Name :- ______________________

AOAD

PART 1

• NOTES •

Contents

||Sadguru's|| .....always a step ahead of others

Chapter 1 • Introduction to Algorithm Analysis

The significance of the Algorithm Analysis is for the selection of the best suited algorithm out of the available ones. There can be many measures possible to evaluate an algorithm, but the two most significant are time and space. As the factors like machine speed, type of instruction set used etc. do come in the way of analysis hence the method of analyzing should try to avoid such factors by adopting proper assumptions.

Time complexity basically considers the time required for the execution of the program or to reach to a solution. Space complexity further has two considerations like code size and the data size. Generally, the program size remains fix whereas the amount of data that can be used is unpredictable. Even the facility of dynamically allocating and de-allocating data adds to the complexity. As stated above, when two algorithms are executed on different machines, it is difficult to compare them because the two machines may be running with different speed. Then a worst algorithm may take less time as compared to the best algorithm running on a slow machine. Another problem can be of machines having the different instruction sets.

All the above discussions lead to the need of machine independent model of computation. With the assumption that the two machines are using the same set of instructions, the analysis can be done by computing number of instruction steps used for the algorithm. Given an algorithm, the first task is to determine which operations are employed and their relative costs. These operations may include the four basic arithmetic operations on integers like addition, subtraction, multiplication and division. Other basic operations might include arithmetic on floating point numbers, comparisons assigning values to variables and executing procedure calls. These operations take a fixed amount of time and hence we can say that they are bounded by a constant. But this is not true for all operations of a computer. Some may be composed of an arbitrarily long sequence of more basic operations. For example, a comparison of two character strings may use a character compare instruction which may, in turn, use a shift and bit-compare instruction. The total time for the comparison of two strings will depend upon their lengths, while the time for each character compare is bounded by a constant.

One more important task is to determine the set of input for the algorithms so that to test all possible behavior. Here we need to understand the working of the algorithm and determine proper inputs for the best or worst or typical behavior. In producing a complete analysis of the computing time of an algorithm, we distinguish between two phases: a priori analysis and a posteriori testing. In priori analysis we obtain a function which bounds the algorithm's computing time. In a posteriori testing we collect actual statistics about the algorithm's consumption of time and space while it is executing. Suppose, there is a statement x = x+ y somewhere in the middle of a program. We wish to determine the total time that statement will spend executing. Here we need two important items of information. First is the number of times the statement is going to execute or frequency and the time required for a single execution. Product of these two will give us total time. Since the time per execution depends on both the machine and the programming language together with its compiler, a priori analysis limits itself to determine the frequency count of each statement. This can be directly calculated from the algorithm independent of machine and the programming language used together with the compiler.

Let's take an example of 'finding the maximum' program. The algorithm goes as follows :-

```
1)  Read n
2)  read num
3)  max = num
4)  for (i=1; i<n; i++)
        a)  read num
        b)  if max < num then max = num
5)  display max.
```

The tabulated information of the number of iterations of each step is given below.

Step	No. of Iterations
Read n	1
Read num	1
Max = num	1
i=1	1
i<n	n
i++	n-1
Read num	n-1
If max <num	n-1
max = num	n-1
Display max	1
Total	5n+1

Here we assumed the worst case, hence comparison is assumed to be done n-1 times. Hence the time complexity of the algorithm can be taken as 5n+1. But consider the following example.

Program 1 → 5n +3

Program 2 → 7n – 2

Suppose both the programs contains some read, assign, add and multiply instructions. Here it seems that P1 will take less time as compared to P2. But we can not compare them like this. Because it can be this way, add takes less time than compare and P1 has very few reads but many compare instructions. We have counted both read and compare as 1 only so even though P1 has less complexity it may take more time. Hence the assumption is needed that every instruction takes unit time and can be considered uniformly. So for 5n+3 and 7n-2 we can say that both oft then are proportional to input, hence they will take time proportional to input i.e. n.

But when we have P1 → 5n + 200 and P2 → $2n^2$ – 3n +1 and when n is sufficiently large then surely P1 is faster than P2. So in general when,

P1 → C1n + C2

P2 → $C3n^2$ – C4n + C5

Where C1, C2, C3, C4, C5 >0 and for n > n_0, P1 is faster than P2.

