear time** - $O(nlog_2n)$: Two nested loops – one with linear iteration and the other with iterative halving. In which order? Does not matter (e.g., sorting methods such as merge sort, quick sort).
- **Quadratic time** – $O(n^2)$: Two nested loops with linear iteration (e.g., naïve sorting methods such as bubble sort).
- **Polynomial time** – $O(n^k)$: k many nested loops with linear iterations of size n.

Categorizing Algorithms with Examples (continued)

- **Loglinear time** - $O(nlog_2n)$: Two nested loops – one with linear iteration and the other with iterative halving. In which order? Does not matter (e.g., sorting methods such as merge sort, quick sort).
- **Quadratic time** – $O(n^2)$: Two nested loops with linear iteration (e.g., naïve sorting methods such as bubble sort).
- **Polynomial time** – $O(n^k)$: k many nested loops with linear iterations of size n.
- **Exponential time** – $O(2^n)$ **or** $O(c^n)$: Generating all possible combinations (e.g., Tower of Hanoi).

Categorizing Algorithms with Examples (continued)

- **Loglinear time** - $O(nlog_2n)$: Two nested loops – one with linear iteration and the other with iterative halving. In which order? Does not matter (e.g., sorting methods such as merge sort, quick sort).
- **Quadratic time** – $O(n^2)$: Two nested loops with linear iteration (e.g., naïve sorting methods such as bubble sort).
- **Polynomial time** – $O(n^k)$: k many nested loops with linear iterations of size n.
- **Exponential time** – $O(2^n)$ **or** $O(c^n)$: Generating all possible combinations (e.g., Tower of Hanoi).
- **Factorial time** – $O(n!)$: Generating all unrestricted permutations.

Outline for Part II

1. Time Complexity
   - Introduction
   - Asymptotics
   - Efficiency Classes
2. Recurrence Relations
   - Introduction
   - Solving Recurrences using Substitution
   - Solving Recurrences using Recurrence Tree
3. Master Theorem
   - Introduction
   - Drawbacks
   - Examples

A *recursive algorithm* is one in which objects are defined in terms of other objects of the same type.

Advantages:

- Simplicity of code
- Easy to understand

Drawbacks:

- Memory
- Speed
- Possibly redundant work

Tail recursion offers a solution to the memory problem, but really, do we need recursion?

Algorithm 1: POWER(x, n))

Input : Base x, Exponent n

Output: Power of x raised to n

```
begin
    if n == 0 then
        return 1;
    else
        return x*POWER(x,n-1)
```

- The pseudo-code simulates a subroutine that generates power of x raised to n.
- How many times will it call itself?
- Time complexity of recursive algorithms will depend on how many times the recursive function is called.

Forming and Solving Recurrence Relations

$$T(0) = c_1, T(n) = T(n-1) + c_2$$

If we knew $T(n\text{-}1)$, we could find $T(n)$

$$
\begin{aligned}
T(n) &= T(n-1) + c_2 \qquad T(n-1) = T(n-2) + c_2 \\
&= T(n-2) + c_2 + c_2 \\
&= T(n-2) + 2c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-3) + c_2 + 2c_2 \\
&= T(n-3) + 3c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-4) + c_2 + 3c_2 \\
&= T(n-4) + 4c_2 \\
&= \ldots \\
&= T(n-k) + kc_2
\end{aligned}
$$

Forming and Solving Recurrence Relations

$$T(0) = c_1, T(n) = T(n-1) + c_2$$

If we knew $T(n\text{-}1)$, we could find $T(n)$

$$\begin{aligned}
T(n) &= T(n-1) + c_2 \qquad T(n-1) = T(n-2) + c_2 \\
&= T(n-2) + c_2 + c_2 \\
&= T(n-2) + 2c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-3) + c_2 + 2c_2 \\
&= T(n-3) + 3c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-4) + c_2 + 3c_2 \\
&= T(n-4) + 4c_2 \\
&= \dots \\
&= T(n-k) + kc_2
\end{aligned}$$

If we set $k = n$, we have -

$$\begin{aligned}
T(n) &= T(n-n) + nc_2 \\
&= T(0) + nc_2 \\
&= c_1 + nc_2 \\
&\in \Theta(n)
\end{aligned}$$

Is there a better solution (in terms of efficiency)?

