G.PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY

LECTURE NOTES ON
DESIGN AND ANALYSIS OF ALGORITHMS

Department of Computer Science and Engineering

UNIT 1

Basic Concepts

Algorithm

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs areentered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

Algorithm Design Goals

The three basic design goals that one should strive for in a program are:

1. Try to save Time
2. Try to save Space
3. Try to save Face

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to "save face" by preventing the program from locking up or generating reams of garbled data.

Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

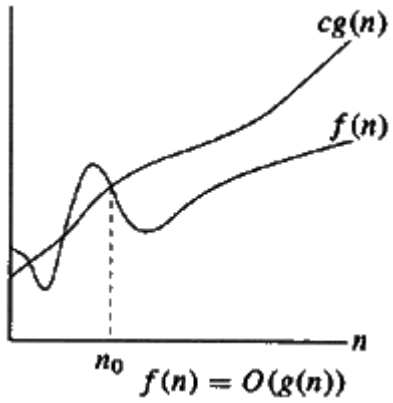
Rate of Growth:

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O)[1],
2. Big–OMEGA (Ω),
3. Big–THETA (θ) and
4. Little–OH (o)

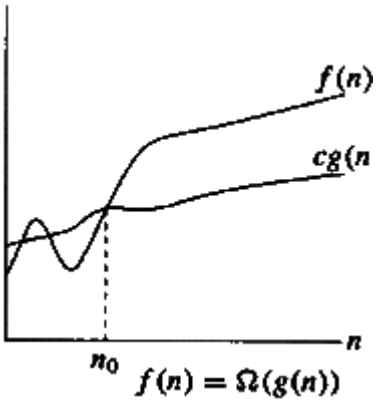
Big–OH O (Upper Bound)

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (≤) that of g(n).



Big–OMEGA Ω (Lower Bound)

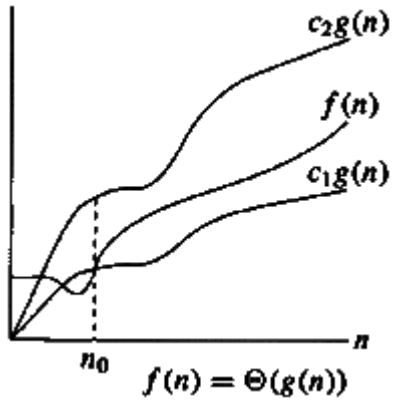
f(n) = Ω(g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (≥) that of g(n).



[1] In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as *big oh notation*.

Big–THETA θ (Same order)

f(n) = Θ(g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = Ω (g(n)].



Little–OH (o)

T(n) = o(p(n)) (pronounced little oh), says that the growth rate of T(n) is less than the growth rate of p(n) [if T(n) = O(p(n)) and T(n) ≠ θ(p(n))].

Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

O(1), O($\log_2$ n), O(n), O(n. $\log_2$ n), O(n^2), O(n^3), O(2^n), n! and n^n

Numerical Comparison of Different Algorithms

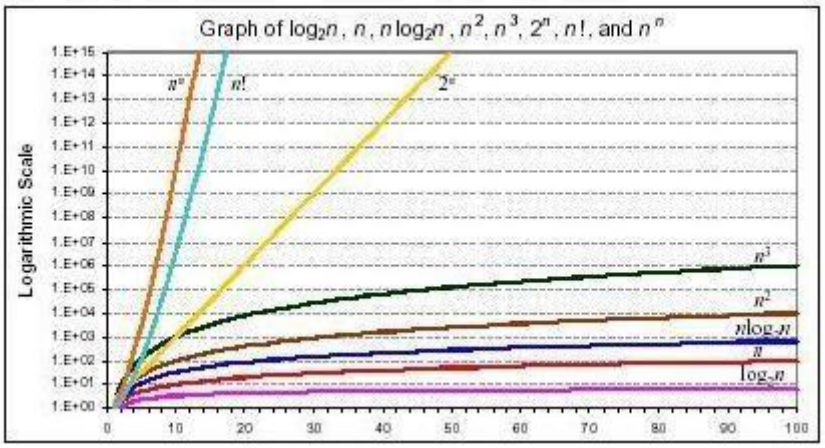
The execution time for six of the typical functions is given below:

n	**$\log_2$ n**	**n*$\log_2$n**	**n^2**	**n^3**	**2^n**
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billion years.

Graph of log n, n, n log n, n^2, n^3, 2^n, n! and n^n



O(log n) does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low–order terms while computing a Big–Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'. Then,

$$f(n) = O(g(n))$$

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is O (n)
- Binary search is O (log n)
- Bubble sort is O (n^2)
- Merge sort is O (n log n)

The rule of sums

Suppose that $T_1(n)$ and $T_2(n)$ are the running times of two programs fragments P_1 and P_2, and that $T_1(n)$ is O(f(n)) and $T_2(n)$ is O(g(n)). Then $T_1(n) + T_2(n)$, the running time of P_1 followed by P_2 is O(max f(n), g(n)), this is called as rule of sums.

For example, suppose that we have three steps whose running times are respectively $O(n^2)$, $O(n^3)$ and O(n. log n). Then the running time of the first two steps executed sequentially is O ($\max(n^2, n^3)$) which is $O(n^3)$. The running time of all three together is O(max (n^3, n. log n)) which is $O(n^3)$.

The rule of products

If $T_1(n)$ and $T_2(n)$ are O(f(n) and O(g(n)) respectively. Then $T_1(n)*T_2(n)$ is O(f(n) g(n)). It follows term the product rule that O(c f(n)) means the same thing as O(f(n)) if 'c' is any positive constant. For example, $O(n^2/2)$ is same as $O(n^2)$.
Suppose that we have five algorithms A_1–A_5 with the following time complexities:

A_1 : n
A_2 : n log n
A_3 : n^2
A_4: n^3
A_5: 2^n

The time complexity is the number of time units required to process an input of size 'n'. Assuming that one unit of time equals one millisecond. The size of the problems that can be solved by each of these five algorithms is:

Algorithm	Time complexity	Maximum problem size		
		1 second	1 minute	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	n log n	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

The speed of computations has increased so much over last thirty years and it might seem that efficiency in algorithm is no longer important. But, paradoxically, efficiency matters more today then ever before. The reason why this is so is that our ambition has grown with our computing power. Virtually all applications of computing simulation of physical data are demanding more speed.