All the above discussion leads to the need of machine independent model of computation.

Asymptotic Comparison

Lets take one example of two programs P_1 and P_2 where

$$P_1 \rightarrow 50n+200$$

$$P_2 \rightarrow 2n^2-3n+1$$

For smaller values of n, P_2 may seem faster than P_1 because of the low co-efficient, but as n is appreciably increased it is clear that P_1 would be always faster. It shows that for small vales of n, constants play a vital role in analysis which is not desirable. The effect of constants can be dropped or at least minimized by taking larger values of n as seen in the above example.

$$P_1 \rightarrow C_{1n}+C_2$$
$$P_2 \rightarrow C_3 n^2+C_{4n}+C_5$$

Then for $n > n_0$, P_1 will be faster than P_2 provided $C_1, C_2, C_3, C_4, C_5 > 0$.

Hence comparing the complexity of algorithms when input is large is known as 'Asymptotic Comparison'.

Suppose we have two functions $f(n)$ and $g(n)$ then,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \rightarrow 0 \qquad f(n) \text{ better than } g(n)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \rightarrow \infty \qquad g(n) \text{ better than } f(n)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \to const$$ Same complexity

This is the terminology when n is large but when n is small we have to find exact constants.

Asymptotic Measurement

Generally we measure the time complexity (in terms of steps) as a function of the input size. Say $f(n) = 12n^3 + 15n \log n - 5n^{\log n}$ where $f(n)$ is time complexity function and n is input size. $\mathrm{f(n) = 5n - 3}$.

Now suppose we are sorting 5 numbers like 12, 15, 13, 9, 7 and hence $n = 5$. But if the same input is given as a character string like 12 # 15 # 13 # 9 # 7 $ then $n = 1$ and the comparison becomes senseless because complexity is proportional to length of the string.

Concept of Input size

The concept of input size is the most important thing to handle in analysis. Suppose we have to do addition of two integers like $x = a + b$. We say we will require one unit time, assuming a fixed constant data size. But suppose we need to add two arbitrary size numbers, then analysis will change suppose we can add 1 nibble of data at a time and input is of n nibble then it will take $2n-1$ time, as first addition will take unit time and remaining $(n-1)$ times we will require to add with carry, so it will take twice the unit time. Hence,

$$\begin{aligned} f(n) &= 2(n-1) + 1 \\ &= 2n - 2 + 1 \\ &= 2n - 1 \end{aligned}$$

	Carry		Carry
	6813	5372	2184
+	2410	0357	3852
	2 unit	2 unit	1 unit

...n times

As given in earlier discussion, even if input is given as a string 12 # 15 # 13 # 9 # 7 $

We will take it as n multiples of basic input size hence here n = 5. Analysis is simple for smaller values of n. It becomes difficult in the worst case for all valid inputs.

$$f_1(n) = 305n^2 + 3n + 2$$
$$f_2(n) = n^3 - 8n^2 + 8n + 9$$
$$f_3(n) = 3000n \log n + 9$$

When n is small $f_2(n)$ is better. As it increases appreciably high, $f_1(n)$ is better but for large values of n, $f_3(n)$ is better.

So, when n tends to infinity, constants do not matter and that is good because they have come out of lot of approximation. It is only the highest order term that matters. Hence analyzing algorithms for their behavior and comparing them for the larger values of inputs is known as Asymptotic Analysis.

Asymptotic Complexity

f(n) and g(n) are both positive going functions. It can be seen that

till nA	g(n) > f(n)
till nB	f(n) > g(n)

So if two functions keep on toggling over each other then one cannot decide which one is better. But here after n_0, it is clearly seen that f(n) > g(n). Hence we can say that g(n) is better than f(n) asymptotically i.e. after n0, f(n) ≥ g(n).

Asymptotic Comparison

For smaller values of n, the constants play significant role in the complexity values. Those constants are the byproduct of the assumptions done and are required to be overcome. As the value of n increases, the effect of constants gets vanished. We can see functions fluctuating for the smaller values of n, attains a proper rate as the value of n is increased. Hence the comparison of the algorithms is done for the higher values of inputs, such comparison is known as Asymptotic Comparison. A priori analysis of computing time ignores all of the factors which are machine or programming language dependent and concentrates on determining order of the magnitude of the frequency of execution of statements. The functions used for the asymptotic comparison are known as asymptotic growth functions. Following are some of the asymptotic growth functions used in algorithm analysis.