Alternative Solution - Is it better?

Algorithm 2: POWER(x, n))

Input : Base x, Exponent n
Output: Power of x raised to n

```
begin
    if n == 0 then return 1

    if n == 1 then return x

    if (n%2) == 0 then
        return POWER(x, n/2)*POWER(x, n/2)
    else
        return POWER(x, n/2)*POWER(x, n/2)*x
```

The recurrence relation would look something like this,

$$
\begin{aligned}
T(0) &= c_1 \\
T(1) &= c_2 \\
T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c_3 \\
&= 2T\left(\frac{n}{2}\right) + c_3
\end{aligned}
$$

Solving Recurrence Relation : Modified Solution

$$T(0) = c_1, T(n) = T(n-1) + c_2$$

If we knew $T(n\text{-}1)$, we could find $T(n)$

$$
\begin{aligned}
T(n) &= T(n-1) + c_2 \qquad T(n-1) = T(n-2) + c_2 \\
&= T(n-2) + c_2 + c_2 \\
&= T(n-2) + 2c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-3) + c_2 + 2c_2 \\
&= T(n-3) + 3c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-4) + c_2 + 3c_2 \\
&= T(n-4) + 4c_2 \\
&= \dots \\
&= T(n-k) + kc_2
\end{aligned}
$$

Solving Recurrence Relation : Modified Solution

$$T(0) = c_1, T(n) = T(n-1) + c_2$$

If we knew $T(n\text{-}1)$, we could find $T(n)$

$$\begin{aligned}
T(n) &= T(n-1) + c_2 \qquad T(n-1) = T(n-2) + c_2 \\
&= T(n-2) + c_2 + c_2 \\
&= T(n-2) + 2c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-3) + c_2 + 2c_2 \\
&= T(n-3) + 3c_2 \qquad T(n-2) = T(n-3) + c_2 \\
&= T(n-4) + c_2 + 3c_2 \\
&= T(n-4) + 4c_2 \\
&= \ldots \\
&= T(n-k) + kc_2
\end{aligned}$$

If we set $k = n$, we have -

$$\begin{aligned}
T(n) &= T(n-n) + nc_2 \\
&= T(0) + nc_2 \\
&= c_1 + nc_2 \\
&\in \Theta(n)
\end{aligned}$$

Is there a better solution (in terms of efficiency)?

Solving Recurrence Relation : Modified Solution

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + c_3 \qquad\qquad T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c_3 \\
&= 2\left(2T\left(\frac{n}{4}\right) + c_3\right) + c_3 \\
&= 4T\left(\frac{n}{4}\right) + 3c_3 \qquad\qquad T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + c_3 \\
&= 4\left(2T\left(\frac{n}{8}\right) + c_3\right) + c_3 \\
&= 8T\left(\frac{n}{8}\right) + 7c_3 \qquad\qquad T\left(\frac{n}{8}\right) = 2T\left(\frac{n}{16}\right) + c_3 \\
&= 8\left(2T\left(\frac{n}{16}\right) + c_3\right) + 7c_3 \\
&= 16T\left(\frac{n}{16}\right) + 15c_3 \\
&= \dots \\
&= 2^kT\left(\frac{n}{2^k}\right) + (2^k - 1)c_3
\end{aligned}
$$

Solving Recurrence Relation : Modified Solution (contd.)

$$T(0) = c_1$$
$$T(1) = c_2$$
$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)c_3$$

Pick a value for k, such that $\frac{n}{2^k} = 1$

$$\frac{n}{2^k} = 1$$
$$n = 2^k$$
$$k = logn$$

$$T(n) = 2^{logn} T\left(\frac{n}{2^{logn}}\right) + (2^{logn} - 1)c_3$$
$$= nT\left(\frac{n}{n}\right) + (n-1)c_3 = nT\left(1\right) + (n-1)c_3$$
$$= nc_2 + (n-1)c_3 \in \Theta(n)$$

Another Alternative Solution - How about this one?