The faster the computer run, the more need are efficient algorithms to take advantage of their power. As the computer becomes faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose the next generation of computers is ten times faster than the current generation, from the table we can see the increase in size of the problem.

Algorithm	Time Complexity	Maximum problem size before speed up	Maximum problem size after speed up
A_1	n	S1	10 S1
A_2	n log n	S2	≈10 S2 for large S2
A_3	n^2	S3	3.16 S3
A_4	n^3	S4	2.15 S4
A_5	2^n	S5	S5 + 3.3

Instead of an increase in speed consider the effect of using a more efficient algorithm. By looking into the following table it is clear that if minute as a basis for comparison, by replacing algorithm A4 with A3, we can solve a problem six times larger; by replacing A4 with A2 we can solve a problem 125 times larger. These results are for more impressive than the two fold improvement obtained by a ten fold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant.

We therefore conclude that the asymptotic complexity of an algorithm is an important measure of the goodness of an algorithm.

The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

i. We would like an algorithm that is easy to understand, code and debug.

ii. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.

2. The quality of code generated by the compiler used to create the object program.

3. The nature and speed of the instructions on the machine used to execute the program, and

4. The time complexity of the algorithm underlying the program.

The running time depends not on the exact input but only the size of the input. For many programs, the running time is really a function of the particular input, and not just of the input size. In that case we define T(n) to be the worst case running time, i.e. the maximum overall input of size 'n', of the running time on that input. We also consider $T_{avg}(n)$ the average, over all input of size 'n' of the running time on that input. In practice, the average running time is often much harder to determine than the worst case running time. Thus, we will use worst–case running time as the principal measure of time complexity.

Seeing the remarks (2) and (3) we cannot express the running time T(n) in standard time units such as seconds. Rather we can only make remarks like the running time of such and such algorithm is proportional to n^2. The constant of proportionality will remain un-specified, since it depends so heavily on the compiler, the machine and other factors.

Asymptotic Analysis of Algorithms:

Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

Rules for using big-O:

The most important property is that big-O gives an upper bound only. If an algorithm is $O(n^2)$, it doesn't have to take n^2 steps (or a constant multiple of n^2). But it can't

take more than n^2. So any algorithm that is O(n), is also an $O(n^2)$ algorithm. If this seems confusing, think of big-O as being like "<". Any number that is < n is also < n^2.

1. Ignoring constant factors: O(c f(n)) = O(f(n)), where c is a constant; e.g. $O(20\ n^3) = O(n^3)$

2. Ignoring smaller terms: If a<b then O(a+b) = O(b), for example $O(n^2+n) = O(n^2)$

3. Upper bound only: If a<b then an O(a) algorithm is also an O(b) algorithm. For example, an O(n) algorithm is also an $O(n^2)$ algorithm (but not vice versa).

4. n and log n are "bigger" than any constant, from an asymptotic view (that means for large enough n). So if k is a constant, an O(n + k) algorithm is also O(n), by ignoring smaller terms. Similarly, an O(log n + k) algorithm is also O(log n).

5. Another consequence of the last item is that an O(n log n + n) algorithm, which is O(n(log n + 1)), can be simplified to O(n log n).

Calculating the running time of a program:

Let us now look into how big-O bounds can be computed for some common algorithms.

Example 1:

Let's consider a short piece of source code:

```
x = 3*y + 2;
z = z + 1;
```

If y, z are scalars, this piece of code takes a *constant* amount of time, which we write as O(1). In terms of actual computer instructions or clock ticks, it's difficult to say exactly how long it takes. But whatever it is, it should be the same whenever this piece of code is executed. O(1) means *some* constant, it might be 5, or 1 or 1000.

Example 2:

$2n^2 + 5n - 6 = O(2^n)$ $2n^2 + 5n - 6 = O(n^3)$ $2n^2 + 5n - 6 = O(n^2)$ $2n^2 + 5n - 6 \neq O(n)$	$2n^2 + 5n - 6 \neq \Theta(2^n)$ $2n^2 + 5n - 6 \neq \Theta(n^3)$ $2n^2 + 5n - 6 = \Theta(n^2)$ $2n^2 + 5n - 6 \neq \Theta(n)$
$2n^2 + 5n - 6 \neq \Omega(2^n)$ $2n^2 + 5n - 6 \neq \Omega(n^3)$ $2n^2 + 5n - 6 = \Omega(n^2)$ $2n^2 + 5n - 6 = \Omega(n)$	$2n^2 + 5n - 6 = o(2^n)$ $2n^2 + 5n - 6 = o(n^3)$ $2n^2 + 5n - 6 \neq o(n^2)$ $2n^2 + 5n - 6 \neq o(n)$

Example 3:

If the first program takes $100n^2$ milliseconds and while the second takes $5n^3$ milliseconds, then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5\ n^3/100\ n^2 = n/20$$

for inputs $n < 20$, the program with running time $5n^3$ will be faster than those the one with running time $100\ n^2$. Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is n/20, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as O(n) or O(n log n) are always better.

Example 4:

Analysis of simple for loop

Now let's consider a simple for loop:

```
for (i = 1; i<=n; i++)
        v[i] = v[i] + 1;
```

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n. We write it as O(n). The actual number of instructions might be 50n, while the running time might be 17n microseconds. It might even be 17n+3 microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it's a linear function which is proportional to n, the correct notation is O(n) and the code is said to have *linear* running time.

Example 5:

Analysis for nested for loop

Now let's look at a more complicated example, a nested for loop:

```
for (i = 1; i<=n; i++)
        for (j = 1; j<=n; j++)
                a[i,j] = b[i,j] * x;
```

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes $n \times n = n^2$ times. The assignment statement in the inner loop takes constant time, so the running time of the code is $O(n^2)$ steps. This piece of code is said to have *quadratic* running time.

