# CSC 421: Algorithm Design and Analysis (3 credits)

Dr. K. S. Adewole

**COURSE SYLLABUS:**

**Module 1: Introduction**
- Overview of Algorithm
- Algorithm Analysis
- Algorithm Complexity
- Rate of Growth
- Classification of Growth
- Mathematical Background
- Divide and Conquer Algorithms
- Recurrence Relation

**Module 2: Searching and Selection Algorithms**
- Sequential Search
- Binary Search
- Selection algorithms

**Module 3: Sorting Algorithms**
- Insertion Sort
- Bubble Sort
- Shellsort
- Radix Sort
- Heap Sort
- Merge Sort
- Quick Sort

**Module 4: Some algorithm design techniques**
- Divide and Conquer revisited
- Greedy Algorithms
- Dynamic Programming
- Amortized and Potential Analysis

**Module 5: Numerical Algorithms**
- Calculating Polynomials
- Horner's Method
- Matrix Multiplication
- Winograd's Matrix Multiplication
- Strassen's Matrix Multiplication
- Linear Equations

**Module 6: Matching Algorithm**
- String Matching
- Knuth-Morris-Pratt Algorithm
- Boyer-Moore Algorithm

**Module 7: Nondeterministic Algorithms**
- What is NP?

- Typical NP Problems

**Expectation:**
Students are expected to attend all lectures, submit all assignments/projects, write test and final examination.

**Performance Evaluation:**
Attendance: compulsory; Assignments: 10%; Test: 30%; Final Examination: 60%.

**Bibliography:**

McConnell, J. J. (2001). Analysis of Algorithms: AN Active Learning Approach. USA:Jones and Bartlett Publishers.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. S. (2001). Introduction to Algorithms (2nd Ed.). England:MIT Press.

## MODULE 1: INTRODUCTION

### 1.1 Overview of Algorithm

The word **Algorithm** means "a process or set of rules to be followed in calculations or other problem-solving operations". Therefore, Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.
It can be understood by taking an example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Similarly, algorithms help to do a task in programming to get the expected output. The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

### What are the Characteristics of an Algorithm?

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finiteness:** The algorithm must be finite, i.e. it should not end up in an infinite loop.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed with the available resources.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

### How to Design an Algorithm?

In order to write an algorithm, the following things are needed as pre-requisite:

1. The **problem** that is to be solved by this algorithm.
2. The **contraints** of the problem that must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem the is solved.
5. The **solution** to this problem, in the given contraints. Naturally, there can be many solutions to a given problem. *This brings about the need to analyze which solution is the best within the solution space*.

Then the algorithm is written with the help of above parameters such that it solves the problem.

**Example:** Consider the example to add three numbers and print the sum.

- **Step 1: Fulfilling the pre-requisites**
  As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.

    1. **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.

2. **The contraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
3. **The input to be taken to solve the problem:** The three numbers to be added.
4. **The output to be expected when the problem is solved:** The sum of the three numbers taken as the input.
5. **The solution to this problem, in the given contraints:** The solution consists of adding the 3 numbers. It can be done in different ways such as with the help of '+' operator, or bitwise, or any other method.

- **Step 2: Designing the algorithm**
  Now let's design the algorithm with the help of above pre-requisites:

  **Algorithm to add 3 numbers and print their sum:**
  **Input:** three numbers to be added
  **Output:** the sum of the three numbers
  1. START
  2. Declare 3 integer variables num1, num2 and num3.
  3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
  4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
  5. Add the 3 numbers and store the result in the variable sum.
  6. Print the value of variable sum
  7. END

- **Step 3: Testing the algorithm by implementing it.**
  In order to test the algorithm, let's implement it in C language.

  **Program:**

```
// C program to add three numbers
// with the help of above designed algorithm

#include <stdio.h>

int main()
{

  // Variables to take the input of the 3 numbers
  int num1, num2, num3;

  // Variable to store the resultant sum
  int sum;

  // Take the 3 numbers as input
  printf("Enter the 1st number: ");
  scanf("%d", &num1);

  printf("Enter the 2nd number: ");
  scanf("%d", &num2);

  printf("Enter the 3rd number: ");
  scanf("%d", &num3);
```

```
    // Calculate the sum using + operator
    // and store it in variable sum
    sum = num1 + num2 + num3;

    // Print the sum
    printf("\nSum of the 3 numbers is: %d", sum);

    return 0;
}
```

- **Output:**
- Enter the 1st number: 2
- Enter the 2nd number: 3
- Enter the 3rd number: 5
- Sum of the 3 numbers is: 10

### 1.2 Algorithm Analysis

As stated earlier, there can be many solutions to a given problem (i.e many algorithms may exist). For a standard algorithm to be good, it must be *efficient*. Hence, the efficiency of an algorithm must be checked and maintained.

*Analysis of an algorithm provides background information that gives us a general idea of how long an algorithm will take for a given problem set*. For each algorithm considered, we will come up with an estimate of ***how long*** it will take to solve the problem or ***what space*** it will take to solve the given problem. The formal is called ***time complexity*** check of an algorithm while the latter is ***space complexity***. Studying the analysis of algorithms gives us the tools to choose between algorithms.

### 1.3 Algorithm Complexity

An algorithm is defined as complex based on the amount of ***Space*** and ***Time*** it consumes. Hence, the *Complexity* of an algorithm refers to the measure of the Time that it will need to execute and get the expected output, and the Space it will need to store all the data (*input, temporary data and output*). Hence, these *two factors* define the ***efficiency*** of an algorithm.

**The two factors of Algorithm Complexity are:**

- **Time Factor**: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor**: Space is measured by counting the maximum memory space required by the algorithm.

Therefore, the complexity of an algorithm can be divided into two types:

1. **Space Complexity:** Space complexity of an algorithm refers to the *amount of memory* that this algorithm requires to execute and get the result. This can be for inputs, temporary operations, or outputs.

   **How to calculate Space Complexity?**
   The space complexity of an algorithm is calculated by determining the following 2 components:

- **Fixed Part:** This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.
- **Variable Part:** This refers to the *space that can be different based on the implementation of the algorithm*. For example, temporary variables, dynamic memory allocation, recursion stack space, etc. *Adding the memory spaces consumed for both fixed part and variable part gives us the space complexity of the algorithm*. E.g if the *fixed part* takes 4 bytes and the *variable part* takes 10 bytes, for a particular algorithm, then the *Space complexity* of such algorithm is 14 bytes.

