Analysis and Design of Algorithm

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT - I ALGORITHM ANALYSIS

Algorithm analysis – Time space tradeoff – Asymptotic notations – Conditional asymptotic notation – Removing condition from the conditional asymptotic notation – Properties of Big-oh notation – Recurrence equations – Solving recurrence equations – Analysis of linear search.

1.1 Introduction

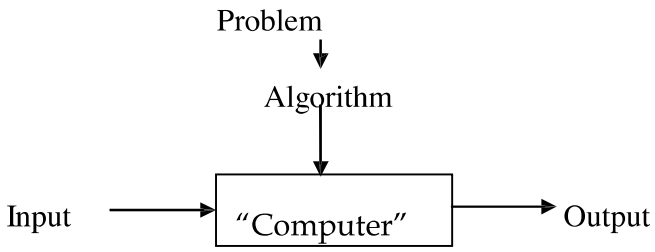
An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.

Definition

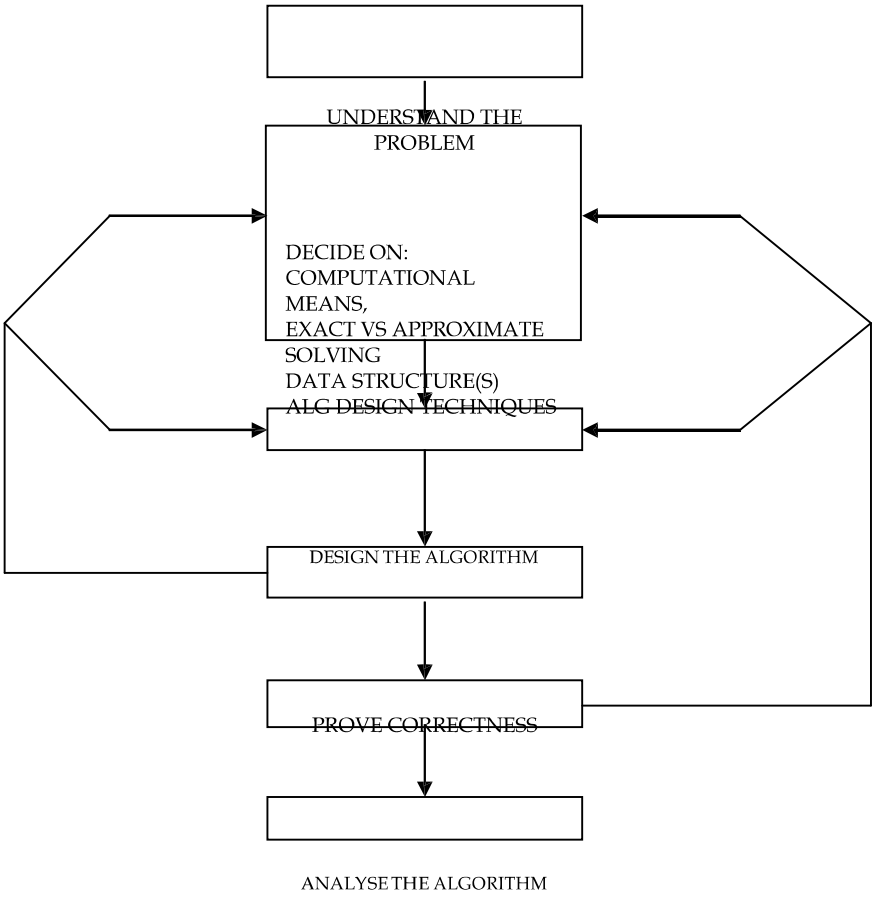
"Algorithmic is more than the branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant it most of science, business and technology"

Understanding of Algorithm

An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.



ALGORITHM DESIGN AND ANALYSIS PROCESS



1.2 FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

1.2.1 ANALYSIS FRAME WORK

there are two kinds of efficiency

Time efficiency - indicates how fast an algorithm in question runs.

Space efficiency - deals with the extra space the algorithm requires.

1.2.2 MEASURING AN INPUT SIZE

An algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

In most cases, selecting such a parameter is quite straightforward.

For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.

For the problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

There are situations, of course, where the choice of a parameter indicating an input size does matter.

Example - computing the product of two n-by-n matrices.

There are two natural measures of size for this problem.

The matrix order n.