Example 6:

Analysis of matrix multiply

Lets start with an easy case. Multiplying two n $\times$ n matrices. The code to compute the matrix product C = A * B is given below.

```
for (i = 1; i<=n; i++)
        for (j = 1; j<=n; j++)
                C[i, j] = 0;
                for (k = 1; k<=n; k++)
                        C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes n*n*n = n^3 times. The innermost statement, which contains a scalar sum and product takes constant O(1) time. So the algorithm overall takes $O(n^3)$ time.

Example 7:

Analysis of bubble sort

The main body of the code for bubble sort looks something like this:

```
for (i = n-1; i<1; i--)
        for (j = 1; j<=i; j++)
                if (a[j] > a[j+1])
                        swap a[j] and a[j+1];
```

This looks like the double. The innermost statement, the if, takes O(1) time. It doesn't necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here. The outer loop executes n times, but the inner loop executes a number of times that depends on i. The first time the inner for executes, it runs i = n-1 times. The second time it runs n-2 times, etc. The total number of times the inner if statement executes is therefore:

(n-1) + (n-2) + ... + 3 + 2 + 1

This is the sum of an arithmetic series.

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

The value of the sum is n(n-1)/2. So the running time of bubble sort is O(n(n-1)/2), which is $O((n^2-n)/2)$. Using the rules for big-O given earlier, this bound simplifies to $O((n^2)/2)$ by ignoring a smaller term, and to $O(n^2)$, by ignoring a constant factor. Thus, bubble sort is an $O(n^2)$ algorithm.

Example 8:

Analysis of binary search

Binary search is a little harder to analyze because it doesn't have a for loop. But it's still pretty easy because the search interval halves each time we iterate the search. The sequence of search intervals looks something like this:

n, n/2, n/4, ..., 8, 4, 2, 1

It's not obvious how long this sequence is, but if we take logs, it is:

$\log_2 n, \log_2 n - 1, \log_2 n - 2, ..., 3, 2, 1, 0$

Since the second sequence decrements by 1 each time down to 0, its length must be $\log_2 n + 1$. It takes only constant time to do each test of binary search, so the total running time is just the number of times that we iterate, which is $\log_2 n + 1$. So binary search is an $O(\log_2 n)$ algorithm. Since the base of the log doesn't matter in an asymptotic bound, we can write that binary search is $O(\log n)$.

General rules for the analysis of programs

In general the running time of a statement or group of statements may be parameterized by the input size and/or by one or more variables. The only permissible parameter for the running time of the whole program is 'n' the input size.

1. The running time of each assignment read and write statement can usually be taken to be O(1). (There are few exemptions, such as in PL/1, where assignments can involve arbitrarily larger arrays and in any language that allows function calls in arraignment statements).

2. The running time of a sequence of statements is determined by the sum rule. I.e. the running time of the sequence is, to with in a constant factor, the largest running time of any statement in the sequence.

3. The running time of an if–statement is the cost of conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally O(1) the time for an if–then–else construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.

4. The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination (usually the latter is O(1)). Often this time is, neglected constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure.

Divide and Conquer

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.
Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
        if SMALL (P) then return S (p);
        else
        {
                divide p into smaller instances p1, p2, …. Pk, k ≥ 1;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p_1, p_2, . . . , p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

Recurrence Relations

Recurrence Relation for a sequence of numbers S is a formula that relates all but a finite number of terms of S to previous terms of the sequence, namely, $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$, for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer. Recurrence relations are also called as difference equations.

Sequences are often most easily defined with a recurrence relation; however the calculation of terms by directly applying a recurrence relation can be time consuming. The process of determining a closed form expression for the terms of a sequence from its recurrence relation is called solving the relation. Some guess and check with respect to solving recurrence relation are as follows:

- Make simplifying assumptions about inputs
- Tabulate the first few values of the recurrence
- Look for patterns, guess a solution
- Generalize the result to remove the assumptions

Examples: Factorial, Fibonnaci, Quick sort, Binary search etc.

Recurrence relation is an equation, which is defined in terms of itself. There is no single technique or algorithm that can be used to solve all recurrence relations. In fact, some recurrence relations cannot be solved. Most of the recurrence relations that we encounter are linear recurrence relations with constant coefficients.

Several techniques like substitution, induction, characteristic roots and generating function are available to solve recurrence relations.

The Iterative Substitution Method:

One way to solve a divide-and-conquer recurrence equation is to use the iterative substitution method. This is a "plug-and-chug" method. In using this method, we assume that the problem size n is fairly large and we than substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, performing such a substitution with the merge sort recurrence equation yields the equation.