1) Big O (Upper Bounding) Function

2) Big Ω (Lower Bounding) Function

3) Theta θ (Order) Function

Asymptotic Growth Functions

1) Big O (Upper Bounding) Function

A function f(n) = O (g (n)) (read as "f of n is big oh of g of n") if and only if, there are two positive constants c and n0 such that, |f(n)| ≤ c . |g(n)| for all n ≥ n0.

Suppose, g(n) = 502n2+3n+7 and f(n) = 715 n2 + 100n + 10. Here by making c = 1 and n0=1, we get, 0 ≤ g(n) ≤ c . f(n).

Hence g(n) = O (f(n)) and, it is said that g(n) is upper bounded by f(n). If g(n) = 502n2 + 3n - 7 and f(n) = n3 + 200n2 - 5. Now we want to prove g(n) = O(f(n)) then c =1 and n0 =502. If f(n) = 0.3n3 + 200n2 – 5 then we can say f(n) = O(n3). Hence the highest power term matters. Now suppose g(n) = 502n2 + 3n - 7 and f(n) = n2 – 7n - 15. Then we can show g(n) = O(f(n)) by taking c = 200000 and n =1 i.e. g(n) = O(n2).

But f(n) = 7n + 5 and g(n) = $502n^2 + 3n - 7$. Now even if we take c = 200000, it cannot be proved that there is some n_0 such that g(n) ≤ c . f(n) for all n > n_0. Hence g(n) ≠ O(f(n)).

Suppose we are determining the computing time, f(n) of some algorithm. The variable n might be the number of inputs or outputs or their sum. Since f(n) is machine dependent, a priori analysis will not suffice to determine it. However, a priori analysis can be used to determine a g(n) such that f(n) = O(g(n)). When we say that the algorithm has computing time O(g(n)) we mean that of the algorithm is run on some computer on the same type of data but increasing values of n, the resulting times will always be less than some constant times |g(n)|.

The most common computing times for algorithms we will see here are

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

Here O(1) means the number of executions of basic operations is fixed and hence total time is bounded by a constant. It shows that the computing time is independent of n i.e. the number of inputs. The first six orders of magnitude have an important property in common, they are bounded by a polynomial, O(n), $O(n^2)$ and $O(n^3)$ are themselves polynomials referred by their degrees: linear, quadratic and cubic. However there is no integer m such that n^m bounds 2^n or $2^n \neq O(n^m)$ for any integer m. The order of this formula is $O(2^n)$.

Generally, the exponential algorithms require so much time, that neither subsequent improvements in the speed of sequential computers nor improvements which effect the leading constant of the computing time, will ever produce a much greater range of solvable problem size. One possible recourse is to devise new algorithms with much improved orders of magnitude.

So far we have concentrated on O-notation as a means of describing an algorithm's performance. Whereas O-notation is used to express an upper bound, we might also wish to determine a function which is a lower bound. What is needed is a mathematical notation for expressing a formula which is a lower bound on the computing time of an algorithm to within a constant.

2) Omega (Lower Bounding) Function

A function f(n) = Ω (g(n)) (read as "f of n is omega of g of n") if and only if there are two positive constants c and n_0, such that, $0 \le c\,.\,|g(n)| \le f(n)$ for all $n > n_0$.

So when f(n) = Ω (g(n)), it is said that f(n) is lower bounded by g(n).

In some cases the time for an algorithm, f(n), will be such that f(n) = Ω g(n) and f(n) = O(g(n)). For this circumstance we will use the following notation.

3) Theta (Order) Function

A function f(n) = θ (g(n)) (read as "f of n is order of g of n") if and only if there are positive constants c_1, c_2 and n_0 such that $0 \le c_1.\,|g(n)| \le |f(n)| \le c_2\,.\,|g(n)|$ for all $n > n_0$.

If f(n) = θ (g(n)) then g(n) is both an upper and lower bound on f(n). This means that the worst and best cases require the same amount of time to within a constant factor. As an example consider the algorithm which finds the maximum of n elements. The computing time for this algorithm is both O(n) and Ω(n) since the for loop always makes n-1 iterations. Thus, we say that its time is θ (n). The algorithm for searching an array of n elements for a single value has computing time which is O(n) but Ω(1). In the best case it might find the value on the first comparison, but in the worst case it will look at all elements once.