The total number of elements N in the matrices being multiplied.

Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.

The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

We should make a special note about measuring size of inputs for algorithms involving properties of numbers (e.g., checking whether a given integer n is prime).

For such algorithms, computer scientists prefer measuring size by the number b of bits in the n's binary representation:

$$b = \lfloor \log_2 n \rfloor + 1$$

This metric usually gives a better idea about efficiency of algorithms in question.

1.2.3 UNITS FOR MEASURING RUN TIME:

We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.

There are obvious drawbacks to such an approach. They are

Dependence on the speed of a particular computer

Dependence on the quality of a program implementing the algorithm

The compiler used in generating the machine code

The difficulty of clocking the actual running time of the program.

Since we are in need to measure algorithm efficiency, we should have a metric that does not depend on these extraneous factors.

One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both difficult and unnecessary.

The main objective is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

WORST CASE, BEST CASE AND AVERAGE CASE EFFICIENCES

It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.

But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.

Example, sequential search. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

Here is the algorithm's pseudo code, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition A[i] i= K will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

ALGORITHM Sequential Search(A[0..n -1], K)

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n -1] and a search key K
//Output: Returns the index of the first element of A that matches K
//          or -1 ifthere are no matching elements
 i   0
while i < n and A[i] ≠ K do
    i   i+1
if i < n return i
else return -1
```

Clearly, the running time of this algorithm can be quite different for the same list size n.

Worst case efficiency

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$\mathbf{C_{worst}(n) = n.}$$

The way to determine is quite straightforward

To analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst-case value $\mathbf{C_{worst}(n)}$

The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $\mathbf{C_{worst}(n)}$ its running time on the worst-case inputs.

Best case Efficiency

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.

We can analyze the best case efficiency as follows.

First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)

Then ascertain the value of C (n) on these most convenient inputs.

Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, $C_{best}(n) = 1$.

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency.

But it is not completely useless. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast.

Thus, such an algorithm might well be the method of choice for applications dealing with almost sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

Average case efficiency

It yields the information about an algorithm about an algorithm‘s behaviour on a typical|| and random|| input.

To analyze the algorithm's average-case efficiency, we must make some assumptions about possible i*n*puts of size n.

The investigation of the average case efficiency is considerably more difficult than investigation of the worst case and best case efficiency.

It involves dividing all instances of size n .into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same.

Then a probability distribution of inputs needs to be obtained or assumed so that the expected value of the basic operation's count can then be derived.

The average number of key comparisions $C_{avg}(n)$ can be computed as follows,

let us consider again sequential search. The standard assumptions are,

In the case of a successful search, the probability of the first match occurring in the ith position of the list is pin for every i, and the number of comparisons made by the algorithm in such a situation is obviously i.

In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being (1 - p). Therefore,

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \ldots + i \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n}\right] + n \cdot (1 - p)$$

$$= \frac{p}{n}\left[1 + 2 + 3 + \ldots + i + \ldots + n\right] + n(1 - p)$$

$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1 - p)$$

$$= \frac{p(n+1)}{2} + n(1 - p)$$

