Design and Analysis of Computer Algorithms

Unit I

Algorithm: An algorithm is a finite set of instructions that, if followed accomplishes a particular task. All algorithms must satisfy the following criteria:

1. **Input**: Zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced
3. **Definiteness:** Each instruction is clear and unambiguous
4. **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps
5. **Effectiveness**: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. Performing arithmetic on integers is an example of an effective e operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

Algorithm Specification

Pseudocode Conventions

We can describe an algorithm in many ways such as use natural language like English, flowcharts etc. Most of our algorithms are using a pseudocode that resembles C and Pascal.

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces: { and }. A compound sttatement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;.
3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, Boolean, and so on. Compound data types can be formed with records. Here is an example:

```
node = record
        {
        Datatype_1 data-l;
                ..
        Datatype_n data_n;
        node    *link;
        }
```

   In this example, link is a pointer to the record type node. Individual data items of a record can be accessed with --> and period.
4. Assignment of values to variables is done using the assignment statement

```
<variable> := <expression>
```

5. There are two boolean values true and false. In order to produce these values, the logical operators and, or, and not and the relational operators <,≤,=, ≠, ≥ and > are provided.
6. Elements of multidimensional arrays are accessed using [and]. For example, the (i,j)th element of the array is denoted as A[i, j]. Array indices start at zero.
7. The following looping statements are employed: for, while, and repeat-until. The while loop takes the following form:

```
while <condition> do
{
    <statement 1>
      ……..
    <statement n>
}
```

The general form of a for loop is

```
for variable := value1 to value2 step step do
{
<statement 1>
………………..
<statement n>
}
```

Here valuel, value2, and step are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "step step" is optional and taken as +1 if it does not occur, step could either be positive or negative. variable is tested for termination at the start of each iteration.

A repeat-until statement is constructed as follows:

```
repeat
{
<statement 1>
………………….
<statement n>
until <condition>
```

The statements are executed as long as (condition) is false. The value of <condition> is computed after executing the statements.

The instruction break; can be used within any of the above looping instructions to force exit. In case of nested loops, break, results in the exit of the innermost loop that it is a part of. A return statement within any of the above also will result in exiting the loops. A return statement results in the exit of the function itself.

A conditional statement has the following forms:

```
if <condition> then <statement>
if <condition> then <statement 1>   else  <statement 2>
```

We also employ the following case statement:

```
case
{
        : <condition 1): <statement 1>
                  ………….
        : <condition 1): <statement 1>
        :else:<statement n + 1>
}
```

Here <statement 1>, <statement 2>, etc. could be either simple statements or compound statements. A case statement is interpreted as follows. If <condition 1> is true, <statement 1> gets executed and the statement is exited. If <statement 1> is false, <condition 2> is evaluated. If <condition 2> is true, <statement 2> gets executed and the case statement exited, and so on. If none of the conditions <condition 1>, ... , {condition n) are true, <statement n+1> is executed and the case statement is exited. The else clause is optional.

Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.

There is only one type of procedure: Algorithm. An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name ({parameter list})

where Name is the name of the procedure and ({parameter list)) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type:

As an example, the following algorithm finds and returns the maximum of n given numbers:

```
1. Algorithm Max(A, n)
2. //If A is an array of size n
3. {
4. Result := A[1];
5. for i := 2 to n do
6. lf A[i] > Result then Result := A[i];
7. return Result;
8. }
```

In this algorithm (named Max), A and n are procedure parameters. Result and i are local variables.

Recursive Algorithms

A recursive function is a function that is defined in terms of itself. Similarly an algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is direct recursive. Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

Example of Simple Recursive Algorithm:

Example (Algorithm 1.1): To find the factorial of a number

```
1. Int fact (n)
2. {
3. If (n==0) then return 1
4. Else
5. Return n*fact (n-1);
6. }
```

Tower of Hanoi

The Towers of Hanoi puzzle is fashioned after the ancient Tower of Brahma ritual.

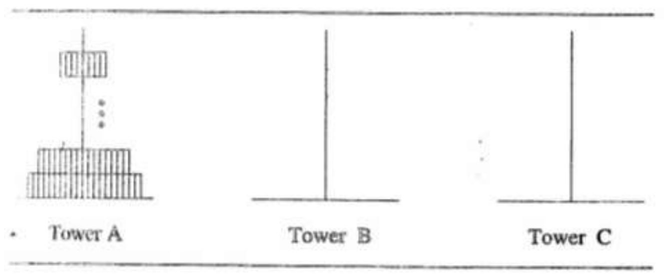


Figure 1.1 Towers of Hanoi

According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top. Besides this tower there were two other diamond towers (labeled B and C). Since the time of creation, Brahman priests have been attempting to move the disks from tower A to tower B using tower

C for intermediate storage. As the disks are very heavy, they can be moved only one at a time. In addition, at no time can a disk be on top of a smaller disk. According to legend, the world will come to an end when the priests have completed their task.

A very elegant solution results from the use of recursion. Assume that the number of disk is n. To get the largest disk to the bottom of tower B, we move the remaining *n* - 1 disks to tower C and then move the largest to tower B. Now we are left, with the task of moving the disks from tower *C to* tower B. To do this, we have towers A and B available. The fact that tower B has a disk on it can be ignored as the disk is larger than the dinks being moved from tower C and so any disk can be placed on top of it.

The recursive nature of the solution is apparent from given Algorithm. This algorithm is invoked by TowersOfHanoi(n,A,B,C). Observe that our solution for an n-disk problem is formulated in terms of solutions to two (n — 1) disk problems.

Algorithm

```
1. Algorithm TowersOfHanoi(n,x, y, z)
2. //Move the top n disks from tower x to tower y
3. {
4. If (n ≥ 1) then
5. {
6. TowersOfHanoi(n – 1, x, z, y);
7. Write ( " move top idsk from tower", x,
8. " to top of tower ", y );
9. TowersOfHanoi(n, z, y, x);
10. }
11. }
```

Algorithm Design Techniques

For a given problem, there are many ways to solve them. The different methods are listed below.

1. Divide and Conquer.
2. Greedy Algorithm.
3. Dynamic Programming.
4. Branch and Bound.
5. Backtracking Algorithms.
6. Randomized Algorithm.