An even stronger mathematical notation is given by the following.

4) Small o (Asymptotic) Function

A function f(n) ~ o(g(n)) (read as "f of n is asymptotic to g of n") if and only if

limit f(n) / g(n) → 1 as n → ∞

Since the ratio in the limit is one, the functions f(n) and g(n) must agree even closer than by a constant. If there is an algorithm whose exact computing time is f(n) and we can determine a g(n) such that f is asymptotic to g, then we will have a more precise description of the computing time than if we had used the big O-notation. In practice it implies that we will know both the order of the leading term and its constant. For example, if $f(n) = a_k n^k + \dots + a_0$ then

$f(n) = O(n^k)$

and

$f(n) \sim o(a_k n^k)$.

Randomized Algorithms

In many cases, we require to take decision in random manner. Using a normal algorithm we can follow a pattern but that will not be random in true sense. A good example is a professor asking questions to students on an assignment. The student to ask question should be selected with a random pattern. There we can not apply criteria to select bright students or students with odd roll numbers etc., as it will soon become predictable. So what we use is called 'Randomized Algorithms'. At least once during the algorithm, a random number is used to make a decision. The running time of the algorithm depends not only on the particular input but also on the random numbers that occur.

Generally the worst case running time of a randomized algorithm is almost same as a nonrandomized algorithm. Consider two variants of quick sort. Variant A uses the first element as pivot element whereas variant B uses randomly selected element as pivot element. In both case the worst case running time can be of the order of n^2 as randomly chosen element may always turn out to be the largest one. Variant A will run for $O(n^2)$ every time it is given already sorted input. But if variant B is presented to the same input twice, it may have two different running times; depending on what random numbers occur. So the effect of already sorted input is nullified because of randomization.

We know that already sorted inputs cause problem in quick sort and binary search tree. By using the randomized algorithm the particular input is no longer important. The random numbers are important, and we can get expected running time, where we now average over all possible random numbers instead of over all possible inputs. Using quick sort with a random pivot gives an O(n log n) expected time algorithm. This means that, for any input including already sorted input, the running time is expected to be O(n log n), based on the statistics of random numbers.

There are many applications of randomized algorithms. It helps in supporting the binary tree operations in O(log n) expected time. The other application of randomized algorithm is to test the primality of large numbers. No efficient nonrandomized algorithms are known for this problem.

Random Number Generators

Since the randomized algorithms require random numbers, we must have methods to generate them. Actually true randomness is virtually impossible to do on a computer since these numbers will depend on the algorithm and thus cannot be possibly random. Generally, it suffices to produce pseudo-random numbers which are numbers that appear to be random. The standard method to generate random numbers is the linear congruential generator. Numbers X_1, X_2, ... are generated satisfying

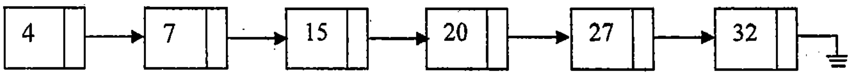
$X_{i+1} = a X_i \bmod m$

To start the sequence, some value of X_0 must be given. This value is known as the seed. If $X_0 = 0$, then the sequence is far from random, but if a and m are correctly chosen, then any other $X_0 < m$ is equally valid. If m is prime then X_i is never 0. As an example, if m = 11, a= 7 and $X_0 = 1$. then the numbers generated are

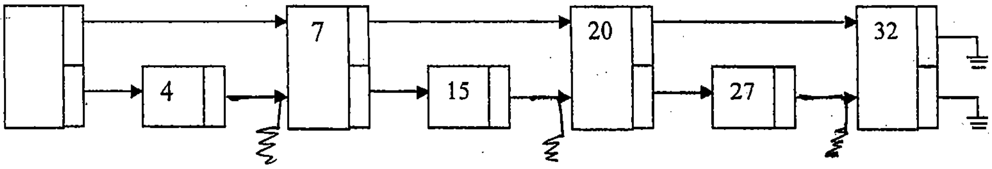
7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2, ...