Algorithm 3: POWER(x, n))

Input : Base x, Exponent n
Output: Power of x raised to n

```
begin
    if n == 0 then return 1

    if n == 1 then return x

    if (n%2) == 0 then
        return POWER(x*x, n/2))
    else
        return POWER(x*x, n/2)*x
```

The recurrence relation would look something like this,

$$T(0) = c_1$$
$$T(1) = c_2$$
$$T(n) = T\left(\frac{n}{2}\right) + c_3$$

Solving Recurrence Relation : Another Modified Solution

$$
\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + c_3 \qquad\qquad T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + c_3 \\
&= \left(T\left(\frac{n}{4}\right) + c_3\right) + c_3 \\
&= T\left(\frac{n}{4}\right) + 2c_3 \qquad\qquad T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + c_3 \\
&= \left(T\left(\frac{n}{8}\right) + c_3\right) + 2c_3 \\
&= T\left(\frac{n}{8}\right) + 3c_3 \qquad\qquad T\left(\frac{n}{8}\right) = T\left(\frac{n}{16}\right) + c_3 \\
&= \left(T\left(\frac{n}{16}\right) + c_3\right) + 3c_3 \\
&= T\left(\frac{n}{16}\right) + 4c_3 \\
&= \dots \\
&= T\left(\frac{n}{2^k}\right) + kc_3
\end{aligned}
$$

Solving Recurrence Relation : Another Modified Solution (contd.)

$$T(0) = c_1$$
$$T(1) = c_2$$
$$T(n) = T\left(\frac{n}{2^k}\right) + kc_3$$

Pick a value for k, such that $\frac{n}{2^k} = 1$

$$\frac{n}{2^k} = 1$$
$$n = 2^k$$
$$k = logn$$

$$T(n) = T\left(\frac{n}{2^{logn}}\right) + lognc_3$$
$$= T(1) + lognc_3$$
$$\in \Theta(logn)$$

More generally, recurrences can have the following forms,

$$T(n) = \alpha T(n-\beta) + f(n), T(\delta) = c$$

or

$$T(n) = \alpha T\left(\frac{n}{\beta}\right) + f(n), T(\delta) = c$$

The initial conditions are defined by T(δ).

More generally, recurrences can have the following forms,

$$T(n) = \alpha T(n - \beta) + f(n), T(\delta) = c$$

or

$$T(n) = \alpha T\left(\frac{n}{\beta}\right) + f(n), T(\delta) = c$$

The initial conditions are defined by T(δ).

Recurrence relations may consist of two parts: recursive and non-recursive.

$$T(n) = \underbrace{2T(n-2)}_{\text{recursive}} + \underbrace{n^2 - 10}_{\text{non-recursive}}$$

Recursive terms come from when an algorithm calls itself.
Non-recursive terms correspond to the *non-recursive* cost of the algorithm — work the algorithm performs *within* a function.

Forward Substitution Method

The forward substitution method for solving recurrences consists of two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find constants in the form and show that the solution works.

The inductive hypothesis is applied to smaller values, similar like recursive calls bring us closer to the base case.

The substitution method is powerful to establish lower or upper bounds on a recurrence.

Example 1

The recurrence relation for the cost of a divide-and-conquer method is,

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

Our induction hypothesis is $T(n)$ is $O(nlog_2(n))$ or $T(n) \leq cnlog_2(n)$ for some constant c, independent of n.
Assume the hypothesis holds for all m ¡ n and substitute:

$$\begin{aligned} T(n) &\leq 2\left(c\left\lfloor \frac{n}{2} \right\rfloor log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) + n \\ &\leq cnlog_2\left(\frac{n}{2}\right) + n \\ &= cnlog_2(n) - cnlog_2(2) + n \\ &= cnlog_2(n) - cn + n \\ &\leq cnlog_2(n) \quad \text{(as long as } \ c \geq 1) \end{aligned}$$

Boundary Conditions

We should also show that the base case holds.

Assuming $T(1) = 1$, we would like to show,

$$T(1) \leq c.1.log_2(1) = c.0 = 0 \quad \text{(which is impossible when } T(1) > 0)$$

which is impossible when $T(1) > 0$.