Example, if p = 1 (i.e., the search must be successful), the average number of key comparisons made by sequential search is (n + 1) /2.
If p = 0 (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

1.2.5 Asymptotic Notations

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Because the values are not exact quantities. We need only comparative statements like $c_1n^2 \leq t_p(n) \leq c_2n^2$.

For example, consider two programs with complexities $c_1n^2 + c_2n$ and c_3n respectively. For small values of n, complexity depend upon values of c_1, c_2 and c_3. But there will also be an n beyond which complexity of c_3n is better than that of $c_1n^2 + c_2n$.This value of n is called break-even point. If this point is zero, c_3n is always faster (or at least as fast). Common asymptotic functions are given below.

Function	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	n log n
n^2	Quadratic
n^3	Cubic

2^n	Exponential
n!	Factorial

Big'Oh'Notation(O)

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

It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as

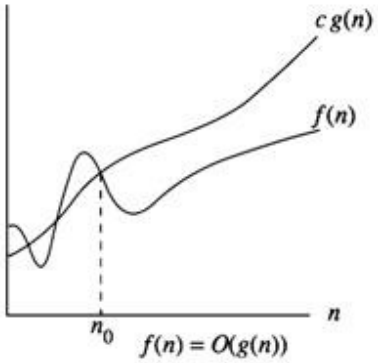


Fig 1.1

Find the Big Oh' for the following functions:

Linear Functions

Example 1.6
f(n) = 3n + 2

General form is f(n) ≤ cg(n)

When n ≥ 2, 3n + 2 ≤ 3n + n = 4n
Hence f(n) = O(n), here c = 4 and n_0 = 2

When n ≥ 1, 3n + 2 ≤ 3n + 2n = 5n
Hence f(n) = O(n), here c = 5 and n_0 = 1

Hence we can have different c,n_0 pairs satisfying for a given function.

Example
f(n) = 3n + 3
When n ≥ 3, 3n + 3 ≤ 3n + n = 4n
Hence f(n) = O(n), here c = 4 and n_0 = 3

Example
f(n) = 100n + 6
When n ≥ 6, 100n + 6 ≤ 100n + n = 101n
Hence f(n) = O(n), here c = 101 and n_0 = 6

Quadratic Functions

Example 1.9

$f(n) = 10n^2 + 4n + 2$

When $n \geq 2$, $10n^2 + 4n + 2 \leq 10n^2 + 5n$

When $n \geq 5$, $5n \leq n^2$, $10n^2 + 4n + 2 \leq 10n^2 + n^2 = 11n^2$

Hence $f(n) = O(n^2)$, here $c = 11$ and $n_0 = 5$

Example 1.10

$f(n) = 1000n^2 + 100n - 6$

$f(n) \leq 1000n^2 + 100n$ for all values of n.

When $n \geq 100$, $5n \leq n^2$, $f(n) \leq 1000n^2 + n^2 = 1001n^2$

Hence $f(n) = O(n^2)$, here $c = 1001$ and $n_0 = 100$

Exponential Functions

Example 1.11

$f(n) = 6*2^n + n^2$

When $n \geq 4$, $n^2 \leq 2^n$

So $f(n) \leq 6*2^n + 2^n = 7*2^n$

Hence $f(n) = O(2^n)$, here $c = 7$ and $n_0 = 4$

Constant Functions

Example 1.12

$f(n) = 10$

$f(n) = O(1)$, because $f(n) \leq 10*1$

OmegaNotation( )

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

It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as

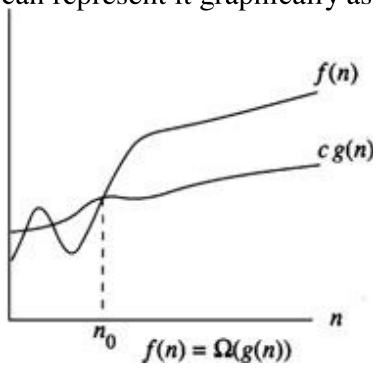


Fig 1.2

Example 1.13

f(n) = 3n + 2

3n + 2 > 3n for all n.

Hence f(n) = (n)

Similarly we can solve all the examples specified under Big Oh'.

ThetaNotation(Θ)

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

If f(n) = Θ(g(n)), all values of n right to n_0 f(n) lies on or above $c_1 g(n)$ and on or below $c_2 g(n)$. Hence it is asymptotic tight bound for f(n).

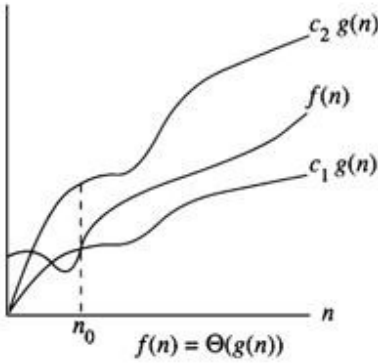


Fig 1.3

.

Little-O Notation

For non-negative functions, f(n) and g(n), f(n) is little o of g(n) if and only if f(n) = O(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = o(g(n))".

This represents a loose bounding version of Big O. g(n) bounds from the top, but it does not bound the bottom.

Little Omega Notation

For non-negative functions, f(n) and g(n), f(n) is little omega of g(n) if and only if f(n) = Ω(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = ω(g(n))".

Much like Little Oh, this is the equivalent for Big Omega. g(n) is a loose lower boundary of the function f(n); it bounds from the bottom, but not from the top.

Conditional asymptotic notation

Many algorithms are easier to analyse if initially we restrict our attention to instances whose size satisfies a certain condition, such as being a power of 2. Consider, for example, the divide and conquer algorithm for multiplying large integers that we saw in the Introduction. Let n be the size of the integers to be multiplied.

The algorithm proceeds directly if $n = 1$, which requires a microseconds for an appropriate constant a. If $n>1$, the algorithm proceeds by multiplying four pairs of integers of size $\lceil n/2 \rceil$ (or three if we use the better algorithm). Moreover, it takes a linear amount of time to carry out additional tasks. For simplicity, let us say that the additional work takes at most bn microseconds for an appropriate constant b.

Properties of Big-Oh Notation

Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. For example, if you want to write a function that searches through an array of numbers and returns the smallest one:

```
function find-min(array a[1..n] )
  let j :=
  for i := 1 to n:
    j := min(j, a[i] )
  repeat
return  j
end
```

Regardless of how big or small the array is, every time we run find-min, we have to initialize the i and j integer variables and return j at the end. Therefore, we can just think of those parts of the function as constant and ignore them.

So, how can we use asymptotic notation to discuss the find-min function? If we search through an array with 87 elements, then the for loop iterates 87 times, even if the very first element we hit turns out to be the minimum. Likewise, for n elements, the for loop iterates n times. Therefore we say the function runs in time O(n).

```
function find-min-plus-max(array a[1..n] )
  // First, find the smallest element in the array
  let j := ;
  for i := 1 to n:
    j := min(j, a[i] )
  repeat
  let minim := j

  // Now, find the biggest element, add it to the smallest and
  j := ;
  for i := 1 to n:
    j := max(j, a[i] )
  repeat

  let maxim := j

  // return the sum of the two
  return minim + maxim;
end
```

What's the running time for find-min-plus-max? There are two for loops, that each iterate n times, so the running time is clearly O(2n). Because 2 is a constant, we throw it away and write the running time as O(n). Why can you do this? If you recall the definition of Big-O notation, the function whose bound you're testing can be multiplied by some constant. If f(x)=2x, we can see that if g(x) = x, then the Big-O condition holds. Thus O(2n) = O(n). This rule is general for the various asymptotic notations.

Recurrence

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence

Recurrence Equation

A recurrence relation is an equation that recursively defines a sequence. Each term of the sequence is defined as a function of the preceding terms. A difference equation is a specific type of recurrence relation.

An example of a recurrence relation is the logistic map:

$$x_{n+1} = r x_n (1 - x_n)$$

1.3 Another Example: Fibonacci numbers

The Fibonacci numbers are defined using the linear recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values:

$$F_0 = 0$$
$$F_1 = 1$$

Explicitly, recurrence yields the equations:

$$F_2 = F_1 + F_0$$
$$F_3 = F_2 + F_1$$
$$F_4 = F_3 + F_2$$

We obtain the sequence of Fibonacci numbers which begins:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

It can be solved by methods described below yielding the closed form expression which involve powers of the two roots of the characteristic polynomial $t^2 = t + 1$; the generating function of the sequence is the rational function $t / (1 - t - t^2)$.

Solving Recurrence Equation

i. substitution method

The substitution method for solving recurrences entails two steps:

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution works. The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful, but it obviously

can be applied only in cases when it is easy to guess the form of the answer. The substitution method can be used to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n ,$$

which is similar to recurrences (4.2) and (4.3). We guess that the solution is T (n) = O(n lg n).Our method is to prove that T (n) ≤ cn lg n for an appropriate choice of the constant c > 0. We start by assuming that this bound holds for ⌊n/2⌋, that is, that T (⌊n/2⌋) ≤ c ⌊n/2⌋ lg(⌊n/2⌋).

Substituting into the recurrence yields

$$
\begin{aligned}
T(n) &\le 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\
&\le cn \lg(n/2) + n \\
&= cn \lg n - cn \lg 2 + n \\
&= cn \lg n - cn + n \\
&\le cn \lg n,
\end{aligned}
$$

where the last step holds as long as $c \ge 1$.

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. For the recurrence (4.4), we must show that we can choose the constant c large enough so that the bound T(n) = cn lg n works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that T (1) = 1 is the sole boundary condition of the recurrence. Then for n = 1, the bound T (n) = cn lg n yields T (1) = c1 lg 1 = 0, which is at odds with T (1) = 1. Consequently, the base case of our inductive proof fails to hold.

This difficulty in proving an inductive hypothesis for a specific boundary condition can be easily overcome. For example, in the recurrence (4.4), we take advantage of asymptotic notation only requiring us to prove T (n) = cn lg n for n ≥ n0, where n0 is a constant of our choosing. The idea is to remove the difficult boundary condition T (1) = 1 from consideration

1. In the inductive proof.

Observe that for n > 3, the recurrence does not depend directly on T (1). Thus, we can replace T (1) by T (2) and T (3) as the base cases in the inductive proof, letting n0 = 2. Note that we make a distinction between the base case of the recurrence (n = 1) and the base cases of the inductive proof (n = 2 and n = 3). We derive from the recurrence that T (2) = 4 and T (3) = 5. The inductive proof that T (n) ≤ cn lg n for some constant c ≥ 1 can now be completed by choosing c large enough so that T (2) ≤ c2 lg 2 and T (3) ≤ c3 lg 3. As it turns out, any choice of c ≥ 2 suffices for the base cases of n = 2 and n = 3 to hold. For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n.

2. The iteration method