Skip Lists

Skip Lists is a data structure with the use of randomization which supports search and insert operations with the computing time of O(log n). This means the running time for each operation on any expected input has expected value O(log n), where the expectation is based on random number generator. It is possible to add deletion and all the operations that involve ordering and obtain expected time bounds that match the average time bounds of binary search tree. The simplest possible data structure to support searching is linked list. The figure shows simple linked list. The time to perform search is proportional to the number of nodes that have to be examined, which is at most n.



The figure below shows a linked list in which every other node has an additional pointer to the node two ahead of it in the list. Because of this, at most [n/2]+1 nodes are examined in the worst case.



We can extend this idea and obtain a linked list in which every fourth node has a pointer to the node four ahead. Only [n/4] +2 nodes are examined. The limiting case would be that every 2^i th node has a pointer to the node 2^i ahead of it. The total number of pointers has only doubled, but now at most [log n] nodes are examined during a search. The total time spent in searching is O(log n) because search consists of either advancing to a new node or dropping to a lower pointer in the same node. Each of these steps consumes at most O(log n) total time during a search. Notice that the search in this data structure is essentially a binary search.

Here we define level k node to be a node that has k pointers. The i^{th} pointer in any level k node (k >= i) points to the next node with at least I levels. Even there is a restriction that the i^{th} pointer points to the node 2^i ahead. This condition becomes too rigid for an efficient insertion. Hence in skip list this condition is dropped. For insertion, we allocate a new node fort it and the level of the node is decided by the randomization. The random number generated should within the range of the levels allowed. To perform a find in skip list, we start at the highest pointer at the header. We traverse along this level until we find that the next node is greater than the one we are looking for (or null). When this occurs we go to next low level and continue the same strategy. When progress is stopped at level 1, we are either in front of the node we are looking for, or it is not in the list. To perform an insert we proceed as in a Find, and keep a track of each point where we switch to a lower level. The new node whose level is then determined randomly, is then spliced into the list.

Recursive Algorithms

An object is said to be recursive if it partially consists of is defined in terms of itself. Recursion is particularly powerful means in mathematical definitions. A few familiar examples are natural numbers, tree structures, factorial, Fibonacci and gcd. The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions. Recursive algorithms, however, are primarily appropriate when the problem to be solved, or the function to be computed, or the data structure to be processed are defined in recursive terms. In general a recursive program can be expressed as a composition C of base statements Si (not containing P) and P itself.

$$P \equiv C\ [Si, P]$$

The necessary and sufficient tool for expressing programs recursively is the procedure or subroutine, for it allows a statement to be given a name by which this statement may be invoked. If a procedure P contains an explicit reference (call) to itself, then it is said to be directly recursive. If P contains a reference (call) to another procedure Q which contains a direct or indirect reference (call) to P, then P is said to be indirectly recursive. The use of recursion may therefore not be immediately apparent from the program text.It is common to associate a set of local objects with a procedure i.e., a set of variables, constants types and procedures which are defined locally to this procedure and have no existence or meaning outside the procedures. Each time such a procedure is activated recursively, a new set of local, bound variables is created. Although they have same names as their corresponding elements in the set of local to previous instance of the procedure, their values are distinct and any conflict in naming is avoided by the rules of scope of identifiers. The identifiers always refer to the most recently created set of variables. The same rule holds for procedure parameters which by definition are bound to the procedure.

Like repetitive statements, recursive procedures introduce the possibility of non-terminating computations and thereby also the necessity of considering the problem of termination. A fundamental requirement is evidently that the recursive call of a procedure P is subjected to a condition B, which at some time becomes non-satisfied. The scheme for recursive algorithms may therefore be expressed more precisely as

$$P \equiv \text{ if B then } C\ [Si, P]$$

OR

$$P \equiv C\ [Si, \text{if B then } P]$$

A basic technique of demonstrating that a repetition terminates is to define a function f(x) (x is the set of variables in the program), such that f(x) <= 0 implies the terminating condition and to prove that f(x) decreases during each repetition. In the same manner, termination of recursive program can be proved by showing that each execution of P decreases f(x). a particularly evident way to ensure termination is to associate a parameter say n, with P and to recursively call P with n-1 as parameter value. This may be expressed by the following.

$$P(n) \equiv \text{ if } n > 0 \text{ then } C\ [Si, P(n-1)]$$

$$P(n) \equiv C\ [Si, \text{if } n > 0 \text{ then } P(n-1)]$$

In practical applications it is mandatory to show that the ultimate depth of recursion is not only finite, but that it is actually small. The reason is that upon each recursive activation of a procedure P some amount of storage is required to accommodate its variables. In addition to these local bound variables, the current state of computation must be recorded in order to be retrievable when the new activation of P is terminated and the old one has to be resumed.