We only want to show that $T(n) \leq cnlog_2(n)$ for sufficiently large values of n; i.e. $\forall\ n \geq n_0$. (We can try $n_0 > 1$).

The base case

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

We have,

$$T(1) = 1 \Rightarrow \begin{cases} T(2) = 4 \\ T(3) = 5 \end{cases}$$

We want to satisfy simultaneously,

$$\begin{cases} 4 = T(2) \leq c.2.log_2(2) \\ 5 = T(3) \leq c.3.log_2(3) \end{cases} \implies \begin{cases} c \geq 2 \\ c \geq \frac{5}{3log_2(3)} \approx 1.052 \end{cases} \implies c \geq 2$$

We have to check both $T(2)$ and $T(3)$ simultaneously because of the nature of the recursive equation.

Lower bound

We want to show $T(n) \geq cnlog_2(n)$. Assume that n is a power of 2. We have,

$$\begin{aligned} T(n) &\geq 2\left(c\left\lfloor\frac{n}{2}\right\rfloor log_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\right) + n \\ &= cnlog_2\left(\frac{n}{2}\right) + n \\ &= cnlog_2(n) - cnlog_2(2) + n \\ &= cnlog_2(n) - cn + n \\ &\geq cnlog_2(n) \quad (\text{as long as } \ c \leq 1) \end{aligned}$$

We also want to satisfy the boundary condition ($T(2) = 4$).

$$T(2) \geq c.2.log_2(2) = 2.c$$

In other words, it is enough if $c \leq 2$. By the requirement $c \leq 1$ for the induction step, we choose $c = 1$.

Lower bound

We will prove that $T(n)$ is strictly increasing.
For the base case, note that $T(1) = 1 < 4 = T(2)$

Assuming that for all $k \leq n$ it holds $T(k) > T(k - 1)$, we want to show that $T(n + 1) > T(n)$. We distinguish cases for $n + 1$.

$(n + 1)$ is odd
Say, $n + 1 = 2m + 1$. Then, it holds

$$\begin{aligned} T(2m+1) &= 2T\left(\left\lfloor \frac{2m+1}{2} \right\rfloor\right) + 2m + 1 \\ &= 2T(m) + 2m + 1 \\ &= T(2m) + 1 \\ &> T(2m) \end{aligned}$$

Lower bound

$(n+1)$ is even

Say, $n+1 = 2m$. Then, it holds

$$\begin{aligned}
T(2m) &= 2T\left(\left\lfloor \frac{2m}{2} \right\rfloor\right) + 2m \\
&= 2T(m) + 2m \\
&> 2T(m-1) + 2m \\
&= 2T\left(\left\lfloor \frac{2m-1}{2} \right\rfloor\right) + (2m-1) + 1 \\
&= T(2m-1) + 1 \\
&> T(2m-1)
\end{aligned}$$

Note that the induction hypothesis is used only when $(n+1)$ is even.

Loose bound

Consider the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

Our guess is $O(n)$, so we try to show $T(n) \leq cn$.

$$T(n) \leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1$$

which **does not** imply that $T(n) \leq cn$, for any c. We need to show the exact form.

Ideas to overcome the hurdle:

1. Revise our guess; say $T(n) = O(n^2)$.
    - However, our original guess was correct.
2. Sometimes it is easier to prove something stronger.

A stronger bound

We will attempt to show $T(n) = cn$ - b, where b is another constant.
We have,

$$\begin{aligned} T(n) &= \left(c\left\lfloor\frac{n}{2}\right\rfloor - b\right) + \left(c\left\lceil\frac{n}{2}\right\rceil - b\right) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \quad (\text{for b} \geq 1) \end{aligned}$$

We still have to specify c.

Assume that $T(1) = 1$. We want $T(1) = 1 \leq c.1$ - b

Hence, it is enough to set $c = 2$ and $b = 1$.

Consider the recurrence,

$$T(n) = 2T(\sqrt{n}) + log_2(n)$$

Replace, $m = log_2(n)$. We have,

$$T(2m) = 2T\left(2\frac{m}{2}\right) + m$$

Define $S(m) = T(2m)$. We get,

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

Hence, the solution is $O(mlog_2(m))$, or with substitution $O(log_2(n)log_2(log_2(n)))$

How to use recurrence trees

Recurrence trees are considered as a useful tool to solve recurrence relations.