The method of iterating a recurrence doesn't require us to guess the answer, but it may require more algebra than the substitution method. The idea is to expand (iterate) the

recurrence and express it as a summation of terms dependent only on n and the initial conditions. Techniques for evaluating summations can then be used to provide bounds on the solution.

As an example, consider the recurrence

T(n) = 3T(n/4) + n.
We iterate it as follows:

T(n) = n + 3T(n/4)

= n + 3 (n/4 + 3T(n/16))

= n + 3(n/4 + 3(n/16 + 3T(n/64)))

= n + 3 n/4 + 9 n/16 + 27T(n/64),
where n/4/4 = n/16 and n/16/4 = n/64 follow from the identity (2.4).

How far must we iterate the recurrence before we reach a boundary condition? The ith term in the series is $3^i\ n/4^i$. The iteration hits $n = 1$ when $n/4^i = 1$ or, equivalently, when i exceeds $\log_4 n$. By continuing the iteration until this point and using the bound $n/4^i \leq n/4^i$, we discover that the summation contains a decreasing geometric series:

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^{\log_4 n}\Theta(1) \\ &\leq n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n)\ . \end{aligned}$$

3. The master method

The master method provides a "cookbook" method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The master method requires memorization of three cases, but then the solution of many recurrences can be determined quite easily, often without pencil and paper.

The recurrence (4.5) describes the running time of an algorithm that divides a problem of size
n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$. (That is, using the notation from Section 2.3.2, $f(n) = D(n)+C(n)$.) For example, the recurrence arising from the MERGE-SORT procedure has $a = 2$, $b = 2$, and $f(n)$ =
$\Theta(n)$.

As a matter of technical correctness, the recurrence isn't actually well defined because n/b might not be an integer. Replacing each of the a terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ doesn't affect the asymptotic behavior of the recurrence, however. We normally find it convenient, therefore, to omit the floor and ceiling functions when writing divide-and- conquer recurrences of this form.

1.4 Analysis of Linear Search

Linear Search, as the name implies is a searching algorithm which obtains its result by traversing a list of data items in a linear fashion. It will start at the beginning of a list, and mosey on through until the desired element is found, or in some cases is not found. The aspect of Linear Search which makes it inefficient in this respect is that if the element is not in the list it will have to go through the entire list. As you can imagine this can be quite cumbersome for lists of very large magnitude, keep this in mind as you contemplate how and where to implement this algorithm. Of course conversely the best case for this would be that the element one is searching for is the first element of the list, this will be elaborated more so in the Analysis & Conclusion|| section of this tutorial.

Linear Search Steps:

Step 1 - Does the item match the value I'm looking for?

Step 2 - If it does match return, you've found your item!
Step 3 - If it does not match advance and repeat the process.
Step 4 - Reached the end of the list and still no value found? Well obviously the item is not in the list! Return -1 to signify you have not found your value.

As always, visual representations are a bit more clear and concise so let me present one for you now. Imagine you have a random assortment of integers for this list:

Legend:
-The key is blue
-The current item is green.
-Checked items are red

Ok so here is our number set, my lucky number happens to be 7 so let's put this value as the key, or the value in which we hope Linear Search can find. Notice the indexes of the

---

array above each of the elements, meaning this has a size or length of 5. I digress let us look at the first term at position 0. The value held here 3, which is not equal to 7. We move on.
--0 1 2 3 4 5
[3 2 5 1 7 0]

So we hit position 0, on to position 1. The value 2 is held here. Hmm still not equal to 7. We march on.
--0 1 2 3 4 5
[3 2 5 1 7 0]

Position 2 is next on the list, and sadly holds a 5, still not the number we're looking for. Again we move up one.
--0 1 2 3 4 5
[3 2 5 1 7 0]

Now at index 3 we have value 1. Nice try but no cigar let's move forward yet again.
--0 1 2 3 4 5
[3 2 5 1 7 0]

Ah Ha! Position 4 is the one that has been harboring 7, we return the position in the array which holds 7 and exit.
--0 1 2 3 4 5
[3 2 5 1 7 0]

As you can tell, the algorithm may work find for sets of small data but for incredibly large data sets I don't think I have to convince you any further that this would just be down right inefficient to use for exceeding large sets. Again keep in mind that Linear Search has its place and it is not meant to be perfect but to mold to your situation that requires a search.

Also note that if we were looking for lets say 4 in our list above (4 is not in the set) we would traverse through the entire list and exit empty handed. I intend to do a tutorial on Binary Search which will give a much better solution to what we have here however it requires a special case.