Most of what we will be discussing in this course is going to be how efficient various algorithms are in terms of time (i.e Time Complexity). *Space complexity analysis was critical in the early days of computing* when storage space on a computer (both internal and external) was limited. Looking at software that is on the market today, it is easy to see that space analysis is not being done. Programs, even simple ones, regularly quote space needs in a number of megabytes.

2. **Time Complexity:** Time complexity of an algorithm refers to the amount of time that this algorithm requires to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

**How to calculate Time Complexity?**
The time complexity of an algorithm is also calculated by determining the following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, etc.
- **Variable Time Part:** Any instruction that is executed more than once, say *n* times, comes in this part. For example, loops, recursion, etc.

For instance, consider the algorithm analysis below:

| | |
|---|---|
| Step 1: | --Constant Time |
| Step 2: | --Constant Time |
| Step 3: | --Variable Time (Till the length of the Array, say n, or the index of the found element) |
| Step 4: | --Constant Time |
| Step 5: | --Constant Time |
| Step 6: | --Constant Time |

Hence, $T(p) = 5 + n$, which can be said as $T(n)$. $T(n)$ is taken as the final time complexity because it is the highest term. We shall discuss better on how to calculate time complexity as we progress in this Module.

**What to Count and Consider**

Deciding what to count involves two steps. The first is choosing the significant operations and the second is deciding which of those operations are integral to the algorithm. *There are two classes of operations* that are typically chosen for the significant operation: ***comparison***

***or arithmetic***. The *comparison* operators are all considered equivalent and are counted in algorithms such as searching and sorting. In these algorithms, the important task being done is the comparison of two values to determine, *when searching*, if the value is the one, we are looking for or, *when sorting*, if the values are out of order. *Comparison operations include equal, not equal, less than, greater than, less than or equal, and greater than or equal*. The second significant operation is arithmetic operation. *Arithmetic operators* are in two groups: ***additive and multiplicative***. Additive operators (usually called additions for short) include *addition, subtraction, increment, and decrement*. Multiplicative operators (usually called multiplications for short) include *multiplication, division, and modulus*. These two groups are counted separately because multiplications are considered to take longer than additions. In fact, some algorithms are viewed more favorably if they reduce the number of multiplications even if that means a similar increase in the number of additions.

**Cases to Consider**

***Input*** to an algorithm has a lot do when analyzing algorithms. *Choosing what input to consider when analyzing an algorithm can have a significant impact on how an algorithm will perform*. If the input list is already sorted, some sorting algorithms will perform very well, but other sorting algorithms may perform very poorly. The opposite may be true if the list is randomly arranged instead of sorted. *Because of this, we will not consider just one input set when we analyze an algorithm*. We will look at the input sets that allow an algorithm to perform the ***most quickly*** and the ***most slowly***. We will also consider an overall ***average*** performance of the algorithm as well. There are three cases that can be considered when carrying out time complexity of an algorithm.

**(1) Best Case:** As its name indicates, the best case for an algorithm is the ***input*** that requires the algorithm to take the ***shortest time***. This input is the combination of values that causes the algorithm to do the least amount of work. If we are looking at a searching algorithm, the best case would be if the value we are searching for (commonly called the target or key) was the value stored in the first location that the search algorithm would check. This would then require only one comparison no matter how complex the algorithm is. *Notice that for searching through a list of values, no matter how large, the best case will result in a constant time of 1*. Because the best case for an algorithm will usually be a very small and frequently constant value, we will not do a best-case analysis very frequently.
**(2) Worst Case:** *Worst case is an important analysis because it gives us an idea of the most time an algorithm will ever take*. Worst-case analysis requires that we identify the ***input*** values that cause an algorithm to do the ***most work***. *For searching algorithms, the worst case is one where the value is in the last place we check or is not in the list*. This could involve comparing the key to each list value for a total of N comparisons. The worst case gives us an ***upper bound*** on how slowly parts of our programs may work based on our algorithm choices.
**(3) Average Case:** Average-case analysis is the toughest to do because there are a lot of details involved. The *basic* process begins by determining the number of different groups into which all possible input sets can be divided. The *second step* is to determine the probability that the input will come from each of these groups. The *third step* is to determine how long the algorithm will run for each of these groups. All of the input in each group should take the same amount of time, and if they do not, the group must be split into two separate groups. When all of this has been done, the average case time is given by the following formula:

$$A(n) = \sum_{i=1}^{m} p_i * t_i \qquad (1.1)$$

where $n$ is the size of the input, $m$ is the number of groups, $p_i$ is the probability that the input will be from group $i$, and $t_i$ is the time that the algorithm takes for input from group $i$. In some cases, we will consider that each of the input groups has equal probabilities. *In other words, if there are five input groups, the chance the input will be in group 1 is the same as the chance for group 2, and so on*. This would mean that for these five groups all probabilities would be 0.2. We could use the following simplified formula where all groups are equally probable:

$$A(n) = \frac{1}{m}\sum_{i=1}^{m} t_i \qquad (1.2)$$

### 1.4 Rate of growth

In analysis of algorithms, it is not important to know exactly how many operations an algorithm does. *Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to as the rate of growth of the algorithm.* What happens with small sets of input data is not as interesting as what happens when the data set gets large. The graph in Figure 1.1 shows the rate of growth of four functions.