$$T(n) = 2\,(2\,T(n/2^2) + b\,(n/2)) + b\,n$$
$$= 2^2\,T(n/2^2) + 2\,b\,n$$

Plugging the general equation for T again yields the equation.

$$T(n) = 2^2\,(2\,T(n/2^3) + b\,(n/2^2)) + 2\,b\,n$$
$$= 2^3\,T(n/2^3) + 3\,b\,n$$

The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general closed-form equation (with T only appearing on the left-hand side). In the case of merge-sort recurrence equation, the general form is:

$$T(n) = 2^i T(n/2^i) + i\, b\, n$$

Note that the general form of this equation shifts to the base case, $T(n) = b$, where $n = 2^i$, that is, when $i = \log n$, which implies:

$$T(n) = b\, n + b\, n \log n.$$

In other words, $T(n)$ is $O(n \log n)$. In a general application of the iterative substitution technique, we hope that we can determine a general pattern for $T(n)$ and that we can also figure out when the general form of $T(n)$ shifts to the base case.

Example 2.10.2: Consider the following recurrence equation (assuming the base case $T(n) = b$ for $n < 2$): $T(n) = 2T(n/2) + \log n$

This recurrence is the running time for the bottom-up heap construction. Which is $O(n)$. Nevertheless, if we try to prove this fact with the most straightforward inductive hypothesis, we will run into some difficulties. In particular, consider the following:

First guess: $T(n) \leq c\, n$.

For some constant $c > 0$. We can choose c large enough to make this true for the base case, so consider the case when $n \geq 2$. If we assume this guess as an inductive hypothesis that is true of input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2\,(c\,(n/2)) + \log n \\ &= c\, n + \log n \end{aligned}$$

But there is no way that we can make this last line less than or equal to cn for n > 2. Thus, this first guess was not sufficient, even though T (n) is indeed O (n). Still, we can show this fact is true by using:

Better guess: $T(n) \leq c\,(n - \log n)$

For some constant $c > 0$. We can again choose c large enough to make this true for the base case; in fact, we can show that it is true any time $n < 8$. So consider the case when $n \geq 8$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2\,c\,((n/2) - \log(n/2)) + \log n \\ &= c\, n - 2\, c \log n + 2\, c + \log n \\ &= c\,(n - \log n) - c \log n + 2\, c + \log n \\ &\leq c\,(n - \log n) \end{aligned}$$

Provided $c \geq 3$ and $n \geq 8$. Thus, we have shown that $T(n)$ is indeed $O(n)$ in this case.

The Master Theorem Method:

Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively. There is, nevertheless, one method for solving divide-and-conquer recurrence equations that is quite general and does not require explicit use of induction to apply correctly. It is the master method. The master method is a "cook-book" method for determining the asymptotic characterization of a wide variety of recurrence equations. It is used for recurrence equations of the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ a\,T(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Where d > 1 is an integer constant, a > 0, c > 0, and b > 1 are real constants, and f (n) is a function that is positive for $n \geq d$.

The master method for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing f (n) to the special function $n^{\log_b a}$ (we will show later why this special function is so important.

The master theorem: Let f (n) and T (n) be defined as above.

1. If there is a small constant ε > 0, such that f (n) is $O(n^{\log_b a - \varepsilon})$, then T (n) is $\Theta\left(n^{\log_b a}\right)$.
2. If there is a constant $K \geq 0$, such that f (n) is $\Theta\left(n^{\log_b a} \log^k n\right)$, then T (n) is $\Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
3. If there are small constant ε > 0 and δ < 1, such that f (n) is $\Omega\left(n^{\log_b a + \varepsilon}\right)$ and af(n/b) < δf(n), for $n \geq d$, then T (n) is Θ(f(n)).

Case 1 characterizes the situation where f (n) is polynomial smaller than the special function, $n^{\log_b a}$.

Case 2 characterizes the situation when f (n) is asymptotically close to the special function, and

Case 3 characterizes the situation when f (n) is polynomially larger than the special function.

We illustrate the usage of the master method with a few examples (with each taking the assumption that T (n) = c for n < d, for constants c > 1 and d >1).

Example 2.11.1: Consider the recurrence T (n) = 4 T (n/2) + n

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. Thus, we are in case 1, for f (n) is $O(n^{2-\varepsilon})$ for ε = 1. This means that T (n) is $\Theta(n^2)$ by the master.

Example 2.11.2: Consider the recurrenceT (n) = 2 T (n/2) + n log n

In this case, $n^{\log_b a} = n^{\log_2 2} = n$. Thus, we are in case 2, with k=1, for f (n) is Θ(n log n). This means that T (n) is $\Theta(n \log^2 n)$ by the master method.

Example 2.11.3: consider the recurrenceT (n) = T (n/3) + n

In this case $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for f (n) is $\Omega(n^{0+\varepsilon})$, for $\varepsilon = 1$, and af(n/b) = n/3 = (1/3) f(n). This means that T (n) is Θ (n) by the master method.

Example 2.11.4: Consider the recurrence T (n) = 9 T (n/3) + $n^{2.5}$

In this case, $n^{\log_b a} = n^{\log 9_3} = n^2$. Thus, we are in Case 3, since f (n) is $\Omega(n^{2+\varepsilon})$ (for ε=1/2) and af(n/b) = 9 $(n/3)^{2.5} = (1/3)^{1/2}$ f(n). This means that T (n) is Θ ($n^{2.5}$) by the master method.

Example 2.11.5: Finally, consider the recurrence T (n) = 2T ($n^{1/2}$) + log n

Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable k = log n, which lets us write:

T (n) = T (2^k) = 2 T ($2^{k/2}$) + k

Substituting into this the equation S (k) = T (2^k), we get that

S (k) = 2 S (k/2) + k

Now, this recurrence equation allows us to use master method, which specifies that S (k) is O (k log k). Substituting back for T (n) implies T (n) is O (log n log logn).