Recurrence trees can help solving a recurrence relation in following steps,

1. Expanding the recurrence relation into a tree.
2. Determining of the height of the tree.
3. Summing the cost at each level.
4. Accumulate cost from each level of the tree over all the levels (i.e., the height of the tree).

Problem 1:

Consider the recurrence relation,

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

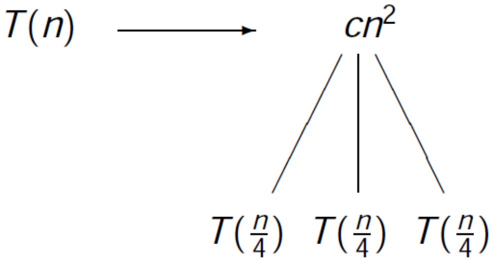
We assume, c is a constant and n is an exact power of 4.

Problem 1:

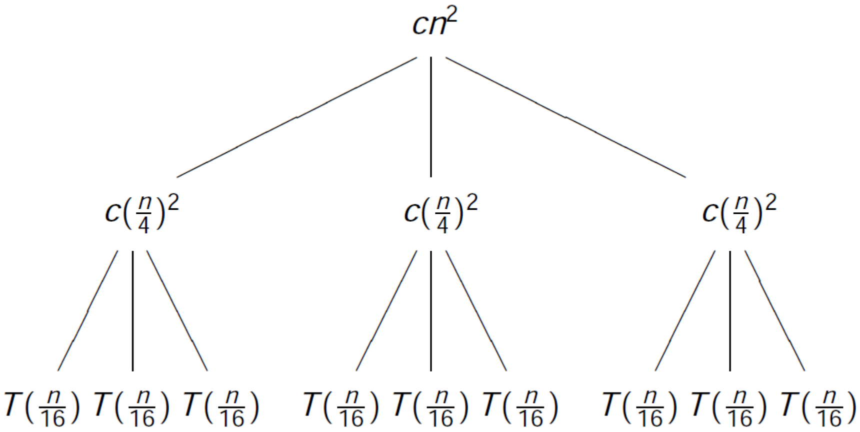
Consider the recurrence relation,

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

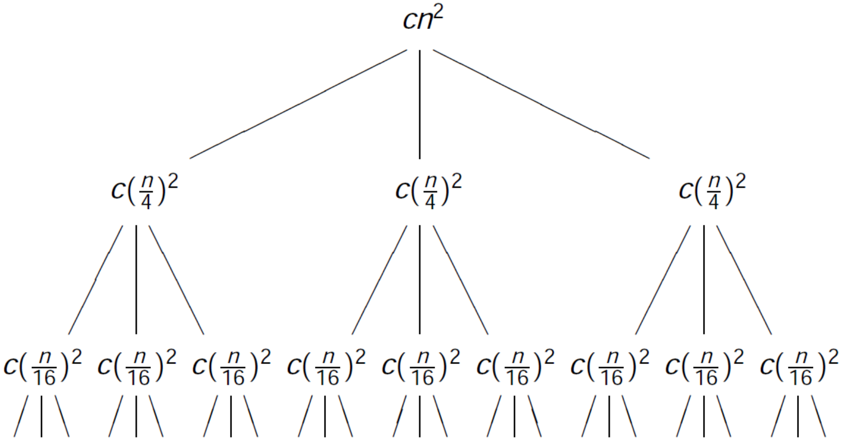
We assume, c is a constant and n is an exact power of 4.