```
//linearSearch Function
int linearSearch(int data[], int length, int val){

  for (int i = 0; i <= length; i++){
     if (val == data[i]){
       return i;
     }//end if
  }//end for
  return -1;    //Value was not in the list
}//end linearSearch Function
```

Analysis & Conclusion

As we have seen throughout this tutorial that Linear Search is certainly not the absolute best method for searching but do not let this taint your view on the algorithm itself. People are always attempting to better versions of current algorithms in an effort to make existing ones more efficient. Not to mention that Linear Search as shown has its place and at the very least is a great beginner's introduction into the world of searching algorithms. With this is mind we progress to the asymptotic analysis of the Linear Search:

Worst Case:
The worse case for Linear Search is achieved if the element to be found is not in the list at all. This would entail the algorithm to traverse the entire list and return nothing. Thus the worst case running time is:

O(N).

Average Case:
The average case is in short revealed by insinuating that the average element would be somewhere in the middle of the list or N/2. This does not change since we are dividing by a constant factor here, so again the average case would be:

O(N).

Best Case:
The best case can be a reached if the element to be found is the first one in the list. This would not have to do any traversing spare the first one giving this a constant time complexity or:

O(1).

IMPORTANT QUESTIONS

PART-A

1. Define Algorithm & Notion of algorithm.
2. What is analysis framework?
3. What are the algorithm design techniques?
4. How is an algorithm‘s time efficiency measured?
5. Mention any four classes of algorithm efficiency.
6. Define Order of Growth.
7. State the following Terms.
(i) Time Complexity
(ii) Space Complexity
8. What are the various asymptotic Notations?
9. What are the important problem types?
10. Define algorithmic Strategy (or) Algorithmic Technique.
11. What are the various algorithm strategies (or) algorithm Techniques?
12. What are the ways to specify an algorithm?
13. Define Best case Time Complexity .
14. Define Worst case Time Complexity.
15. Define Average case time complexity.
16. What are the Basic Efficiency Classes.
17. Define Asymptotic Notation.
18. How to calculate the GCD value?

PART-B

1. (a) Describe the steps in analyzing & coding an algorithm. (10)
(b) Explain some of the problem types used in the design of algorithm. (6)
2. (a) Discuss the fundamentals of analysis framework . (10)
(b) Explain the various asymptotic notations used in algorithm design. (6)
3. (a) Explain the general framework for analyzing the efficiency of algorithm. (8)
(b) Explain the various asymptotic efficiencies of an algorithm. (8)
4. (a) Explain the basic efficiency classes. (10)
(b) Explain briefly the concept of algorithmic strategies. (6)
5. Describe briefly the notions of complexity of an algorithm. (16)
6. (a) What is Pseudo-code? Explain with an example. (8)
(b) Find the complexity C(n) of the algorithm for the worst case, best case and average case.(Evaluate average case complexity for n=3,Where n is the number of inputs) (8)