Example 2.13.1. Solve the following recurrence relation:

$$T(n) = \begin{cases} 2 & , n = 1 \\ 2.T\left(\frac{n}{2}\right) + 7 & , n > 1 \end{cases}$$

Solution: We first start by labeling the main part of the relation as Step 1:

Step 1: Assume n > 1 then,

$$T(n) = 2.T\left(\frac{n}{2}\right) + 7$$

Step 2: Figure out what $T\left(\frac{n}{2}\right)$ is; everywhere you see n, replace it with $\frac{n}{2}$.

$$T\left(\frac{n}{2}\right) = 2.T\left(\frac{n}{2^2}\right) + 7$$

Now substitute this back into the last T (n) definition (last line from step 1):

$$T(n) = 2.\left[2.T\left(\frac{n}{2^2}\right) + 7\right] + 7$$

$$= 2^2.T\left(\frac{n}{2^2}\right) + 3.7$$

Step 3: let's expand the recursive call, this time, it's $T\left(\frac{n}{2^2}\right)$:

$$T\left(\frac{n}{2^2}\right) = 2.T\left(\frac{n}{2^3}\right) + 7$$

Now substitute this back into the last T(n) definition (last line from step 2):

$$T(n) = 2^2 \left\lceil 2 \cdot T\left(\frac{n}{2^3} + 7\right) \right\rceil + 7$$

$$2^3 . T\left(\frac{n}{2^3}\right) + 7.7$$

From this, first, we notice that the power of 2 will always match i, current. Second, we notice that the multiples of 7 match the relation $2^i - 1$: 1, 3, 7, 15. So, we can write a general solution describing the state of T(n).

Step 4: Do the i^{th} substitution.

$$T(n) = 2^i . T\left(\frac{n}{2^i}\right) + \left(2^i - 1\right). 7$$

However, how many times could we take these "steps"? Indefinitely? No… it would stop when n has been cut in half so many times that it is effectively 1. Then, the original definition of the recurrence relation would give us the terminating condition T (1) = 1 and us restrict size $n = 2^i$

When, $1 = \frac{n}{2^i}$

$$\Rightarrow 2^i = n$$

$$\Rightarrow \log_2 2^i = \log_2 n$$

$$\Rightarrow i . \log_2 2 = \log_2 n$$

$$\Rightarrow i = \log_2 n$$

Now we can substitute this "last value for i" back into a our general Step 4 equation:

$$T(n) = 2_i . T\left(\frac{n}{2^i}\right) + \left(2_i - 1\right). 7$$

$$= 2^{\log_2 n} . T\left(\frac{n}{2^{\log_2 n}}\right) + \left(2^{\log_2 n} - 1\right). 7$$

$$= n . T(1) + (n - 1) . 7$$

$$= n . 2 + (n - 1) . 7$$

$$= 9 . n - 7$$

This implies that T (n) is **O (n).**

Example 2.13.2. Imagine that you have a recursive program whose run time is described by the following recurrence relation:

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2.T\left(\frac{n}{2}\right) + 4. n & , n > 1 \end{cases}$$

Solve the relation with iterated substitution and use your solution to determine a tight big-oh bound.

Solution:

Step 1: Assume n > 1 then,

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 4 \cdot n$$

Step 2: figure out what $T\left(\frac{n}{2}\right)$ is:

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

Now substitute this back into the last T (n) definition (last line from step 1):

$$T(n) = 2 \cdot \left[2 \cdot T\left(\frac{\frac{n}{2}}{2^2}\right) + 4 \cdot \frac{n}{2}\right] + 4 \cdot n$$

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

Step 3: figure out what $T\left(\frac{n}{2^2}\right)$ is:

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

Now substitute this back into the last T (n) definition (last time from step 2):

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

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

Step 4: Do the i^{th} substitution.

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

The parameter to the recursive call in the last line will equal 1 when $i = \log_2 n$ (use the same analysis as the previous example). In which case T (1) = 1 by the original definition of the recurrence relation.

Now we can substitute this back into a our general Step 4 equation:

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

$$= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + \log n \cdot 4 \cdot n$$

$$= n \cdot T(1) + 4 \cdot n \cdot \log_2 n$$

$$= n + 4 \cdot n \cdot \log_2 n$$

This implies that T (n) is **O(n log n).**

Example 2.13.4. If K is a non negative constant, then prove that the recurrence

$$T(n) = \begin{cases} k & , n = 1 \\ 3.T\left(\frac{n}{2}\right) + k \cdot n & , n > 1 \end{cases}$$

has the following solution (for n a power of 2)

$$T(n) = 3\ k \cdot n^{\log_2 3} - 2k \cdot n$$

Solution:

Assuming n > 1, we have T (n) = $3T\left(\frac{n}{2}\right) + K \cdot n$ (1)

Substituting $n = \frac{n}{2}$ for n in equation (1), we get

$$T\left(\frac{n}{2}\right) = 3T\left(\frac{n}{4}\right) + k\,\frac{n}{2} \quad (2)$$

Substituting equation (2) for $T\left(\frac{n}{2}\right)$ in equation (1)

$$T(n) = 3\left(3T\left(\frac{n}{4}\right) + k\right) + k \cdot n$$

$$T(n) = 3^2\, T\left(\frac{n}{4}\right) + 3\,k \cdot n + k \cdot n \qquad (3)$$

Substituting $n = \frac{n}{4}$ for n equation (1)

$$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{8}\right) + k\,n \qquad (4)$$

Substituting equation (4) for $T = \frac{n}{4}$ in equation (3)

$$T(n) = 3^2\left(3\,T\left(\frac{n}{8}\right) + k\,\frac{n}{4}\right) + 3\,k \cdot n + k \cdot n$$

$$3^3\, T\left(\frac{n}{8}\right) + \frac{9}{4}k\,n + \frac{3}{2}k.n + k.n \qquad (5)$$

Continuing in this manner and substituting $n = 2^i$, we obtain

$$T(n) = 3^i\, T\left(\frac{n}{2^i}\right) + \frac{3^{i-1}}{2^{i-1}}k.n + \ldots\ldots\ldots + \frac{9kn}{4} + \frac{3}{2}k \cdot n + k \cdot n$$

$$T(n) = 3^i\, T\left(\frac{n}{2^i}\right) + \left(\frac{\left(\frac{3}{2}\right)^i - 1}{\frac{3}{2} - 1}\right) k \cdot n \qquad \text{as} \quad \sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1}$$

$$= 3^i\, T\left(\frac{n}{2^i}\right) + 2 \cdot \left(\left(\frac{3}{2}\right)^i\right) k \cdot n - 2\,k\,n \qquad (6)$$

As $n = 2^i$, then $i = \log_2 n$ and by definition as T (1) = k

$$T(n) = 3^i\, k + 2 \cdot \frac{3^i}{n} \cdot k\,n - 2\,k\,n$$

$$= 3^i\,(3k) - 2\,k\,n$$

$$= 3\,k \cdot 3^{\log_2 n} - 2\,k\,n$$

$$= 3\,k\,n^{\log_2 3} - 2\,k\,n$$

Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm Algorithm