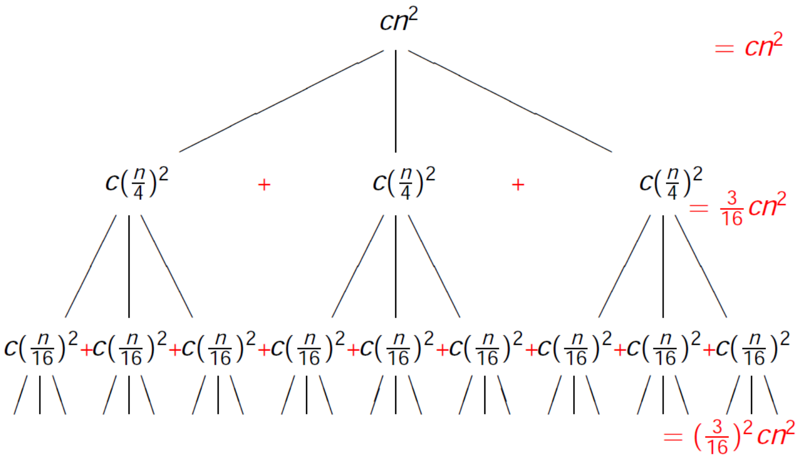
Problem 1: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$



Problem 1: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$



Problem 1: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$



Problem 1: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

$$\begin{aligned} T(n) &= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots \\ &= cn^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \dots\right) \end{aligned}$$

Problem 1: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

$$T(n) = cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots$$
$$= cn^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \dots\right)$$

If n $\geq$ 16, i.e., 4^2, the height of the tree is at least 2.

Problem 1: $\mathrm{T}(n) = 3\mathrm{T}\left(\frac{n}{4}\right) + cn^2$

$$
\begin{aligned}
T(n) &= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots \\
&= cn^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \dots\right)
\end{aligned}
$$

If n $\geq$ 16, i.e., 4^2, the height of the tree is at least 2.
For $n = 4^k$, $k = log_4(n)$, we have,

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

Problem 1: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

$$
\begin{aligned}
T(n) &= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots \\
&= cn^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \dots\right)
\end{aligned}
$$

If n $\geq$ 16, i.e., 4^2, the height of the tree is at least 2.
For $n = 4^k$, $k = log_4(n)$, we have,

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

To remove $log_4(n)$ factor, for some constant d, we consider,

$$T(n) \leq cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i = cn^2 \frac{-1}{\frac{3}{16} - 1} \leq dn^2$$

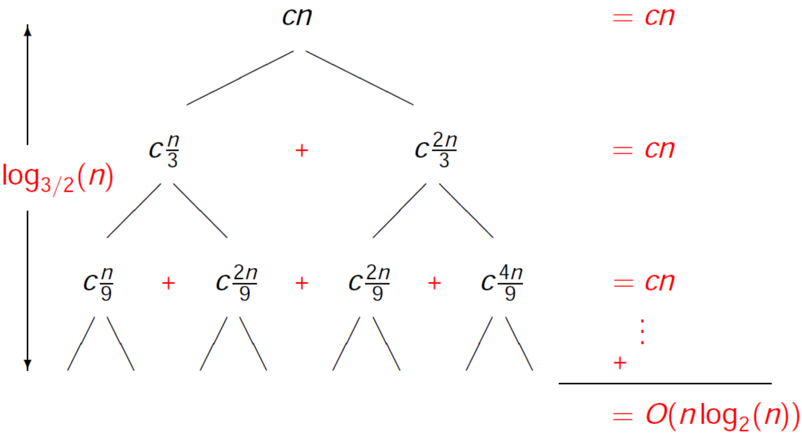
Problem 1: $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

We verify $T(n) \leq dn^2$ by applying substitution method,

$$
\begin{aligned}
T(n) &= 3T\left(\frac{n}{4}\right) + cn^2 \\
&\leq 3d\left(\frac{n}{4}\right)^2 + cn^2 \\
&= \left(\frac{3}{16}d + c\right)n^2 \\
&= \frac{3}{16}\left(d + \frac{16}{3}c\right)n^2 \\
&\leq \frac{3}{16}(2d)n^2, \quad \text{if } d \geq \frac{16}{3}c \\
&\leq dn^2
\end{aligned}
$$

Problem 2: $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$

Problem 2: $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$



Problem 2: $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$

Verification by substitution method,

$$\begin{aligned}
T(n) &\leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \\
&\leq d\left(\frac{n}{3}\right)log\left(\frac{n}{3}\right) + d\left(\frac{2n}{3}\right)log\left(\frac{2n}{3}\right) + cn \\
&= dnlogn - d\left(\left(\frac{n}{3}\right)log(3) + d\left(\frac{2n}{3}\right)log\left(\frac{3}{2}\right)\right) + cn \\
&= dnlogn - dn\left(log(3) - \frac{2}{3}\right) + cn \\
&\leq dnlogn
\end{aligned}$$