1. Divide and Conquer: Divide and conquer method consists of three steps.
   a. Divide the original problem into a set of sub-problems.
   b. Solve every sub-problem individually, recursively.
   c. Combine the solutions of the sub-problems into a solution of the whole original problem.

2. Greedy Approach:

Greedy algorithms seek to optimize a function by making choices which are the best locally but do not look at the global problem. The result is a good solution but not necessarily the best one. Greedy Algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal solution.

3. Dynamic Programming.

Dynamic Programming is a technique for efficient solution. It is a method of solving problems exhibiting the properties of overlapping sub problems and optimal sub-structure that takes much less time than other methods.

4. Branch and Bound Algorithm.

In Branch and Bound Algorithm a given Algorithm which cannot be bounded has to be divided into at least two new restricted sub-problems. Branch and Bound Algorithm can be slow, however in the worst case they require efforts that grows exponentially with problem size. But in some cases the methods converge with much less effort. Branch and Bound Algorithms are methods for global optimization in non-convex problems.

5. Backtracking Algorithm.

They try each possibility until they find the right one. It is a depth first search of a set of possible solution. During the search if an alternative doesn't work, the search backtrack to the choice point, the place which presented different alternatives and tries the next alternative. If there are no more choice points the search fails.

6. Randomized Algorithm.

A Randomized Algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased, random bits, and it is then allowed to use these random bits to influence its computation.

Performance Analysis

There are two criteria for judging algorithms that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

The Space/Time complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Space Complexity

The space needed by each of these algorithms is seen to be the sum of the following components:

1. A fixed part that is independent of the characteristics (eg. Number, size) of the inputs and outputs. This part typically includes the instruction space (space for code), space for simple variable and fixed-size component variables (aggregate), space for constants and so on.

2. A variable part that consists of the consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics) and the recursion stack space (depends on the instance characteristics)

The space requirement S(P) of any algorithm P may therefore be written S (P) = c+ Sp (instance characteristics), where c is a constant. When analyzing the space complexity of an algorithm, we concentrate solely on estimating Sp (instanced characteristics). For any given problem, we need first to determine which instance characteristics to use to measure the space requirements.

Algorithm abc computes a+b+b*c+(a+b-c)/(a+ b)+4.0

```
1. Algorithm abc(a, b, c)
2. {
3.     Return a+b+b*c+(a+b-c)/(a+ b)+4.0;
4. }
```

Algorithm Sum computes $\sum_{i=1}^{n}$ a[i] iteratively, where the a[i]'s are real numbers; and RSum is a recursive algorithm that computes $\sum$ a[i].

Algorithm Sum

```
1. Algorithm Sum(a, n)
2. {
3.    s:=0.0;
4.    for i:= 1 to n do
5.        s:= s+ a [i];
6.    return s;
7. }
```

In the case of the algorithm abc, the problem instance is characterized by the specific values of a, b, and c. One word is adequate to store the values of each of a, b, c so space needed by abc is independent of the instance characteristics. Consequently, Sp(instance characteristics)= 0.

The problem instances for algorithm Sum are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type integer. The space needed by a is the space needed by variables of type array of floating point numbers. Then, Ssum(n) ≥ (n+3) (n for a[], one each for n, I, and s).

Time Complexity

The time T (P) taken by a program P is the sum of the compile time and the run (execution) time. The compile time does not depend on the instance characteristics. This run time is denoted by t*p* (instance characteristics). If we know the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P is used.

So we could obtain an expression for t*p* (n) of the form

$t_p(n) = c_a\, ADD(n) + c_s\, SUB(n) + c_m\, MUL(n) + c_d\, DIV(n) + \ldots$

Where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division and so on, and *ADD, SUB, MUL, DIV* and son, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

The runtime of an execution depends on the parameters such as,

- Number of comparisons
- Number of data exchanges.
- Presence of recursive executions.
- The Runtime is proportional to the size of the program.
- The execution time increases as the Input data set increases in size.

To illustrate how to find Time complexity. It is done by two methods as,

1. Step count method

2. Step per statement execution or frequency method

Step count method

In order to calculate the time of execution it is necessary to obtain a count for the total number of operations. It is done by count the number of program steps in a program. A **program step** is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

return a+b+b*c+(a+b-c)/(a+ b)+4.0; of algorithm abc could be regarded as a step.

The number of steps any program statement is assigned depended on the kind of statement. For example, comments count as zero steps; an assignment statement which does not involve any calls to other algorithms is counted as one step; in an iterative statement such as the for, while, and repeat-until statements, we consider the step counts only for the control part of the statement. The control parts of for and while statements have the following forms:

for i:= <expr> to <expr1> do

while (<expr>) do

Each execution of the control part of a while statement is given a step count equal to the number of step counts assignable to <expr>.

As an example the following algorithm shows the count variable of algorithm sum.

```
1. Algorithm Sum(a, n)
2. {
```

```
3.  s:=0.0;
4.  count:= count + 1; //count is global; it is initially zero.
5.  for i:= 1 to n do
6.  {
7.  count:= count + 1;  //For for
8.      s:= s+ a [i]; count:= count + 1; // For assignment
9.  }
10. count:= count + 1; // For last time of for
11. count:= count + 1; // For the return
12. return s;
13. }
```

Following algorithm shows the simplified form.

```
1. Algorithm Sum (a,n)
2. {
3. for i:= 1 to n do count := count + 2;
4. count := count + 3;
5. }
```

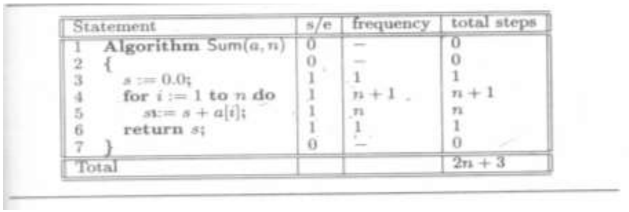
It is easy to see that in the **for** loop, the value of *count* will increase by a total of 2n. If *count* is zero to start with, then it will be 2n + 3 on termination. So each invocation of Sum executes a total of 2n + 3 steps.