```
BINSRCH (a, n, x)
 // array a(1 : n) of elements in increasing order, n ≥ 0,
 // determine whether 'x' is present, and if so, set j such that x = a(j)
 // else return j

 {
        low :=1 ; high :=n ;
        while (low ≤ high) do
        {
                mid :=|(low + high)/2|
                if (x < a [mid]) then high:=mid – 1;
                else if (x > a [mid]) then low:= mid + 1
                        else return mid;
        }
        return 0;
  }
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid	
1	9	5	
6	9	7	
8	9	8	
9	9	9	
			found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid	
1	9	5	
6	9	7	
8	9	8	
			found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid	
1	9	5	
6	9	7	
6	6	6	
7	6		not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid	
1	9	5	
1	4	2	
1	1	1	
2	1		not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a[1], a[1] < x < a[2], a[2] < x < a[3], a[5] < x < a[6], a[6] < x < a[7] or a[7] < x < a[8] the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 =3.4

The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(log n).

Successful searches			**un-successful searches**
Θ(1),	Θ(log n),	Θ(log n)	Θ(log n)
Best	average	worst	best, average and worst

Analysis for worst case

Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

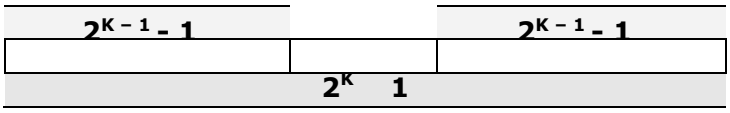
$T(0) = 0$

$T(n) = 1$ if x = a [mid]

$= 1 + T([(n + 1) / 2] - 1)$ if x < a [mid]

$= 1 + T(n - [(n + 1)/2])$ if x > a [mid]

Let us restrict 'n' to values of the form $n = 2^K - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\lceil n+1 \rceil = \lceil 2^K - 1 + 1 \rceil = 2^{K-1}$ for K > 1

$\lfloor \frac{\square}{2} \rfloor \quad \lfloor \frac{\quad}{2} \rfloor$

Giving,

$T(0) = 0$

$T(2^k - 1) = 1$ if x = a [mid]

$= 1 + T(2^{K-1} - 1)$ if x < a [mid]

$= 1 + T(2^{k-1} - 1)$ if x > a [mid]

In the worst case the test x = a[mid] always fails, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

This is now solved by repeated substitution:

$$
\begin{aligned}
w(2^k - 1) &= 1 + w(2^{k-1} - 1) \\
&= 1 + [1 + w(2^{k-2} - 1)] \\
&= 1 + [1 + [1 + w(2^{k-3} - 1)]] \\
&= \ldots\ldots\ldots \\
&= \ldots\ldots\ldots \\
&= i + w(2^{k-i} - 1)
\end{aligned}
$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^K - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

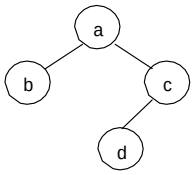
for $n = 2^K - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^K - 1$ weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^K - 1$.

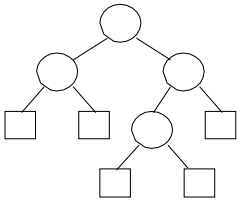
External and Internal path length:

The lines connecting nodes to their non-empty sub trees are called edges. A non-empty binary tree with n nodes has n–1 edges. The size of the tree is the number of nodes it contains.

When drawing binary trees, it is often convenient to represent the empty sub trees explicitly, so that they can be seen. For example:

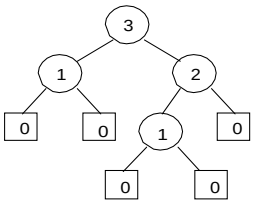


The tree given above in which the empty sub trees appear as square nodes is as follows:

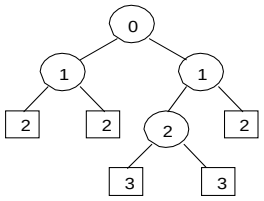


The square nodes are called as external nodes E(T). The square node version is sometimes called an extended binary tree. The round nodes are called internal nodes I(T). A binary tree with n internal nodes has n+1 external nodes.

The height h(x) of node 'x' is the number of edges on the longest path leading down from 'x' in the extended tree. For example, the following tree has heights written inside its nodes:



The depth d(x) of node 'x' is the number of edges on path from the root to 'x'. It is the number of internal nodes on this path, excluding 'x' itself. For example, the following tree has depths written inside its nodes:



The internal path length I(T) is the sum of the depths of the internal nodes of 'T':

$$I(T) = \sum_{x \in I(T)} d(x)$$

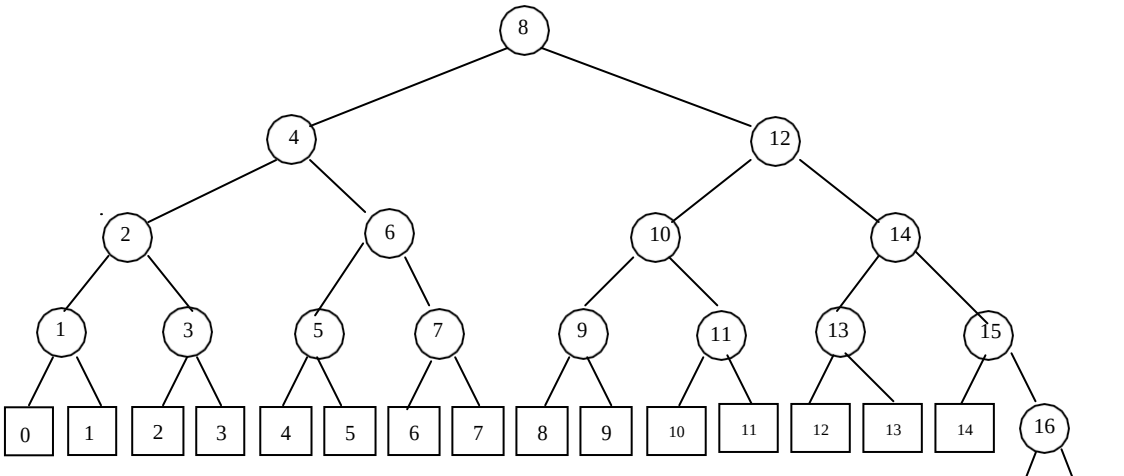
The external path length E(T) is the sum of the depths of the external nodes:

$$E(T) = \sum_{x \in E(T)} d(x)$$

For example, the tree above has I(T) = 4 and E(T) = 12.

A binary tree T with 'n' internal nodes, will have I(T) + 2n = E(T) external nodes.

A binary tree corresponding to binary search when n = 16 is



Represents internal nodes which lead for successful search

External square nodes, which lead for unsuccessful search.

Let C_N be the average number of comparisons in a successful search.

C'_N be the average number of comparison in an un successful search.

Then we have,

$$C_N = 1 + \frac{\text{internal pathlengthoftree}}{N}$$

$$C'_N = \frac{\text{External path length of tree}}{N+1}$$

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

External path length is always 2N more than the internal path length.

Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge mort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr, b ptr* and *c ptr,* which are initially set to the beginning of their respective arrays. The smaller of *a[a ptr]* and *b[b ptr]* is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to'c'.

To illustrate how merge process works. For example, let us consider the array 'a' containing 1, 13, 24, 26 and 'b' containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to 'c'. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
h ptr			

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1							
i ptr							

and then 2 and 13 are compared. 2 is added to 'c'. Increment *b ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2						
	i ptr						

then 13 and 15 are compared. 13 is added to 'c'. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13					
		i ptr					

24 and 15 are compared. 15 is added to 'c'. Increment *b ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h ptr	

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15				
			i ptr				

24 and 27 are compared. 24 is added to 'c'. Increment *a ptr* and *cptr*.