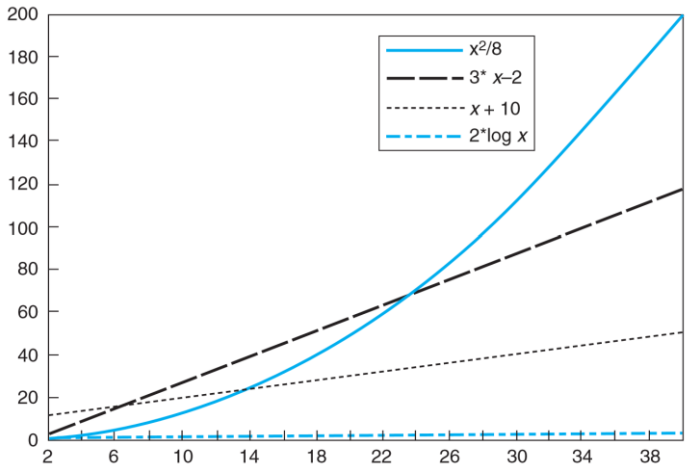


Figure 1.1: Rate of growth of four functions

If we look closely at the graph in Figure 1.1, we will see some trends. The function based on $\boldsymbol{x^2}$ increases slowly at first, but as the problem size gets larger, it begins to grow at a rapid rate. The functions that are based on $\boldsymbol{x}$ both grow at a steady rate for the entire length of the graph. The function based on ***log x*** seems to not grow at all, but this is because it is actually growing at a very slow rate. The relative height of the functions is also different when we have small values versus large ones. *Consider the value of the functions* when $\boldsymbol{x}$ is 2. At that point, the function with the smallest value is $\boldsymbol{x^2 / 8}$ and the one with the largest value is $\boldsymbol{x + 10}$. *We can see that as the value of* $\boldsymbol{x}$ *gets large,* $\boldsymbol{x^2 / 8}$ *becomes and stays the function with the largest value*. Putting all of this together means that as we analyze algorithms, *we will be interested in which rate of growth class an algorithm falls into rather than trying to find out exactly how many of each operation are done by the algorithm*. When we consider the relative "size" of a function, we will do so for large values of $\boldsymbol{x}$, not small ones.

Figure 1.2 shows some common classes of algorithm growth. If we determine that an algorithm's complexity is a combination of two of these classes, *we will frequently ignore all*

*but the fastest growing of these terms*. For example, if we analyze an algorithm and find that it does $x^3 - 30x$ comparisons, we will just refer to this algorithm as growing at the rate of $x^3$. This is because even at an input size of just 100 the difference between $x^3$ and $x3 - 30x$ is only 0.3%.

| | lg *n* | *n* | *n* lg *n* | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| 1 | 0.0 | 1.0 | 0.0 | 1.0 | 1.0 | 2.0 |
| 2 | 1.0 | 2.0 | 2.0 | 4.0 | 8.0 | 4.0 |
| 5 | 2.3 | 5.0 | 11.6 | 25.0 | 125.0 | 32.0 |
| 10 | 3.3 | 10.0 | 33.2 | 100.0 | 1000.0 | 1024.0 |
| 15 | 3.9 | 15.0 | 58.6 | 225.0 | 3375.0 | 32768.0 |
| 20 | 4.3 | 20.0 | 86.4 | 400.0 | 8000.0 | 1048576.0 |
| 30 | 4.9 | 30.0 | 147.2 | 900.0 | 27000.0 | 1073741824.0 |
| 40 | 5.3 | 40.0 | 212.9 | 1600.0 | 64000.0 | 1099511627776.0 |
| 50 | 5.6 | 50.0 | 282.2 | 2500.0 | 125000.0 | 1125899906842620.0 |
| 60 | 5.9 | 60.0 | 354.4 | 3600.0 | 216000.0 | 1152921504606850000.0 |
| 70 | 6.1 | 70.0 | 429.0 | 4900.0 | 343000.0 | 1180591620717410000000.0 |
| 80 | 6.3 | 80.0 | 505.8 | 6400.0 | 512000.0 | 1208925819614630000000000.0 |
| 90 | 6.5 | 90.0 | 584.3 | 8100.0 | 729000.0 | 1237940039285380000000000000.0 |
| 100 | 6.6 | 100.0 | 664.4 | 10000.0 | 1000000.0 | 1267650600228230000000000000000.0 |

Figure 1.2: Some common algorithm classes

**Note:** $\lg n = log 2^n$ = *In(n)/In(2).* E.g *lg 80 = In(80)/In(2)* = 4.382/0.693 = 6.32

### 1.5 Classification of Growth

Having seen that the rate of growth of an algorithm is important, and that the rate of growth is dominated by the ***largest*** term in an equation, *we will discard the terms that grow more slowly*. From this, we are left with what we call the ***order of the function***. We can then group algorithms together based on their order. *We group them in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster*.

*It is important to note that f(n) is a function that represents the growth rate of algorithm already on ground and g(n) is the function that represents the new algorithm we just developed. Our goal is to find out is the new algorithm g(n) is better than f(n) that is already in existence.*

**Big Omega**

We use Ω*(f)*, called **big omega**, to *represent the class of functions that grow at least as fast as the function f.* This means that for all values of *n* greater than some threshold $n_0$, *all of the functions in* Ω*(f) have values that are at least as large as f.* You can view Ω(f) as setting a lower bound on a function, because all the functions in this class will grow as fast as *f* or even faster. Formally, this means that if $g(x) \in \Omega(f)$, $g(n) \geq cf(n)$ for all $n \geq n_0$ (where *c* is a positive constant). *Because we are interested in efficiency,* Ω*(f) will not be of much interest to us because* $\Omega(n^2)$*, for example, includes all functions that grow faster than* $n^2$ *including* $n^3$ *and* $2^n$.

**Big Oh**

Another category is *O(f)*, called **big oh**, which *represents the class of functions that grow no faster than f.* This means that for all values of *n* greater than some threshold $n_0$, *all of the functions in O(f) have values that are no greater than f.* The class *O(f)* has f as an upper bound, so none of the functions in this class grow faster than f. Formally this means that if

$g(x) \in O(f)$, $g(n) \leq cf(n)$ for all $n \geq n_0$ (where c is a positive constant). ***This is the class that will be of the greatest interest to us***. Considering two algorithms, we will want to know if the function categorizing the behavior of the first is in big oh of the second. If so, we know that the second algorithm does no better than the first in solving the problem.

**Big Theta**

We use θ(f), called **big theta**, to *represent the class of functions that grow at the same rate as the function f*. This means that for all values of *n* greater than some threshold $n_0$, *all of the functions in θ(f) have values that are about the same as f*. Formally, this class of functions is defined as the place where big omega and big oh overlap, so $\theta(f) = \Omega(f) \cap O(f)$. When we consider algorithms, we will be interested in finding algorithms that might do better than the one we are considering. So, finding one that is in big theta (in other words, is of the same complexity) is not very interesting.

**Notation**

Because θ(f), Ω(f), and O(f) are sets, it is proper to say that a function ***g*** is an element of these sets. The analysis literature, however, accepts that a function ***g*** is equal to these sets as being equivalent to being a member of the set. So, when you see $g = O(f)$, this really means that $g \in O(f)$.

**Complexity Classes**

Some of the most common instances of Big Oh are listed below. In each case, ***n*** is a measure of the size of the inputs to the function.