■ ■ ■

Chapter 2 • Divide and Conquer

The General Method

Given a function to compare on n inputs, the divide and conquer strategy suggests splitting the inputs into k distinct subsets, 1 < k <= n yielding k sub problems. These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub-problems are still relatively large then the divide and conquer strategy can be reapplied. Often the sub-problems resulting from a divide and conquer strategy are of the same type as the original problem. For those cases the reapplication of the divide and conquer principle is naturally expressed by a recursive procedure. Now smaller and smaller sub-problems of the same kind are generated, eventually producing sub-problems that are small enough to be solved without splitting.

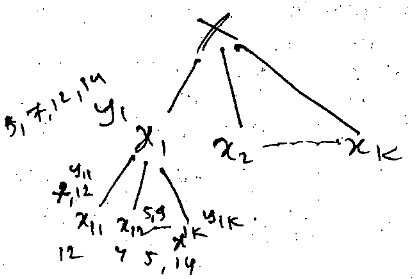
In general, the problem is divided or decomposed into smaller sub-problems till the generated sub-problems are small and solvable by the defined function. Then these sub-problems are solved independently to obtain the number of sub-solutions. Finally all the sub-solutions are combined to get the solution of the original problem.

Let's consider one general example.

Suppose 'basic' is a function (algorithm) which is capable of working on simple sub-problems. So we can consider 'basic' as sub-algorithm that can be used to solve sub-problems of the original problem independently. Then the general procedure for 'Divide and Conquer' can be expressed as follows.

Function DC (x)

```
If x is sufficiently small or simple
then return basic (x)
else decompose x into smaller instances
$x_1, x_2, \dots x_k$.
For i=1 to k do
$y_i$= DC ($x_i$)
recombine all $y_i$'s to obtain solution y to problem x.
return y.
```



Here first x is tested whether it is sufficiently small to be directly solved by the function basic, if yes then it is solved and the solution is returned. If we see carefully, this will act as a terminating condition for sub-problems which would be solvable. If x is still not suffinciently small then it is decomposed into some k instances x1 to xk which can be considered as k sub-problems. Then the same function DC is called with each of the sub-problem and a series of solutions are obtained as y1 to yk. It would be important to note that the sub-problems x1 to xk may also not be small and they may even be further divided. Then these solutions y1 to yk are combined or integrated to form a single solution y which is the expected solution of problem x. it is clearly visible the use of recursion in the divide and conquer strategy.

This is just the outline of "Divide and Conquer", all the DC algorithms may not follow this exactly. Some of them require to solve the first sub problem before formulating the second etc. Some of these algorithms use recursion techniques to divide as well as recompose efficiently. But for instance size n, recursive algorithm uses stack of size log n which can go up to n itself. Sometimes it is possible to replace the recursive DC algorithm by equivalent iterative code which is usually efficient for time.

As we see the divide and conquer strategy divides a problem into sub-problems. Assuming that it divides the problem into two sub-problems of nearly equal size then the computing time can be expressed with the help of following recurrence relation.

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

where T(n) is the time for DC algorithm on n inputs, g(n) is the time to compute the answer directly for small inputs and f(n) is the time for DIVIDE and COMBINE.

Binary Search

Let Ai, 1<=i<=n be a list of elements which are sorted in non-decreasing order. Consider the problem of determining whether a given element x is present in the list. In case x is present the function returns true otherwise false. It can even be designed to get the index of the element x if it is present. Divide and conquer suggests breaking up the instance A(1:n) into sub-instances. One possibility is to pick an index k and obtain three instances I1= A(1:k-1), I2 = A(k) and I3 = A(k+1:n). The search problem for two of these three is easily solved by comparing x with A(k). If x = A(k) then the function can return true and I1 and I3 need not be solved. If x < A(k), then only I1 remains to be solved. If x > A(k) then I3 remains to be solved. After comparing with A(k), the instance remaining to be solved (if any) can be solved using this divide and conquer scheme again. If k is chosen such that A(k) is always a middle element i.e. k = (1+n)/2 then the resulting search algorithm is known as binary search.