1	*2*	*3*	*4*
1	13	24	26
		h ptr	

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24			
				i ptr			

26 and 27 are compared. 26 is added to 'c'. Increment *a ptr* and *cptr.*

1	*2*	*3*	*4*
1	13	24	26
			h ptr

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26		
					i ptr		

As one of the lists is exhausted. The remainder of the b array is then copied to 'c'.

1	*2*	*3*	*4*
1	13	24	26

h ptr

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26	27	28

i ptr

Algorithm

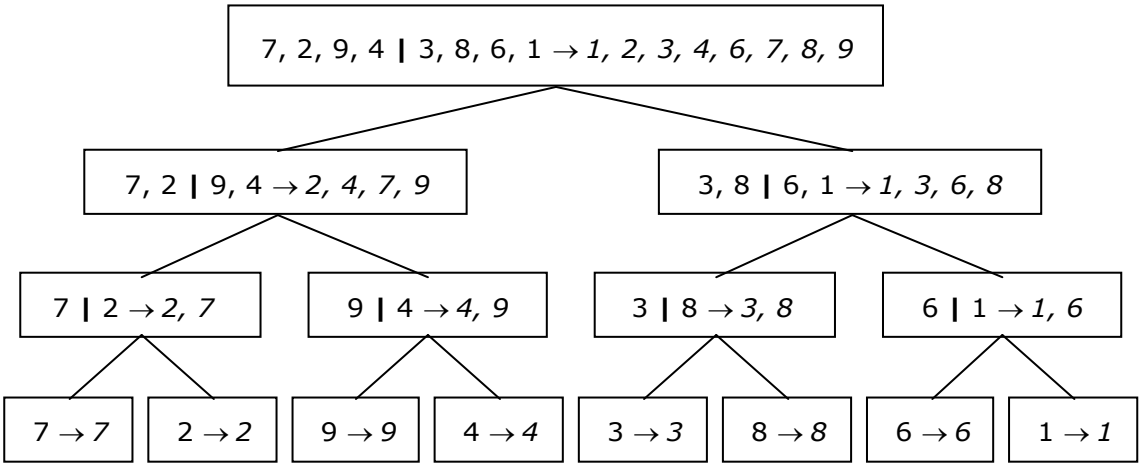
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
    if (low < high)
    {
        mid := ⌊(low + high)/2⌋ //finds where to split the set
        MERGESORT(low, mid) //sort one subset
        MERGESORT(mid+1, high) //sort the other subset
        MERGE(low, mid, high) // combine the results
    }
}

Algorithm MERGE (low, mid, high)

```
// a (low : high) is a global array containing two sorted subsets
// in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.
{
        h :=low; i := low; j:= mid + 1;
        while ((h ≤ mid) and (J ≤ high)) do
        {
                if (a[h] ≤ a[j]) then
                {
                        b[i] := a[h]; h := h + 1;
                }
                else
                {
                        b[i] :=a[j]; j := j + 1;
                }
                i := i + 1;
        }
        if (h > mid) then
                for k := j to high do
                {
                        b[i] := a[k]; i := i + 1;
                }
        else
                for k := h to mid do
                {
                        b[i] := a[K]; i := i + l;
                }
        for k := low to high do
                a[k] := b[k];
}
```

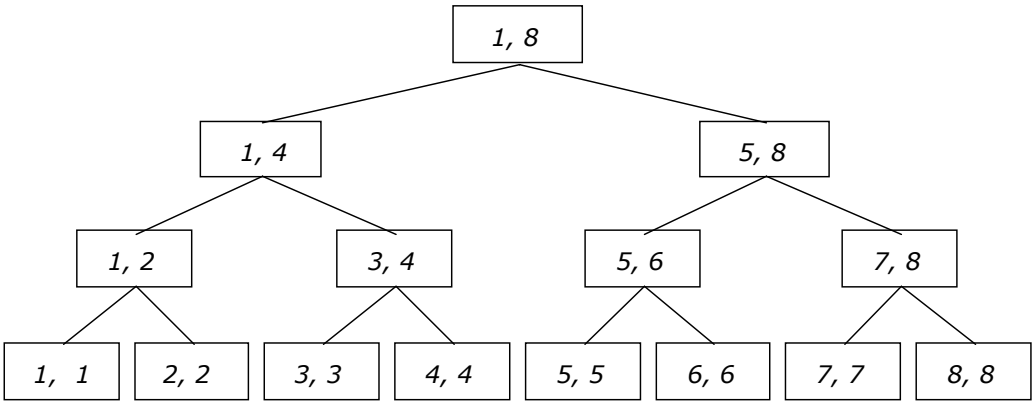
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



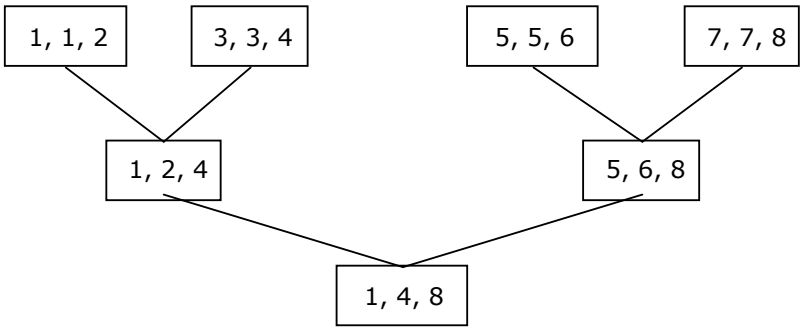
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says this exactly:

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

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right–hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

$2\ T(n/2) = 2\ (2\ (T(n/4)) + n/2)$
$= 4\ T(n/4) + n$

We have,

$T(n/2) = 2\ T(n/4) + n$
$T(n) = 4\ T(n/4) + 2n$

Again, by substituting n/4 into the main equation, we see that

$4T\ (n/4) = 4\ (2T(n/8)) + n/4$
$= 8\ T(n/8) + n$

So we have,

$T(n/4) = 2\ T(n/8) + n$
$T(n) = 8\ T(n/8) + 3n$

Continuing in this manner, we obtain:

$T(n) = 2^k\ T(n/2^k) + K.\ n$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