- O(1) denotes **constant** running time.
- O(log n) denotes **logarithmic** running time.
- O(n) denotes **linear** running time.
- O(n log n) denotes **log-linear** running time.
- $O(n^k)$ denotes **polynomial** running time. Notice that k is a constant. E.g $n^2$, $n^3$, $n^4$ etc
- $O(c^n)$ denotes exponential running time. Notice that a constant ***c*** is being raised to a power based on the size of the input (***n***).

## 1.6 Mathematical Background

### (a) Floor and Ceiling

The floor and ceiling will be used when we need to determine *how many times something is done*, and the value depends on some fraction of the items it is done to. We say that the floor of X (written $\lfloor X \rfloor$) is the *largest integer that is less than or equal to X*. So, $\lfloor 2.5 \rfloor$ would be 2 and $\lfloor -7.3 \rfloor$ would be -8. We say that the ceiling of X (written $\lceil X \rceil$) is the *smallest integer that is greater than or equal to X*. So, $\lceil 2.5 \rceil$ would be 3 and $\lceil -7.3 \rceil$ would be -7. For positive numbers, you can think of the floor as truncation and the ceiling as rounding up. For negative numbers, the effect is reversed.

For example, if we compare a set of N values in pairs, where the first value is compared to the second, the third to the fourth, and so on, the number of comparisons will be $\lfloor N/2 \rfloor$. If N is 10, we will do five comparisons of pairs and $\lfloor 10/2 \rfloor = \lfloor 5 \rfloor = 5$. If N is 11, we will still do five comparisons of pairs and $\lfloor 11/2 \rfloor = \lfloor 5.5 \rfloor = 5$.

### (b) Logarithms

Logarithms will play an important role in our analysis, there are a few properties that must be discussed. The logarithm base *y* of a number *x* is the power of *y* that will produce the number

*x. So, the* $\log_{10}$ *45 is about 1.653 because* $10^{1.653}$ *is 45*. The base of a logarithm can be any number, but we will typically use either base 10 or base 2 in our analysis. ***We will use log as shorthand for*** $\log_{10}$ ***and lg as shorthand for*** $\log_2$. *Logarithms are a strictly increasing function*. This means that given two numbers X and Y, if X > Y, $\log_B X > \log_B Y$ for all bases B. *Logarithms are one-to-one functions*. This means that if $\log_B X = \log_B Y$, for X = Y. Other important properties are:

$$\log_B 1 = 0 \quad \textbf{(1.3)}$$

$$\log_B B = 1 \quad \textbf{(1.4)}$$

$$\log_B(X * Y) = \log_B X + \log_B Y \quad \textbf{(1.5)}$$

$$\log_B X^Y = Y * \log_B X \quad \textbf{(1.6)}$$

$$\log_A X = \frac{(\log_B X)}{(\log_B A)} \quad \textbf{(1.7)}$$

These properties can be combined to help simplify a function. *Equation 1.7* is used for base conversion. Most calculators perform $\log_{10}$ and natural logs, but let's say you need to know $\log_{42}$ 75. Equation 1.7 would help you find the answer of 1.155 (e.g $\log_{10}75/\log_{10}42$).

**(c) Binary Tree**

*A binary tree is a structure in which each node in the tree is said to have at most two nodes as its children, and each node has exactly one parent node*. The top node in the tree is the only one without a parent node and is called the *root* of the tree. A binary tree that has N nodes has at least $\lfloor \lg N \rfloor + 1$ levels to the tree if the nodes are packed as tightly as possible. *For example,* a full binary tree with 15 nodes has one root, two nodes on the second level, four nodes on the third level, eight nodes on the fourth level, and our equation gives $\lfloor \lg 15 \rfloor + 1 = \lfloor 3.9 \rfloor + 1 = 4$ levels. Notice, if we add one more node to this tree, it has to start a new level and now $\lfloor \lg 16 \rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$ levels. The largest binary tree that has N nodes will have N levels if each node has exactly one child (in which case the tree is actually a list). *If we number the levels of the tree,* ***considering the root to be on level 1, there are*** $2^{K-1}$ ***nodes on level K***. A complete binary tree with J levels (numbered from 1 to J) is one where all of the leaves in the tree are on level J, and all nodes on levels 1 to J - 1 have exactly two children. ***A complete binary tree with J levels has*** $2^J - 1$ ***nodes (i.e total nodes)***. This information will be useful in a number of the analyses we will do.

*Summarily, for binary tree that consider root at level 1, we have*:

- $\lfloor \lg N \rfloor + 1$ levels
- $2^{K-1}$ nodes on level K
- $2^J - 1$ *total nodes*

To better understand these formulas, you might want to draw some binary trees and compare your results with that of the formulas.

**(d) Probabilities**

Because we will analyze algorithms relative to their input, we may at times need to consider the likelihood of a certain set of input. This means that we will need to work with the probability that the input will meet some condition. *The probability that something will occur is given as a number in the range of 0 to 1, where 0 means it will never occur and 1 means it will always occur*. If we know that there are exactly 10 different possible inputs, we can say that the probability of each of these is between 0 and 1 and that the total of all of the individual probabilities is 1, because one of these must happen. If there is an equal chance that any of these can occur, each will have a probability of 0.1 (one out of 10, or 1/10).

For most of our analyses, we will first determine how many possible situations or input groups there are and then assume that all are equally likely. If we determine that there are N possible situations, this results in a probability of 1 / N for each of these situations or input groups.