Input size: One of the instance characteristics that is frequently used in the literature is the input size. The input size of any instance of a problem is defined to be the number of words (number of elements) needed to describe that instance. The input size for the problem of summing an array with n elements n + 1, n for listing the n elements and 1 for the value of n. The input size is normally taken to be the number of bits needed to specify that element.

Step per statement execution or frequency method

The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. For this first determining the number of steps per execution (s/e) of the statement and the total number of times (frequency) each statement is executed. The s/e of a statement is the amount by which the count changes as a result of the execution of that statement. By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

Following table shows the s/e of algorithm sum



Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	—	0
2 {	0	—	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	—	0
Total			$2n+3$

In the case of Sum algorithm, the total number of steps required by the algorithm is determined to be 2n+ 3. It is important to note that the frequency of the for statement is n+ 1 and not n. This is so because i has to be incremented to n + 1 before the for loop can terminate.

There are three kinds of step counts: **best case, worst case, and average**. The **best case step count** is the minimum number of steps that can be executed for the given parameters. The **worst case step count** is the maximum number of steps that can be executed for the given parameters. The **average step count** is the average number of steps executed on instances with the given parameters.

Eg. searching

Best case: If the algorithm finds the element at the first search itself, it is referred as a best case algorithm.

Worst case : If the algorithm finds the element at the end of the search or if the searching of the element fails, the algorithm is in the worst case or it requires maximum number of steps that can be executed for the given parameters.

Average case: The analysis of average case behavior is complex than best case and worst case analysis, and is taken by the probability of Input data. As the volume of Input data increases, the average case algorithm behaves like worst case algorithm. In an average case behavior; the searched element is located in between a position of the first and last element.

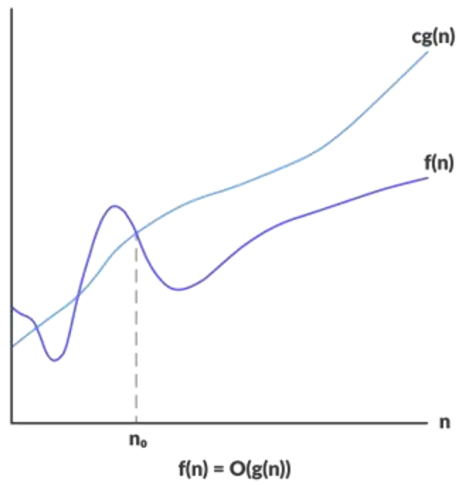
Asymptotic notation (O, Ω, Θ)

The Asymptotic notation introduces some terminology that enables to make meaningful statements about the time and space complexities of an algorithm. The functions *f* and *g* are nonnegative functions.

O – NOTATION (Upper Bound)

The O-notation (pronounced as Big "Oh") is used to measure the performance of an algorithm which depends on the volume of Input data. The O-notation is used to define the order ofgrowth of an algorithm, as the input size increases, the performance varies.

The function f (n) =O (g (n)) (read as f of n is big oh of g of n) iff there exists two positive constants c and n0 such that f (n) <=cg (n) for all n, where n>=n0.



Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

Suppose we have a program having count 3n+2. We can write it as f (n) =3n+2. We say that 3n+2=O (n), that is of the order of n, because f (n) <=4n or 3n + 2 <= 4n, for all n>=2.

Another e.g.

Suppose f (n) =$10n^2$+4n+2. We say that f (n) =O (n^2) sine $10n^2$+4n+2<=$11n^2$ for n>=2 But here we can't say that f (n) =O (n) since 10 n^2+ 4n+2 never less than or equal to cn, That is ,$10n^2$+4n+2 != O (n) , But we are able to say that f (n) =O (n^3) since f (n) can be less than or equal to $cn^{3,}$ same as $10n^2$+4n+2<=$10n^4$, for n>=2.

Normally suppose a program has step count equals 5, and then we say that it has an order O(constant) or O (1).

If f (n) = 3n+5 then O (n) or O (n^2) or O (n^3)

or ---But we cannot express its time

complexity as O (1).

If f (n) = $5n^2$+8n+2 then O (n^2) or O (n^3) or ---

But we cannot express its time complexity as O (n) or O (1).

If it is $6n^3$+3n+2 then O (n^3) or O (n^4) or ----

But we cannot express its time complexity as O (n^2) or O (n) or O (1).

If it is n log n+6n+9 then O (n log n) (to the

base 2) If it is log n+6 then O (log n)

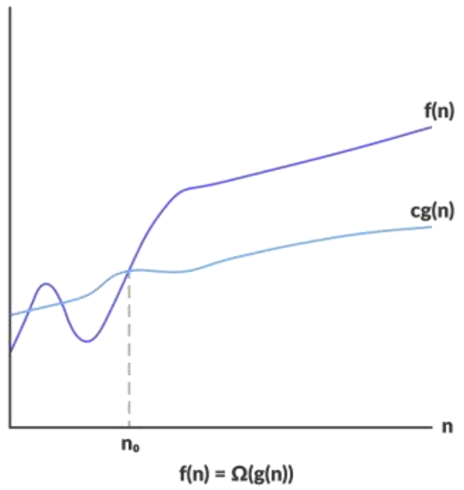
The seven various O-notations used are:

- O (1) – We write O (1) to mean a computing time that is constant. Where the data item is searched in the first search itself.
- O (n) – O (n) is called linear, where all the elements of the list are traversed and searched and the element is located at the n^{th} location.
- O (n^2) – It is called quadratic, where all elements of the list are traversed for each element. E.g.: worst case of bubble sort.
- O (log n) –The O(log n) is sufficiently faster, for a large value of n, when the searching is done by dividing a list of items into 2 half's and each time a half is traversed based on middle element. E.g.: binary searching, binary tree traversal.
- O (nlogn) – The O (nlogn) is better than O(n2),but not good as O(n),when the list is divided into 2-halfs and a half is traversed each time. E.g.: Quick sort.

The other notations are O (n^2), O (n^3), O (2^n) etc.

OMEGA NOTATION (Ω)

The function f (n) = **Ω** (g(n)) (read as " f of n is omega of g of n") iff there exists two positive constants c and n0 such that f(n) >= c * g(n) for all n, where n >= n0.



Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Suppose we have a program having count 3n+2. we can write it as f(n)=3n+2. And we say that3n+2= **Ω** (n), that is of the order of n, because f(n)>=3n or 3n+2>=3n, for all n>=1.

Normally suppose a program has step count equals 5, and then we say that it has an order
Ω (constant) or **Ω** (1).

If f(n) = 3n+5 then **Ω** (n) or **Ω** (1)

But we cannot express its time

complexity as O(n^2). If f(n) = $5n^2$+8n+2

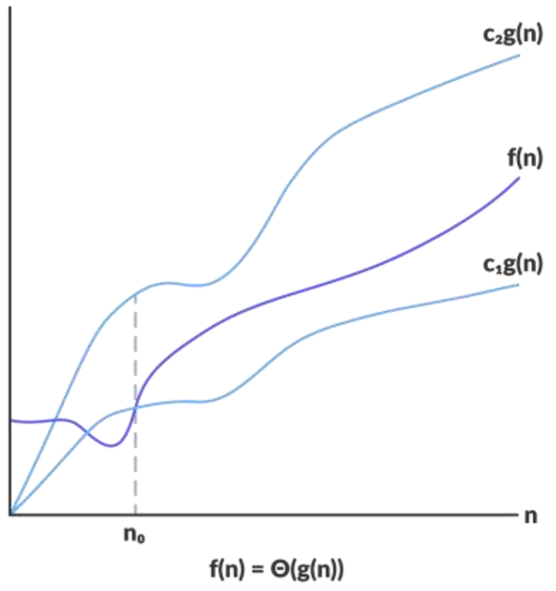
then **Ω** (n^2) or **Ω** (n) or **Ω** (1)

THETA NOTATION (Θ)

The function, f(n) = **Θ** (g(n)) (read as f of n is big Theta of g of n) iff there exists two positiveconstants c1, c2 and n0 such that c1.g(n) >= f(n) <= c2.g(n) for all n, where n >= n0.

Suppose we have a program having count 3n+2. we can write it as f(n)=3n+2. We say that 3n+2= **Θ** (n), that is of the order of n, because c1. n >= 3n + 2 <= c2. n .after a particular value of n.

But we cannot say that 3n + 2 = **Θ** (1) or **Θ** (n2) or **Θ** (n 3).This can be explained by the following graph.

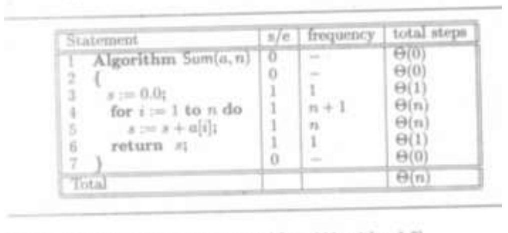


Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.
Normally suppose a program has step count equals 5, then we say that it has an order **Θ** (constant) or
Θ (1).

If f(n) = 3n+5 then **Θ** (n)

If f(n) = $5n^2$+8n+2 then **Θ** (n^2).

The asymptotic complexity can be determined easily without determining the exact step count. This is usually done by first determining the asymptotic complexity of each statement in the algorithm and then adding these complexities. Following tables show asymptotic complexity of Sum



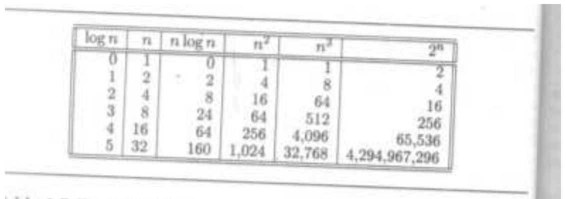
Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 $s := 0.0;$	1	1	$\Theta(1)$
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$\Theta(n)$
5 $s := s + a[i];$	1	n	$\Theta(n)$
6 **return** $s;$	1	1	$\Theta(1)$
7 }	0	–	$\Theta(0)$
Total			$\Theta(n)$

Table 1.4 Asymptotic complexity of Sum (Algorithm 1.6)

Practical complexities

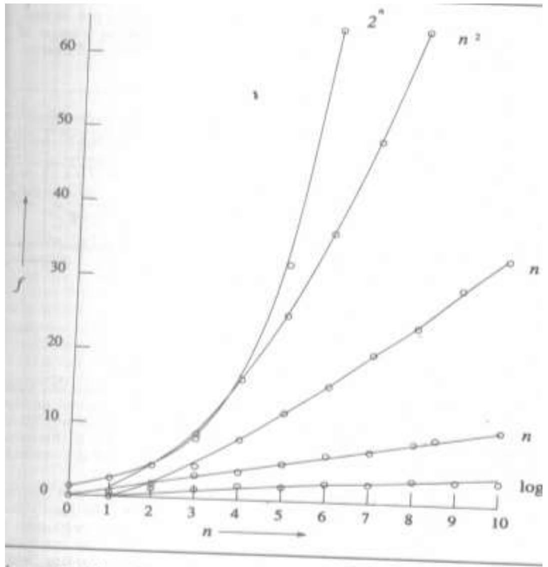
We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. The complexity function can also be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity Θ(n) and algorithm Q has complexity Θ(n^2). We can assert that algorithm P is faster than algorithm Q for sufficiently large n. When deciding which of the two algorithms to use, you must know whether the n you are dealing with is, in fact, sufficiently large. If algorithm P runs in 10^6 n milliseconds, whereas algorithm Q runs in n^2 milliseconds, and if you always have n ≤10^6 , then, other factors being equal, algorithm Q is the one to use.

To get a feel for how the various functions grow with n, observe the following table and figure very closely.



$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

able 1.7 Function values



igure 1.3 Plot of function values

The function 2^n grows very rapidly with n. In fact, if an algorithm needs 2^n steps for execution, then when n = 40, the number of steps needed is approximately 1.1 * 10^{12}. On a computer performing one billion steps per second, this would require about 18.3 minutes. If n = 50, the same algorithm would run for about 13 days on this computer. So we can say that the utility of algorithms with exponential complexity is limited to small n (typically n ≤ 40).

Algorithms that have a complexity that is a polynomial of high degree are also of limited utility. If the value of n is reasonably large (n > 100) small complexity such as n, nlogn, n^2, and n^3 are only feasible.