We assume, d $\geq \frac{c}{log(3)-\frac{2}{3}}$

Outline for Part III

1. Time Complexity
   - Introduction
   - Asymptotics
   - Efficiency Classes
2. Recurrence Relations
   - Introduction
   - Solving Recurrences using Substitution
   - Solving Recurrences using Recurrence Tree
3. Master Theorem
   - Introduction
   - Drawbacks
   - Examples

Solving Recurrences using Master Theorem

When analyzing algorithms, recall that we only care about the *asymptotic behavior*.

Solving Recurrences using Master Theorem

When analyzing algorithms, recall that we only care about the *asymptotic behavior*.

Recursive algorithms are no different. Rather than *solve* exactly the recurrence relation associated with the cost of an algorithm, it is enough to give an asymptotic characterization.

Solving Recurrences using Master Theorem

When analyzing algorithms, recall that we only care about the *asymptotic behavior*.

Recursive algorithms are no different. Rather than *solve* exactly the recurrence relation associated with the cost of an algorithm, it is enough to give an asymptotic characterization.

The main tool for doing this is the *master theorem*.

Master Theorem

Theorem

Let $T(n)$ be a monotonically increasing function that satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d logn) & \text{if } a = b^d \\ \Theta(n^{log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem cannot be used if,

- $T(n)$ is not a monotone, ex: $T(n) = sin(n)$

Master Theorem cannot be used if,

- $T(n)$ is not a monotone, ex: $T(n) = sin(n)$
- $f(n)$ is not a polynomial, ex: $T(n) = 2T(\frac{n}{2}) + n^n$

Master Theorem cannot be used if,

- $T(n)$ is not a monotone, ex: $T(n) = sin(n)$
- $f(n)$ is not a polynomial, ex: $T(n) = 2T(\frac{n}{2}) + n^n$
- b cannot be expressed as a constant, ex: $T(n) = T(\sqrt{n})$

Fourth Condition

Recall that we cannot use the Master Theorem if $f(n)$ (the non-recursive cost) is not polynomial.

There is a limited 4-th condition of the Master Theorem that allows us to consider polylogarithmic functions.

Corollary

If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$ then

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

This final condition is fairly limited and is presented merely for completeness.

Problem 1:

Let $T(n) = T(\frac{n}{2}) + \frac{1}{2}n^2 + n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a =$

Problem 1:

Let $T(n) = T(\frac{n}{2}) + \frac{1}{2}n^2 + n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$
$b =$

Problem 1:

Let $T(n) = T(\frac{n}{2}) + \frac{1}{2}n^2 + n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$
$b = 2$
$d =$

Problem 1:

Let $T(n) = T(\frac{n}{2}) + \frac{1}{2}n^2 + n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$
$b = 2$
$d = 2$

Since, $1 < 2^2$, case 1 ($a < b^d$) applies.

Problem 1:

Let $T(n) = T(\frac{n}{2}) + \frac{1}{2}n^2 + n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$
$b = 2$
$d = 2$

Since, $1 < 2^2$, case 1 ($a < b^d$) applies.

Thus we conclude that

$$T(n) \in \Theta(n^d) = \Theta(n^2)$$

Problem 2:

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a =$

Problem 2:

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 2$
$b =$

Problem 2:

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 2$
$b = 4$
$d =$

Problem 2:

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 2$
$b = 4$
$d = \frac{1}{2}$

Since, 2 equals to $4^{\frac{1}{2}}$, case 2 ($a = b^d$) applies.

Problem 2:

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 2$
$b = 4$
$d = \frac{1}{2}$

Since, 2 equals to $4^{\frac{1}{2}}$, case 2 ($a = b^d$) applies.