**(e) Summations**

We will be adding up sets of values as we analyze our algorithms. *Let's say we have an algorithm with a loop*. We notice that when the loop variable is 5, we do 5 steps and when it is 20, we do 20 steps. We determine in general that when the loop variable is M, we do M steps. Overall, the loop variable will take on all values from 1 to N, so the total steps is the sum of the values from 1 through N. To easily express this, we use the equation $\sum_{i=1}^{N} i$. The expression below the $\boldsymbol{\Sigma}$ represents the initial value for the summation variable, and the value above the $\boldsymbol{\Sigma}$ represents the ending value. You should see how this expression corresponds to the sum we are looking for. Once we have expressed some solution in terms of this summation notation, we will want to simplify this so that we can make comparisons with other formulas. Deciding whether $\sum_{i=11}^{N}(i^2 - i)$ or $\sum_{i=0}^{N}(i^2 - 20i)$ would be difficult to do by inspection, so we use the following set of standard summation formulas to determine the actual values these summations represent.

$$\sum_{i=1}^{N} C * i = C * \sum_{i=1}^{N} i, \text{ with } C \text{ a constant expression not dependent on } i \quad \textbf{(1.8)}$$

$$\sum_{i=C}^{N} i = \sum_{i=0}^{N-C} (i + C) \quad \textbf{(1.9)}$$

$$\sum_{i=C}^{N} i = \sum_{i=0}^{N} i - \sum_{i=0}^{C-1} i \quad \textbf{(1.10)}$$

$$\sum_{i=1}^{N} (A + B) = \sum_{i=1}^{N} A + \sum_{i=1}^{N} B \quad \textbf{(1.11)}$$

$$\sum_{i=0}^{N} (N - i) = \sum_{i=0}^{N} i \quad \textbf{(1.12)}$$

Equation 1.12 just shows that adding the numbers from N down to 0 is the same as adding the numbers from 0 up to N. In some cases, it will be easier to solve equations if we can apply Equation 1.12.

$$\sum_{i=1}^{N} 1 = N \tag{1.13}$$

$$\sum_{i=1}^{N} C = C * N \tag{1.14}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \tag{1.15}$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} = \frac{2N^3 + 3N^2 + N}{6} \tag{1.16}$$

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \tag{1.17}$$

Equation 1.17 is easy to remember if you consider binary numbers. *When you add the powers of 2 from 0 to 10*, this is the same as the binary number 11111111111. If we add 1 to this number, we get 100000000000, which is $2^{11}$. But because we added 1 to it, it is 1 larger than the sum of the powers of 2 from 0 to 10, so the sum must be $2^{11}$ - 1. If we now substitute N for 10, we get Equation 1.17.

**Note that** $\sum_{i=0}^{N} i = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

$$\sum_{i=1}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}, \quad \text{for some number } A \tag{1.18}$$

$$\sum_{i=1}^{N} i2^i = (N-1)2^{N+1} + 2 \tag{1.19}$$

$$\sum_{i=1}^{N} \frac{1}{i} = \ln N \tag{1.20}$$

$$\sum_{i=1}^{N} \lg i \approx N \lg N - 1.5 \tag{1.21}$$

When we are trying to simplify a summation equation, we can apply Equations 1.8 through 1.12 to break down the equation into simpler parts and then apply the rest to get an equation without summations.

**Also note**:

$$\sum_{i=C}^{N} 1 = N - C + 1 \tag{1.22}$$

**Examples:**
(1) The algorithm below will take $\sum_{i=1}^{N} 1 = N = O(n)$, ***linear running time***, since only one constant operation is performed for each loop variable (*see Equation 1.13*). Note that 1 constant operation is performed inside the loop for each loop variable.

```
For I = 1 to N Do
      Print I
End For
```

(2) The algorithm below will take $\sum_{i=1}^{N} 3 = 3 * N = O(n)$, ***linear running time***, since only three constant operations are performed for each loop variable (*see Equation 1.14*).

```
For I = 1 to N Do
      Print I
      Print I +2
      Print I + 3
End For
```

(3) The algorithm below will take $\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = O(n^2)$, ***polynomial running time***, since we discovered that when *i* is 1, 1 operation is done, when *i* is 2, 2 operations are performed and when *i* is *N*, *N* operations are performed (*see Equation 1.15*).

```
For I = 1 to N Do
    For J = 1 to I Do
        Print I
    End For
End For
```

**Solution**
Looking at the algorithm loops, we have the following summations. Note that 1 constant operation is performed at the innermost loop.

$$\sum_{i=1}^{N} \sum_{j=1}^{I} 1$$

Using Equation 1.13, we have, since $\sum_{j=1}^{I} 1 = 1 * i = i$

$$\sum_{i=1}^{N} i$$

Using Equation 1.15, we have $\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = (N^2 + N)/2 = O(n^2)$.

(4) The algorithm below will take $O(n^4)$, ***polynomial running time***.

```
Sum = 0
For I = 1 to n*n Do
        For J = 1 to I Do
                For K = 1 to 6 Do
```

```
                Sum++
            End For
        End For
End For
```

**Solution**

Looking at the algorithm loops, we have the following summations. Note that 1 constant operation is performed at the innermost loop.

$$\sum_{i=1}^{n^2}\sum_{j=1}^{i}\sum_{k=1}^{6} 1$$

Using Equation 1.13, we have:

$$\sum_{i=1}^{n^2}\sum_{j=1}^{i} 6$$

Using Equation 1.14, we have:

$$\sum_{i=1}^{n^2} 6i$$

Using Equation 1.8, we have:

$$6 * \sum_{i=1}^{n^2} i$$

Using Equation 1.15 where $N = n^2$, we have:

$$6 \quad \frac{n^2(n^2+1)}{2} = 3(n^4 + n^2) = 3n^4 + 3n^2 = O(n^4)$$

(5) For the following algorithm, compute the worst-case asymptotic complexity (as a function of n), where 'loop body' is a constant number of lines.

```
For (i=0; i<=n-1; i++) Do
        For (j=i+1; j<=n-1; j++) Do
                //loop body
        End For
End For
```

**Solution**

Looking at the algorithm loops, we have:

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

Using Equation 1.22, where *n=n-1* and *C = i+1*, we have:

$$\sum_{i=0}^{n-1}((n-1)-(i+1)+1) = \sum_{i=0}^{n-1}(n-1-i-1+1) = \sum_{i=0}^{n-1}(n-1-i)$$

Applying Equation 1.11, we have:

$$\sum_{i=0}^{n-1}(n-1-i)=\sum_{i=0}^{n-1}n-\sum_{i=0}^{n-1}1-\sum_{i=0}^{n-1}i$$

Since, $\sum_{i=0}^{N} i = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$, then, using Equation 1.14,

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

Using Equation 1.13,

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

Using Equation 1.15,

$$\sum_{i=0}^{n-1}i=\frac{(n-1)\big((n-1)+1\big)}{2}=\frac{(n-1)(n-1+1)}{2}=\frac{(n-1)(n)}{2}=\frac{n^2-n}{2}$$