Polylog, polynomial, and exponential: These are the most common functions that arise in analyzing algorithms:

Polylogarithmic: Powers of log *n*, such as $(\log n)^7$. We will usually write this as $\log^7 n$.

Polynomial: Powers of *n*, such as n^4

Exponential: A constant (not 1) raised to the power *n*, such as 3^n.

Performance measurement

Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as on the computer on which the algorithm is run. To obtain the computing or run time of a program, we need a clocking procedure. We assume the existence of a program GetTime() that returns the current time in milliseconds.

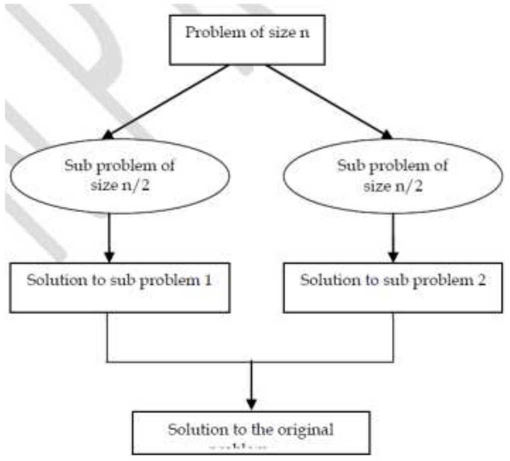
DIVIDE AND CONQUER METHOD

General Method

Divide and Conquer is one of the best-known general algorithm design technique. Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original problem.

The divide-and-conquer technique is diagrammed, which depicts the case of dividing a problem into two smaller sub problems,



A function to compute on n inputs the D and C strategy suggests splitting the inputs into k distinct sub sets, $1<k \le n$. These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. The control abstraction of the divide and conquer strategy is given below.

Control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures.

```
1.  Algorithm DAndC ( P)
2.  {
3.      if Small (P) then return S(P);
4.      else
5.      {
6.          Divide P into smaller instances P1, P2, ……………………Pk, k ≥ 1;
7.          Apply DAndC to each of these sub problems;
8.          return Combine (DAndC (P1), DAndC (P2), …….. , DAndC(Pk));
9.      }
10. }
```

DAndC is initially invoked as DAndC (P) where P is the problem to be solved. Small (P) is a Boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem P is divided into smaller sub problems. These sub problems P1, P2, ……., Pk are solved by recursive applications of DAndC. Combine is the function that determines the solution to P using the solutions to the k sub problems. Computing time of DAndC is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n_1) + T(n_2) + \ldots\ldots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

Where T (n) is the time for DAndC on any input of size n and g(n) is the time to compute the answer directly for small inputs. The function f (n) is the time for dividing P and combing the solutions to sub problems.

One of the methods for solving any such recurrence relation is called the substitution method. This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.

ADVANTAGES:

- The time spent on executing the problem using divide and conquer is smaller than other methods.

- The divide and conquer approach provides an efficient algorithm in computer science.
- The divide and conquer technique is ideally suited for parallel computation in which each sub problem can be solved simultaneously by its own processor.

Merge Sort

Merge sort is another example of divide and conquer approach. A sorting algorithm that has the property in the worst case its complexity is O (n log n) is called merge sort. Elements are sorted in non decreasing order and that are called as keys.

STEPS TO BE FOLLOWED

♦ The first step of the merge sort is to chop the list into two.

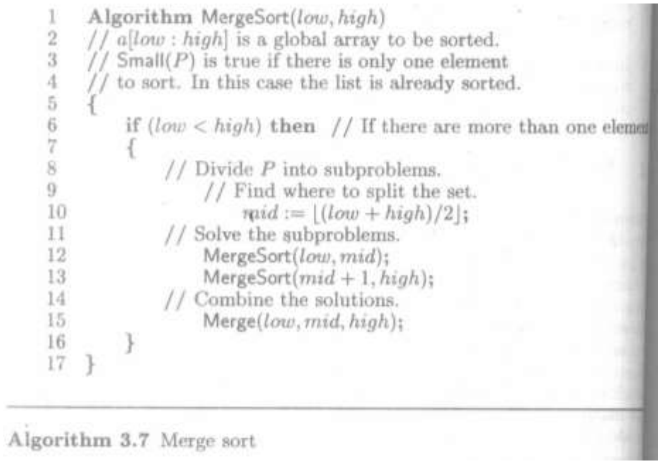
♦ If the list has even length, split the list into two equal sub lists.

♦ If the list has odd length, divide the list into two by making the first sub list one entry greater than the second sub list.

♦ then split both the sub list into two and go on until each of the sub lists are of size one.

♦ finally, start merging the individual sub lists to obtain a sorted list.