$T(n) = 2^{\log_2 n}\ T\left(\frac{2^k}{2^k}\right) + \log_2 n\ .\ n$

$= n\ T(1) + n \log n$
$= n \log n + n$

Representing this in O notation:

$T(n) =$ **O(n log n)**

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is O(n log n).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
        for j :=1 to n do
                c[i, j] := 0;
                for K: = 1 to n do
                        c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$C_{11} = A_{11} . B_{11} + A_{12} . B_{21}$

$C_{12} = A_{11} . B_{12} + A_{12} . B_{22}$

$C_{21} = A_{21} . B_{11} + A_{22} . B_{21}$

$C_{22} = A_{21} . B_{12} + A_{22} . B_{22}$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$T(1) = 1$
$T(n) = 8\ T(n/2)$

Which leads to T (n) = O (n^3), where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$P = (A_{11} + A_{22}) (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11} (B_{12} – B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} – A_{11}) (B_{11} + B_{12})$

$V = (A_{12} – A_{22}) (B_{21} + B_{22})$

$C_{11} = P + S – T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U.$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$T(1) = 1$
$T(n) = 7\ T(n/2)$

Solving this for the case of $n = 2^k$ is easy:

$T(2^k) = 7\ T(2^{k-1})$

$= 7^2\ T(2^{k-2})$

$= - - - - - -$
$= - - - - - -$

$= 7^i\ T(2^{k-i})$

Put i = k

$= 7^k\ T(1)$

$= 7^k$

That is, $T(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$= O(n^{\log_2 7}) = O(^2n^{.81})$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.

- If j > i, interchange a[j] with a[i]

- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . .x[high].

- It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

Algorithm Algorithm

```
QUICKSORT(low, high)
/* sorts the elements a(low), . . . . . , a(high) which reside in the global array A(1 :
n) into ascending order a (n + 1) is considered to be defined and must be greater
than all elements in a(1 : n); A(n + 1) = + ∝ */
{
        if low < high then
        {
                j := PARTITION(a, low, high+1);
                                // J is the position of the partitioning element
                QUICKSORT(low, j – 1);
                QUICKSORT(j + 1 , high);
        }
}

Algorithm PARTITION(a, m, p)
{
        V ← a(m); i ← m; j ← p;                 // A (m) is the partition element
        do
        {
                loop  i  := i   + 1  until  a(i) ≥ v          // i moves left to right
                loop  j  := j  – 1  until  a(j)  ≤ v          // p moves right to left
                if (i < j) then INTERCHANGE(a, i, j)
         } while (i ≥ j);
        a[m] :=a[j]; a[j] :=V; // the partition element belongs at position P
        return j;
}
```

Algorithm INTERCHANGE(a, i, j)
{
P:=a[i];
a[i] := a[j];
a[j] := p;
}

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	**24**								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot & j
	pivot, j	i											
	(04)	**06**											swap pivot & j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot & j
							45 pivot, j, i						swap pivot & j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot & j
									57 pivot, j, i				
											(70	79)	
											pivot, j	i	swap pivot & j
											70		
												79 pivot, j, i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take T (0) = T (1) = 1, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + C\,n \qquad - \qquad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1.

Worst Case Analysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + C\,n \qquad n > 1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly, thus

T (n – 1) = T (n – 2) + C (n – 1)

T (n – 2) = T (n – 3) + C (n – 2)

- - - - - - - -

T (2) = T (1) + C (2)

Adding up all these equations yields

$$T(n) = T(1) + \sum_{i=2}^{n} i$$

= **O (n²)** - (3)

Best Case Analysis

In the best case, the pivot is in the middle. To simply the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big – oh answer.

T (n) = 2 T (n/2) + C n - (4)

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \quad - \quad (5)$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \quad - \quad (6)$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \quad - \quad (7)$$

- - - - - - - -

- - - - - - - -

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \quad - \quad (8)$$

We add all the equations from 4 to 8 and note that there are log n of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \quad - \quad (9)$$

Which yields, T (n) = C n log n + n = **O(n log n)** - (10)

This is exactly the same analysis as merge sort, hence we get the same answer.

Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-2) +T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) + ----- + T(n-2)]\

Subtracting both sides:

nT(n) –(n-1)T(n-1) = [n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1)
nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)
T(n) = 2 + (n+1)T(n-1)/n
The recurrence relation obtained is:
T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Using the method of subsititution:

T(n)/(n+1) = 2/(n+1) + T(n-1)/n
T(n-1)/n = 2/n + T(n-2)/(n-1)
T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)
T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)
. .
. .
T(3)/4 = 2/4 + T(2)/3
T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 + T(0)
Adding both sides:
T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 + T(1)/2]
= [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 + T(1)/2] + T(0) +
[2/(n+1) + 2/n + 2/(n-1) + ---------- +2/4 + 2/3]
Cancelling the common terms:
T(n)/(n+1) = 2[1/2 +1/3 +1/4+----------+1/n+1/(n+1)]

T(n) = (n+1)2[$\sum_{2 \le k \le n+1} 1/k$

=2(n+1) [–]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2
T(n)= O(n log n)

3.8. Straight insertion sort:

Straight insertion sort is used to create a sorted list (initially list is empty) and at each iteration the top number on the sorted list is removed and put into its proper

place in the sorted list. This is done by moving along the sorted list, from the smallest to the largest number, until the correct place for the new number is located i.e. until all sorted numbers with smaller values comes before it and all those with larger values comes after it. For example, let us consider the following 8 elements for sorting:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	27	412	71	81	59	14	273	87

Solution:

Iteration 0:	unsorted	412	71	81	59	14	273	87
	Sorted	27						
Iteration 1:	unsorted	412	71	81	59	14	273	87
	Sorted	27	412					
Iteration 2:	unsorted	71	81	59	14	273	87	
	Sorted	27	71	412				
Iteration 3:	unsorted	81	39	14	273	87		
	Sorted	27	71	81	412			
Iteration 4:	unsorted	59	14	273	87			
	Sorted	274	59	71	81	412		
Iteration 5:	unsorted	14	273	87				
	Sorted	14	27	59	71	81	412	
Iteration 6:	unsorted	273	87					
	Sorted	14	27	59	71	81	273	412
Iteration 7:	unsorted	87						
	Sorted	14	27	59	71	81	87	273	412