Combining these solutions, we have,

$$\sum_{i=0}^{n-1}(n-1-i)=(n^2-n)-(n-1)-\left(\frac{n^2-n}{2}\right)=n^2-n-n+1-\left(\frac{n^2-n}{2}\right)$$

$$\sum_{i=0}^{n-1}(n-1-i)=n^2-2n+1-\left(\frac{n^2-n}{2}\right)$$

$$\sum_{i=0}^{n-1}(n-1-i)=n^2-2n+1-\frac{n^2+n}{2}=\frac{n^2-2n+1}{1}-\frac{n^2+n}{2}$$

$$\sum_{i=0}^{n-1}(n-1-i)=\frac{2n^2-4n+2-n^2+n}{2}=O(n^2)$$

## 1.7 Divide and Conquer Algorithms

*Divide and conquer algorithms can provide a small and powerful means to solve a problem; this section is not about how to write such an algorithm but rather how to analyze one*. When we count comparisons that occur in loops, we only need to determine how many comparisons there are inside the loop and how many times the loop is executed. *This is made more complex when a value of the outer loop influences the number of passes of an inner loop*. ***When we look at divide and conquer algorithms, it is not clear how many times a task will be done because it depends on the recursive calls and perhaps on some preparatory and concluding work***. It is usually not obvious how many times the function will be called recursively. *As an example of this, consider the following generic divide and conquer algorithm*:

```
DivideAndConquer(data, N, solution )
data     a set of input values
N        the number of values in the set
solution the solution to this problem
if (N ≤ SizeLimit) then
        DirectSolution(data, N, solution )
else
        DivideInput(data, N, smallerSets, smallerSizes, numberSmaller )

        for i = 1 to numberSmaller do
                DivideAndConquer(smallerSets[i], smallerSizes[i], smallSolution[i])
        end for
        CombineSolutions(smallSolution, numberSmaller, solution)
end if
```

This algorithm will first check to see *if the problem size is small enough* to determine a solution by some simple *nonrecursive* algorithm (called ***DirectSolution***) and, if so, will do that. *If the problem is too large*, it will first call the routine ***DivideInput***, which will partition the input in some fashion into a number (***numberSmaller***) of smaller sets of input values. *These smaller sets may be all of the same size or they may have radically different sizes*. The elements in the original input set will all be put into at least one of the smaller sets, but values can be put in more than one. *Each of these smaller sets will have fewer elements than the original input set*. The ***DivideAndConquer*** algorithm *is then called recursively* for each of these smaller input sets, and the results from those calls are put together by the ***CombineSolutions*** function.

**Example**

The factorial of a number can easily be calculated by a loop, *but for the purpose of this example, we consider a recursive version*. You can see that the factorial of the number N is just the number N times the factorial of the number N - 1. This leads to the following algorithm:

```
Factorial(N)
N        is the number we want the factorial for
Factorial returns an integer
If (N = 1) then
        return 1
```

```
else
        smaller = N - 1
        answer = Factorial(smaller)
        return (N * answer)
end if
```

The recursive form of the factorial algorithm is written in simple detailed steps so that we can match things up with the standard recursive algorithm above. Even though earlier in this module, we discussed how multiplications are more complex than additions and are, therefore, counted separately, to simplify this example we are going to count them together. In matching up the two algorithms, we see that the *size limit* in this case is 1, and our *direct solution* is to return the answer of 1, which takes no mathematical operations. In all other cases, we use the else clause. The first step in the general algorithm is to "*divide the input*" into smaller sizes, and in the factorial function that is the calculation of *smaller*, which takes one subtraction. The next step in the general algorithm is to make the recursive calls with these smaller problems, and in the factorial function there is *one recursive call with a problem size that is 1 smaller than the original*. The last step in the general algorithm is to *combine the solutions*, and *in the factorial function that is the multiplication in the last return statement*.

**Determining Recursive Algorithm Efficiency**

How efficient is a recursive algorithm? Would it make it any easier if you knew that the direct solution is ***quadratic***, the division of the input is ***logarithmic***, and the combination of the solutions is ***linear***, all with respect to the size of the input, and that the input set is broken up into ***eight*** *pieces all one quarter of the original*? This is probably not a problem for which you can quickly find an answer or for that matter are even sure where to start. It turns out, however, that the process of analyzing any divide and conquer algorithm is very straightforward if you can map the steps of your algorithm into the four steps *shown in the generic algorithm above: a direct solution, division of the input, number of recursive calls, and combination of the solutions*. Once you know how each piece relates to the others, and you know how complex each piece is, *you can use the following formula to determine the complexity of the divide and conquer algorithm*:

$$\mathrm{DAC}(N) = \begin{cases} \mathrm{DIR}(N) & \text{for } N \le \mathrm{SizeLimit} \\ \mathrm{DIV}(N) + \sum_{i=1}^{\mathrm{numberSmaller}} \mathrm{DAC}(\mathrm{smallerSizes}[i]) + \mathrm{COM}(N) & \text{for } N > \mathrm{SizeLimit} \end{cases}$$

where DAC is the complexity of DivideAndConquer
DIR is the complexity of DirectSolution
DIV is the complexity of DivideInput
COM is the complexity of CombineSolutions

To answer the question raised earlier where the direct solution is quadratic, the division of the input is logarithmic, and the combination of the solutions is linear, all with respect to the size of the input, and that the input set is broken up into eight pieces all one quarter of the original. Putting all of this in the formula to determine the complexity of the divide and conquer algorithm, then we have:

$$\text{DAC}(N) = \begin{cases} N^2 & \text{for } N \le \text{SizeLimit} \\ \lg N + \sum_{i=1}^{8} \text{DAC}(N/4) + N & \text{for } N > \text{SizeLimit} \end{cases}$$

Or in a simplify form since every smaller sets are of the same size:

$$\text{DAC}(N) = \begin{cases} N^2 & \text{for } N \le \text{SizeLimit} \\ \lg N + 8\ \text{DAC}(N/4) + N & \text{for } N > \text{SizeLimit} \end{cases}$$

This form of equation is called ***a recurrence relation*** because the value of the function is based on itself. We prefer to have our equations in a form that is dependent only on ***N*** and not other function calls. *To arrive at such equation, we need to solve the recurrence relation equation*. This is cover in the subsequent section.