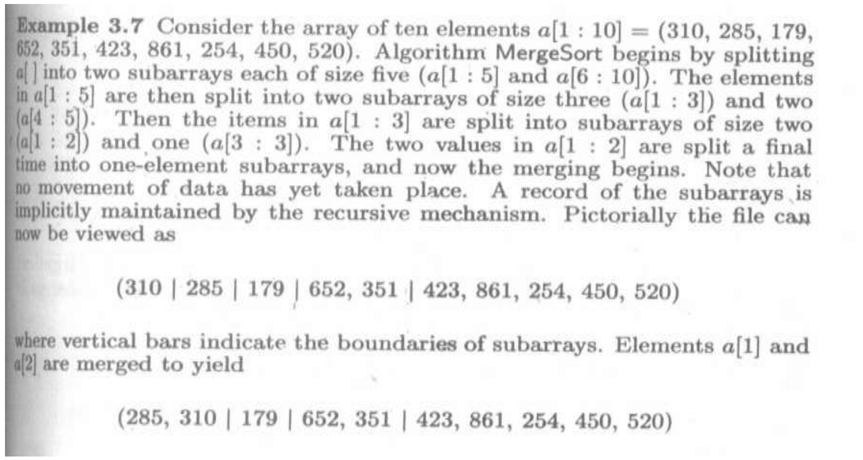
MergeSort algorithm describes this process very succinctly using recursion and a function Merge which merges two sorted sets.



```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one elem
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Algorithm 3.7 Merge sort

Complexity Analysis of Merge sort

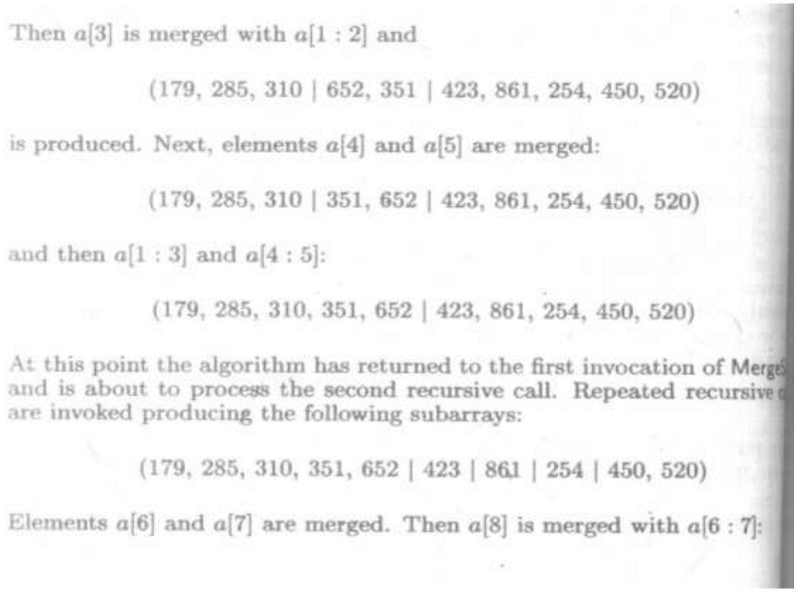


Example 3.7 Consider the array of ten elements $a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$. Algorithm MergeSort begins by splitting $a[\,]$ into two subarrays each of size five ($a[1:5]$ and $a[6:10]$). The elements in $a[1:5]$ are then split into two subarrays of size three ($a[1:3]$) and two ($a[4:5]$). Then the items in $a[1:3]$ are split into subarrays of size two ($a[1:2]$) and one ($a[3:3]$). The two values in $a[1:2]$ are split a final time into one-element subarrays, and now the merging begins. Note that no movement of data has yet taken place. A record of the subarrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

(310 | 285 | 179 | 652, 351 | 423, 861, 254, 450, 520)

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

(285, 310 | 179 | 652, 351 | 423, 861, 254, 450, 520)



Then $a[3]$ is merged with $a[1:2]$ and

(179, 285, 310 | 652, 351 | 423, 861, 254, 450, 520)

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

(179, 285, 310 | 351, 652 | 423, 861, 254, 450, 520)

and then $a[1:3]$ and $a[4:5]$:

(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)

At this point the algorithm has returned to the first invocation of Merge and is about to process the second recursive call. Repeated recursive are invoked producing the following subarrays:

(179, 285, 310, 351, 652 | 423 | 861 | 254 | 450, 520)

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6:7]$:

$$(179, 285, 310, 351, 652 \mid 254, 423, 861 \mid 450, 520)$$

Next $a[9]$ and $a[10]$ are merged, and then $a[6:8]$ and $a[9:10]$:

$$(179, 285, 310, 351, 652 \mid 254, 423, 450, 520, 861)$$

At this point there are two sorted subarrays and the final merge produ
the fully sorted result

$$(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)$$

Following tree represents the sequence of recursive calls that are produced by MergeSort when it is applied to ten elements.

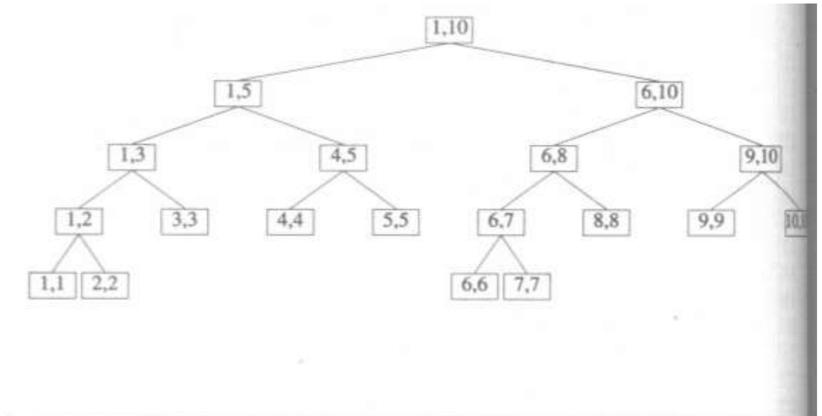


Figure 3.3 Tree of calls of MergeSort(1, 10)

The given figure is a tree representing the calls to procedure Merge by MergeSort

If the time for the merging operation is proportional to n, then the co
puting time for merge sort is described by the recurrence relation

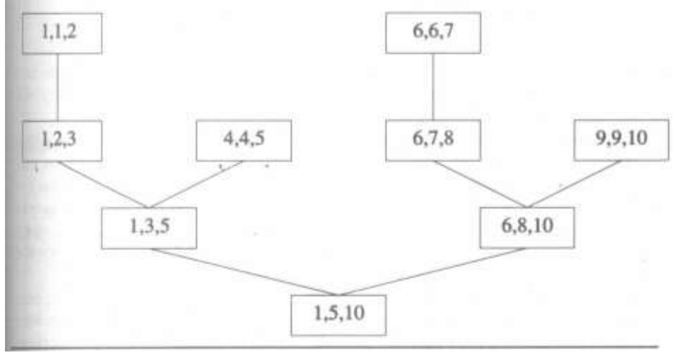
$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^kT(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

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



Limitations

- It uses 2n locations.
- Additional stack space for recursion

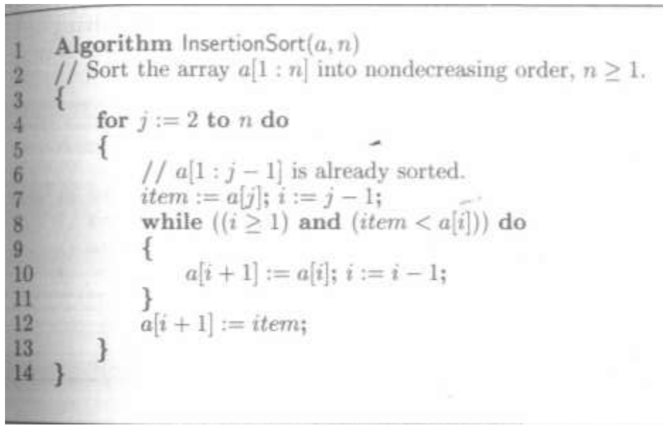
For small set sizes most of the time will be spent processing recursion instead of sorting. Solution to this problem is not allowing the recursion to go to the lowest level. So there is a second sorting algorithm Insertion sort for small sizes. Insertion sort works fast on arrays of less than 16 elements. The basic ideas of sorting is