Thus we conclude that

$$T(n) \in \Theta(n^d \log n) = \Theta(\sqrt{n} \log n)$$

Problem 3:

Let $T(n) = 3T(\frac{n}{2}) + \frac{3}{4}n + 1$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a =$

Problem 3:

Let $T(n) = 3T(\frac{n}{2}) + \frac{3}{4}n + 1$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b =$

Problem 3:

Let $T(n) = 3T(\frac{n}{2}) + \frac{3}{4}n + 1$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b = 2$
$d =$

Problem 3:

Let $T(n) = 3T(\frac{n}{2}) + \frac{3}{4}n + 1$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b = 2$
$d = 1$

Since, $3 > 2^1$, case 3 ($a > b^d$) applies.

Problem 3:

Let $T(n) = 3T(\frac{n}{2}) + \frac{3}{4}n + 1$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b = 2$
$d = 1$

Since, $3 > 2^1$, case 3 $(a > b^d)$ applies.

Thus we conclude that

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.5849})$$

Problem 4:

Let $T(n) = 3T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a =$

Problem 4:

Let $T(n) = 3T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b =$

Problem 4:

Let $T(n) = 3T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b = 2$
$d =$

Problem 4:

Let $T(n) = 3T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b = 2$
$d = 2$

Since, $3 < 2^2$, case 1 $(a < b^d)$ applies.

Problem 4:

Let $T(n) = 3T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 3$
$b = 2$
$d = 2$

Since, $3 < 2^2$, case 1 $(a < b^d)$ applies.

Thus we conclude that

$$T(n) \in \Theta(n^d) = \Theta(n^2)$$

Problem 5:

Let $T(n) = 4T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a =$

Problem 5:

Let $T(n) = 4T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 4$
$b =$

Problem 5:

Let $T(n) = 4T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 4$
$b = 2$
$d =$

Problem 5:

Let $T(n) = 4T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 4$
$b = 2$
$d = 2$

Since, $4 = 2^2$, case 2 ($a = b^d$) applies.

Problem 5:

Let $T(n) = 4T(\frac{n}{2}) + n^2$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 4$
$b = 2$
$d = 2$

Since, $4 = 2^2$, case 2 $(a = b^d)$ applies.

Thus we conclude that

$$T(n) \in \Theta(n^d \log n) = \Theta(n^2 \log n)$$

Problem 6:

Let $T(n) = T(\frac{n}{2}) + 2^n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a =$

Problem 6:

Let $T(n) = T(\frac{n}{2}) + 2^n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$, $b =$

Problem 6:

Let $T(n) = T(\frac{n}{2}) + 2^n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$, $b = 2$, $d =$

Problem 6:

Let $T(n) = T(\frac{n}{2}) + 2^n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$, $b = 2$, $d = ?$
Here, we can apply, modified version of case 1.

1. There is an $\epsilon > 0$, for which $f(n) = \Omega(n^{log_b a+\epsilon})$, and
2. There is a $c < 1$ such that $af(n/b) \leq cf(n)$ for all n sufficiently large, then

$$T(n) \in \Theta(n^d)$$

Problem 6:

Let $T(n) = T(\frac{n}{2}) + 2^n$.
What are the parameters a, b, c? Therefore, which conditions should be applied?

$a = 1$, $b = 2$, $d = ?$
Here, we can apply, modified version of case 1.

1. There is an $\epsilon > 0$, for which $f(n) = \Omega(n^{log_b a+\epsilon})$, and
2. There is a $c < 1$ such that $af(n/b) \leq cf(n)$ for all n sufficiently large, then

$$T(n) \in \Theta(n^d)$$

We can show that, $f(n) \in \Omega(n^2)$
For c=0.5, we can prove $af(n/b) \leq cf(n)$, therefore,

$$T(n) \in \Theta(2^n)$$

Example: Fourth Condition

Say that we have the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + n\text{log}n$$

Example: Fourth Condition

Say that we have the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + nlogn$$

Clearly, $a = 2$, $b = 2$ but $f(n)$ is not a polynomial. However,

$$f(n) \in \Theta(nlogn)$$

Example: Fourth Condition

Say that we have the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + nlogn$$

Clearly, $a = 2$, $b = 2$ but $f(n)$ is not a polynomial. However,

$$f(n) \in \Theta(nlogn)$$

for $k = 1$, therefore, by the 4-th case of the Master Theorem we can say that

$$T(n) \in \Theta(nlog^2 n)$$