Let us return to our factorial example. We identified all of the elements in the factorial algorithm relative to the generic DivideAndConquer. We now use that identification to decide what values get put into the general equation above. *For the Factorial function, we said that the direct solution does no calculations, the input division and result combination steps do one calculation each, and the recursive call works with a problem size that is one smaller than the original*. This results in the following recurrence relation for the number of calculations in the Factorial function:

$$\text{Calc}(N) = \begin{cases} 0 & \text{for } N = 1 \\ 1 + \text{Calc}(N-1) + 1 & \text{for } N > 1 \end{cases}$$

**1.8 Tournament Method**

The tournament method is based on recursion and can be used to solve a number of different problems *where information from a first pass through the data can help to make later passes more efficient*. *If we use it to find the largest value, this method involves building a binary tree with all of the elements in the leaves*. At each level, two elements are paired and the larger of the two gets copied into the parent node. This process continues until the root node is reached. Figure 1.3 shows a complete tournament tree for a given set of data. *Assuming we are to develop an algorithm to find the second largest element in a list using about N comparisons. The tournament method helps us do this*. Every comparison produces one "winner" and one "loser." The losers are eliminated and only the winners move up in the tree. Each element, except for the largest, must "lose" one comparison. Therefore, *building the tournament tree will take* ***N - 1*** comparisons. The second largest element could only have lost to the largest element. We go down the tree and get the set of elements that lost to the largest one. We know that there can be at most $\lceil \lg N \rceil$ of these elements that lost to the largest. There will be $\lceil \mathbf{lg}\, \boldsymbol{N} \rceil$ comparisons to find these elements in the tree and $\lceil \mathbf{lg}\, \boldsymbol{N} \rceil - \mathbf{1}$ comparisons to find the largest in this collection. The entire process takes:

N -1 + $\lceil \lg N \rceil$ + $\lceil \lg N \rceil - 1$ = N + 2$\lceil \lg N \rceil - 2$ comparisons, which is ***O(n)***.

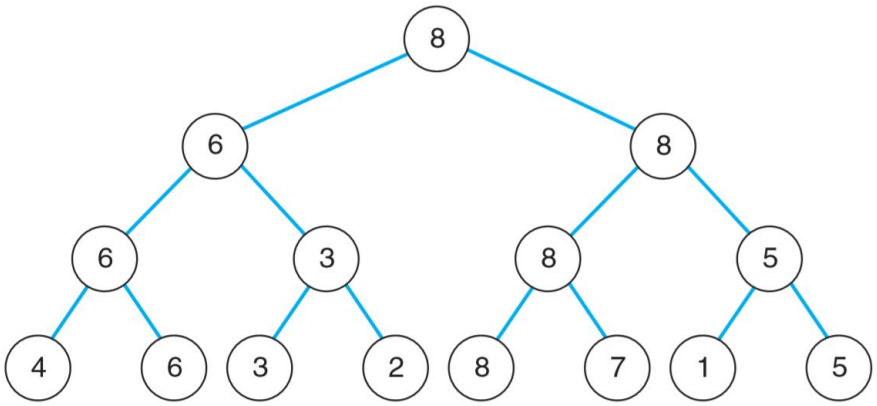


Figure 1.3: Tournament tree for a set of eight values

## 1.9 Recurrence Relation

*Recurrence relations can be directly derived from a recursive algorithm, but they are in a form that does not allow us to quickly determine how efficient the algorithm is*. To do that we need to convert the set of recursive equations into what is called ***closed form*** by removing the recursive nature of the equations. ***This is done by a series of repeated substitutions until we can see the pattern that develops by the recursive nature***. A recurrence relation can be expressed in two ways. The first is used if there are *just a few simple cases* for the formula:

$$T(n) = 2T(n-2) - 15$$
$$T(2) = 40$$
$$T(1) = 40$$

The second is used if the *direct solution is applied for a larger number of cases*:

$$T(n) = \begin{cases} 4 & \text{if } n \leq 4 \\ 4T(n/2) - 1 & \text{otherwise} \end{cases}$$

These forms are equivalent. ***We can convert from the second form to the first by just listing those values for which we have the direct answer***. This means that the second recurrence relation above could also be given as:

$$T(n) = 4T(n/2) - 1$$
$$T(4) = 4$$
$$T(3) = 4$$
$$T(2) = 4$$
$$T(1) = 4$$

**Example**

(1) Consider the following recurrence relation:

$$T(n) = 2T(n-2) - 15$$
$$T(2) = 40$$
$$T(1) = 40$$

We will want to substitute an equivalent value for *T(n - 2)* back into the first equation. To do so, we replace every n in the first equation with *n - 2*, giving:

$$T(n-2) = 2T(n-2-2) - 15$$
$$= 2T(n-4) - 15$$

But now we can see when this substitution is done, we will still have *T(n - 4)* to eliminate. If you think ahead, you will realize that there will be a series of these values that we will need. As a first step, we create a set of these equations for successively smaller values:

$$T(n-2) = 2T(n-4) - 15$$
$$T(n-4) = 2T(n-6) - 15$$
$$T(n-6) = 2T(n-8) - 15$$
$$T(n-8) = 2T(n-10) - 15$$
$$T(n-10) = 2T(n-12) - 15$$

Now we begin to substitute back into the original equation. We will be careful not to simplify the resulting equation too much because that will make the pattern more difficult to see. Doing the substitution gives us:

$$T(n) = 2T(n-2) - 15 = 2(2T(n-4) - 15) - 15$$
$$T(n) = 4T(n-4) - 2 * 15 - 15$$

$$T(n) = 4(2T(n-6) - 15) - 2 * 15 - 15$$
$$T(n) = 8T(n-6) - 4 * 15 - 2 * 15 - 15$$

$$T(n) = 8(2T(n-8) - 15) - 4 * 15 - 2 * 15 - 15$$
$$T(n) = 16T(n-8) - 8 * 15 - 4 * 15 - 2 * 15 - 15$$

$$T(n) = 16(2T(n-10) - 15) - 8 * 15 - 4 * 15 - 2 * 15 - 15$$
$$T(n) = 32T(n-10) - 16 * 15 - 8 * 15 - 4 * 15 - 2 * 15 - 15$$

$$T(n) = 32(2T(n-12) - 15) - 16 * 15 - 8 * 15 - 4 * 15 - 2 * 15 - 15$$
$$T(n) = 64T(n-12) - 32 * 15 - 16 * 15 - 8 * 15 - 4 * 15 - 2 * 15 - 15$$