```
for j:= 2 to n do
{
```

place a[j] in its correct position

in the sorted set a[1: j-1];

}

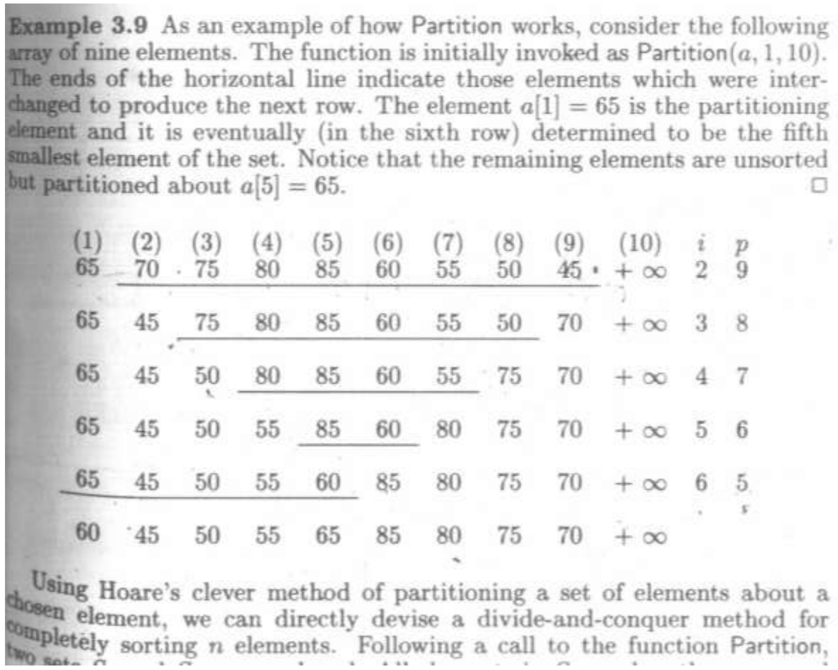


```
Algorithm InsertionSort(a, n)
// Sort the array a[1 : n] into nondecreasing order, n ≥ 1.
{
    for j := 2 to n do
    {
        // a[1 : j − 1] is already sorted.
        item := a[j]; i := j − 1;
        while ((i ≥ 1) and (item < a[i])) do
        {
            a[i + 1] := a[i]; i := i − 1;
        }
        a[i + 1] := item;
    }
}
```

The worst case computing time for this insertion sort is Θ(n^2) and best case time is Θ(n) because body of while loop is never entered in best case.

Quick Sort

In quick sort division into two sub arrays is made but sorted sub arrays do not need to be merged later. This is accomplished by rearranging a [1: n] such that a[i] ≤ a[j] for all i between 1 and m and all j between m+1 and n for some m, 1 ≤ m ≤ n. Rearrangement is performed by picking some elements of a[], t= a[s], then reorder the other elements, so that all elements appearing before t in a[1:n] are ≤ t and all elements appearing after t are ≥ t. This rearranging is referred to as **partitioning**.

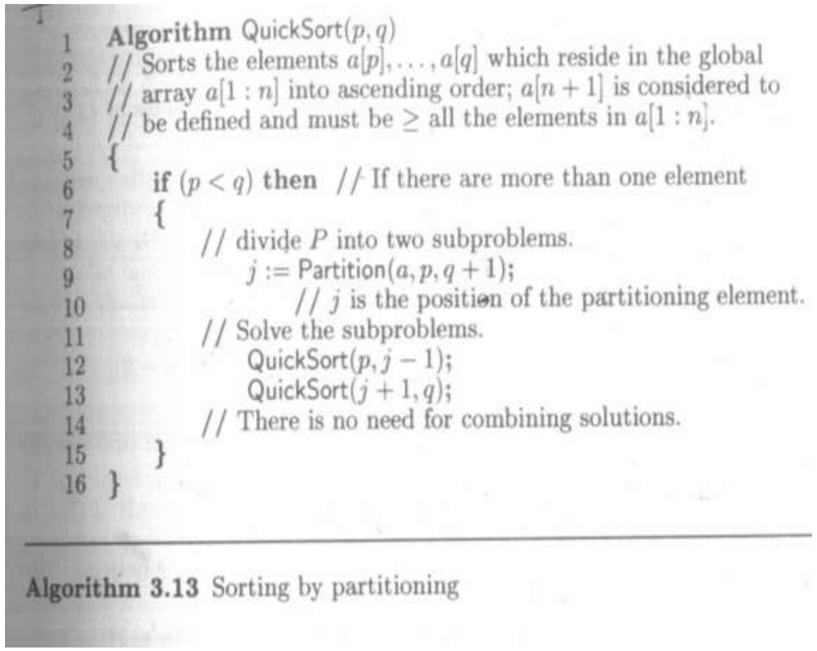


Example 3.9 As an example of how Partition works, consider the following array of nine elements. The function is initially invoked as Partition$(a, 1, 10)$. The ends of the horizontal line indicate those elements which were interchanged to produce the next row. The element $a[1] = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about $a[5] = 65$. □

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

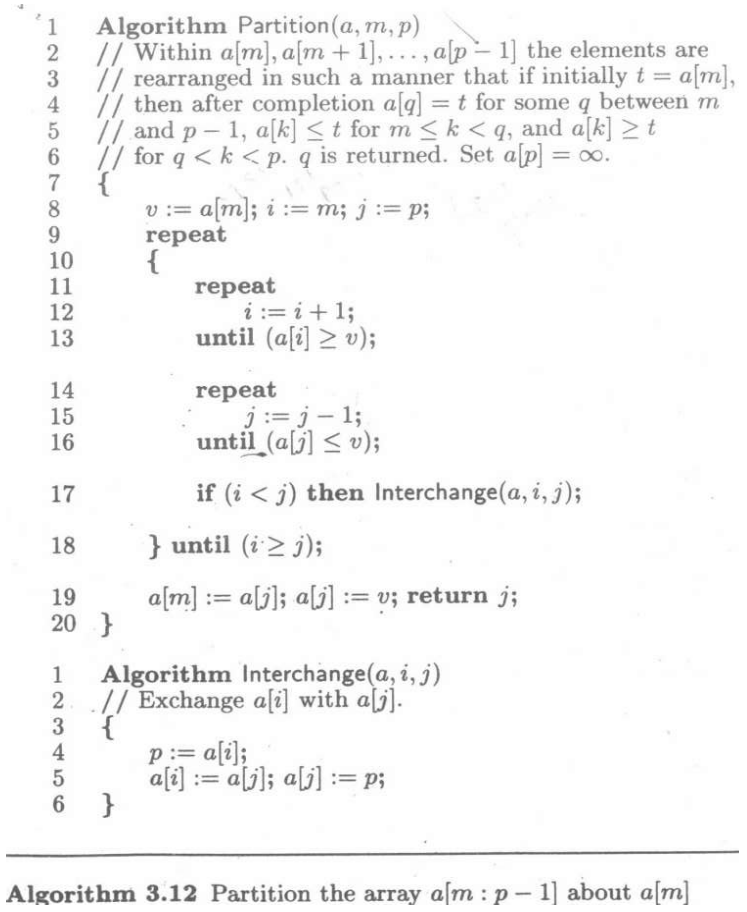
Using Hoare's clever method of partitioning a set of elements about a chosen element, we can directly devise a divide-and-conquer method for completely sorting n elements. Following a call to the function Partition,

Following a call to function partition, two sets S1 and S2 are produced. All elements in S1 are ≤ elements in S2. Each set is sorted by reusing function partition.



```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j :=$ Partition($a, p, q+1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j-1$);
            QuickSort($j+1, q$);
        // There is no need for combining solutions.
    }
}
```

Algorithm 3.13 Sorting by partitioning



```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}

Algorithm Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

Algorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

Worst case time is O (n2) and average time is only O (nlog n).

Stack space for recursion is O (log n) because worst case maximum depth of recursion may be n-1.

Performance measurement

QuickSort and MergeSort were evaluated on a SUN workstation 10/30. Each data set consisted of random integers in the range (0, 1000). Following tables show the results in milliseconds.

Scanning the above tables, we immediately see that QuickSort is faster than MergeSort for all values. Even though both algorithms require O(n log n) time on the average, QuickSort usually performs well in practice.

Selection

From given n elements a[1:n] determine the kth smallest element. Following algorithm shows the working of this.

```
Algorithm Select1(a,n, k)
//selects the kth smallest element  in a[1:n] and places it
//in the kth position of a[].  The remaining elements are
//rarranged such that $a[m] \le a[k]$ for $1 \le m < k$ and
//$a[m] \ge a[k]$ for $k < m \le n$.
{
low:=1; up := n +1;
a[n+1] := ∞;  // a[n+1] is set to infinity
repeat
{
//each time the loop is entered,
//$1 \le low \le k \le up \le n+1$
j:= Partition (a, low, up);
// j is such that a[j] is the jth smallest  value in a[]
If (k=j) then return;
else if (k<j) then up := j; // j is the new upper limit
      else low := j +1; // j+1 is the new lower limit
} until (false);
}
```

The above algorithm places the kth smallest element into position a[k] and partitions the remaining elements so that $a[i] \le a[k]$, $1 \le i < k$, and $a[i] \ge a[k]$, $k < i \le n$.

Assumptions from Select1 that were made for QuickSort are:

1. The n elements are distinct
2. Partition element has equal probability to can be ith smallest

Partition requires O(p – m) time m increases by at least one and j decreases by at least one.

Worst case complexity of Select1 is O(n2) .

Average computing time is O(n)

The space needed by Select1 O(1).

Strassen's Matrix Multiplication

Let A and B be two n x n matrices. To compute C(i, j) = $\sum_{1 \le k \le n} A(i,k)B(k,j)$

This equation needs n multiplications to compute C(i, j). The matrix C has n^2 elements, time for matrix multiplication is $\Theta(n^3)$ by conventional method. Divide and Conquer strategy suggests another way to compute the product. Assume n is the power of 2. When n is not a power of two, rows and columns of zeros can be added to both A and B, resulting dimensions are a power of two.

Imagine A and B are each partitioned into four square sub matrices, each have n/2 x n/2 dimensions. Product of AB can be calculated by the above formula for the product of 2 x 2 matrix.

That is, $C_{11} = A_{11} B_{11} + A_{12} B_{12}$

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

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

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

To compute AB using above equations we need to perform 8 multiplications and four additions of n/2 x n/2 matrices. So over all computing time T(n) by divide and conquer algorithm is

T (n) = {

where b an

$$\begin{matrix} b & n \le 2 \\ 8\,T(n/2) + Cn^2 & n > 2 \end{matrix}$$

This recurrence can be solved the same way as earlier recurrence T(n)= O (n^3). No improvement over conventional method because matrix multiplications are more expensive than matrix addition. Reformulate the equation for Cij with fewer multiplications and more additions.

Volker Strassen discovered a way to compute Cij's of using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven n/2 x n/2 matrices.

P, Q, R, S, T, U and V.

P= (A11 + A22) (B11 + B22)

Q= (A21 + A22) B11

R= A11 (B12 – B22)

S= A22 (B21 – B11)

T= (A11 + A12)B22

U= (A21 – A11) (B11 + B12)

V= (A12 – A22) (B21 + B22)

C11 = P + S – T + V

C12 = R + T

C21 = Q + S

C22 = P + R – Q + U

The resulting recurrence relation for T (n) is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7\,T(n/2) + an^2 & n > 2 \end{cases}$$

where a and b are constants. Working with this formula, we get

$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$