You are probably beginning to see a pattern develop here. **First**, we notice that each new term at the end of the equation is -15 multiplied by the next higher power of 2. **Second**, we notice that the coefficient of the recursive call to T is going through a series of powers of 2. **Third**, we notice that the value that we are calling T with keeps going down by 2 each time. Now, you might wonder when does this process end? If we look back at the original equation,

you will see that we have a fixed value for ***T(1)*** and ***T(2)***. How many times would we have to substitute back into this equation to get to either of these values? We can see that 2 = n - (n - 2) if n is even. This seems to indicate that we would substitute back into this equation **[(n - 2)/2] - 1 times** giving **n/2 - 1 terms** based on -15 in the equation, and the power of the coefficient of T will be **n/2 – 1** if n is even i.e for *T(2)*. If n is odd, such as the case of T(1), we will have **n/2** terms instead (because 13/2 -1 is 5 and not 6. And we need 6 terms. That is why we use n/2 for odd). To see this, consider what we would have if the value of **n** was 14. Remember that we have **six** terms in the final equation and since 14 is even, 14-12 will be 2 (i.e T(2) ) and 14/2 -1 terms is exactly 6 terms. For the second case, when n is odd, consider n to be 13. Then, we have 13-12 will be 1 (i.e T(1) ) and 13/2 terms is exactly 6 terms. *This means that we look for valid case that represent our last equation and that can be generalized to other situations*. Therefore, our two cases are valid for even and odd and they are given below:

$$T(n) = 2^{(n/2)-1}T(2) - 15\sum_{i=0}^{(n/2)-1} 2^i \quad \text{if } n \text{ is even} \qquad T(n) = 2^{n/2}T(1) - 15\sum_{i=0}^{n/2} 2^i \quad \text{if } n \text{ is odd}$$

$$T(n) = 2^{(n/2)-1} * 40 - 15\sum_{i=0}^{(n/2)-1} 2^i \quad \text{if } n \text{ is even} \qquad T(n) = 2^{n/2} * 40 - 15\sum_{i=0}^{n/2} 2^i \quad \text{if } n \text{ is odd}$$

Now, applying Equation 1.17, for an even value of n we get:

$$\begin{aligned} T(n) &= 2^{(n/2)-1} * 40 - 15 * (2^{n/2} - 1) \\ &= 2^{n/2} * 20 - 2^{n/2} * 15 + 15 \\ &= 2^{n/2}(20 - 15) + 15 \\ &= 2^{n/2} * 5 + 15 \end{aligned}$$

$= O(2^{n/2})$, exponential running time

and, if n is odd, we get:

$$\begin{aligned} T(n) &= 2^{n/2} * 40 - 15 * (2^{(n/2)+1} - 1) \\ &= 2^{n/2} * 40 - 2^{n/2} * 30 + 15 \\ &= 2^{n/2}(40 - 30) + 15 \\ &= 2^{n/2} * 10 + 15 \end{aligned}$$

$= O(2^{n/2})$, exponential running time

(2) Consider the recurrence relation:

$$T(n) = \begin{cases} 5 & \text{if } n \leq 4 \\ 4T(n/2) - 1 & \text{otherwise} \end{cases}$$

We will proceed in the same way as we did in the previous example. We first substitute in a set of values for n, only in this case, ***because n is being divided by 2, each of the subsequent equations will have half the value***. This gives us the equations:

$$
\begin{aligned}
T(n/2) &= 4T(n/4) - 1 \\
T(n/4) &= 4T(n/8) - 1 \\
T(n/8) &= 4T(n/16) - 1 \\
T(n/16) &= 4T(n/32) - 1 \\
T(n/32) &= 4T(n/64) - 1
\end{aligned}
$$

Now, we again substitute back into the original giving the following series of equations:

$$
\begin{aligned}
T(n) &= 4T(n/2) - 1 = 4(4T(n/4) - 1) - 1 \\
T(n) &= 16T(n/4) - 4 * 1 - 1
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= 16(4T(n/8) - 1) - 4 * 1 - 1 \\
T(n) &= 64T(n/8) - 16 * 1 - 4 * 1 - 1
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= 64(4T(n/16) - 1) - 16 * 1 - 4 * 1 - 1 \\
T(n) &= 256T(n/16) - 64 * 1 - 16 * 1 - 4 * 1 - 1
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= 256(4T(n/32) - 1) - 64 * 1 - 16 * 1 - 4 * 1 - 1 \\
T(n) &= 1024T(n/32) - 256 * 1 - 64 * 1 - 16 * 1 - 4 * 1 - 1
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= 1024(4T(n/64) - 1) - 256 * 1 - 64 * 1 - 16 * 1 - 4 * 1 - 1 \\
T(n) &= 4096T(n/64) - 1024 * 1 - 256 * 1 - 64 * 1 - 16 * 1 - 4 * 1 - 1
\end{aligned}
$$

We notice that the coefficient of -1 increases by a power of 4 each time we substitute, and it is the case that the power of 2 that we divide n by is 1 greater than the largest power of 4 for this coefficient. Also, we notice that the coefficient of T is the same power of 4 as the power of 2 that we divide n by. When we have $\boldsymbol{T(n/2^i)}$, its coefficient will be $\mathbf{4^i}$. Now, for what value of *i* can we stop the substitution? Well, because we have the direct case specified for $n \leq 4$, we can stop when we get to $\boldsymbol{T(4) = T(n/2^{lgn\,-\,2})}$. Putting this together we get (the *lgn-3* we help to operate at the same level with *lgn-2* after substitution in eqn. 1.18):

$$
T(n) = 4^{\lg n - 2} T(4) - \sum_{i=0}^{\lg n - 3} 4^i
$$

Using the value for the direct case, and Equation 1.18, we get:

$$T(n) = 4^{\lg n-2} * 5 - \frac{4^{\lg n-2} - 1}{4 - 1}$$

$$T(n) = 4^{\lg n-2} * 5 - \frac{4^{\lg n-2} - 1}{3}$$

$$T(n) = \frac{15 * 4^{\lg n-2} - 4^{\lg n-2} + 1}{3}$$

$$T(n) = \frac{4^{\lg n-2}(15 - 1) + 1}{3}$$

$$T(n) = \frac{4^{\lg n-2} * 14 + 1}{3}$$

$= O(4^{lgn-2})$, exponential running time

As you can see, the ***closed form*** of a recurrence relation may not be simple or neat, however, it does eliminate the recursive "call" so that we can quickly compare equations and determine their order.