Algorithm

```
Function binsearch(A, n, x) returns Boolean
        A → array of elements (1:n) in non-decreasing order
        n → number of elements
        x → the element to be searched
integers → low, high, mid
1) start
2) assign low = 1, high = n
3) if low > high then
        return false   // element x is not found
4) mid = (low + high)/2
5) if A(mid) = x then
        Return true   // element x is found .. the position is mid
6) if x < A(mid) then
        high = mid -1
7) if x > A(mid) then
        Low = mid +1
8) go to step 3
```

In the above algorithm, we must be sure that all the operations such as comparisons between x and A(mid) are well defined. If the elements of A are integers, real or character strings then the comparisons should be carried out accordingly. We observe that 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 integers approaching each other and eventually low will become grater than high causing termination in a finite number of steps if x is not present. Even if x is present after certain steps that will be tracked by some A(mid) calculated and search terminates.

Working

Let us take the list of 7 elements stored in A(1:7) as follows.

-8, -5, -1, 7, 19, 27, 32

The element to search is x = 15. The following table summarizes the steps taken by the algorithm.

Low	high	mid	Action
1	7	4	A(mid) = 7 x > A(mid), hence low = mid +1 = 5
5	7	6	A(mid) = 27 x < A(mid), hence high = mid -1 = 5

5	5	5	A(mid) = 19 x < A(mid), hence high = mid – 1 = 4
5	4	--	low > high, hence return false 15 is not found.

The element to search is x = -5. The following table summarizes the steps taken by the algorithm.

Low	high	mid	Action
1	7	4	A(mid) = 7 x < A(mid), hence high = mid - 1 = 3
1	3	2	A(mid) = -5 x = A(mid), hence return true -5 is found

Analysis

The working of binary search can be explained with the help of binary search tree. It certainly follows the same flow. As we know the number of comparisons to search the element in binary search tree is equal to its depth. For a binary search tree of n nodes, the depth is given by $\log_2$ n. Hence even in case of binary search the number of element comparisons made would be equal to log n. There are two cases to be considered as successful search and unsuccessful search. In case of successful search, the best case complexity is O(1) whereas the average and worst case complexity is O(log n). In case of unsuccessful search all the three case give complexity of O(log n) as it has to traverse the complete branch.

In binary search for each pass, the array is divided into two sub-arrays. Hence if we have n elements then we require m passes such that n = 2^m. Hence number of passes m is equal to log n. In each pass we are performing two operations; one is comparing for equality and second is deciding which part of array to search next.

Hence total number of comparisons = 2 * m
= 2 * log n

Hence binary search is O(log n).

Implementation

```
public boolean binSearch(int low,int high,int ele)
{
 if  (low>high)
    return false;
 int mid=(low+high)/2;
 if (x[mid]==ele)
     return true;
 if (ele<x[mid])
     return (binSearch(low,mid-1,ele));
 else return (binSearch(mid+1,high,ele));
}
```

Finding Minimum and Maximum

Let us consider another problem that can be solved by the divide and conquer technique. The problem is to find maximum and minimum items in set of n elements. We have already seen a simple and straightforward solution to this problem but it can also be solve by applying divide and conquer strategy. Let us first see the straightforward algorithm.

```
Algorithm
Function minmax(A, n, max, min)
          A → array of elements (1:n)
          n → number of elements stored in A
          max → parameter passed to store maximum element found
          min → parameter passed to store the minimum element found
integer → i, n
1) start
2) set max = min = A(1)
3) for i = 2 to n do
          a) if A(i) > max then
                    max = A(i)
          b) if A(i) < min then
                    min = A(i)
4) return
```

In analyzing the time complexity of this algorithm, we shall concentrate on the number of element comparisons, because the count for the other operations in the above algorithm is of the same order as that of the element comparisons. More importantly, when the elements of the array are vectors, polynomials, very large numbers or strings of characters then the cost of element comparison is much higher than the cost of other operations. Hence the time required is determined by the total cost of element comparisons. The above algorithm requires 2(n-1) element comparisons in the best, average and worst cases. An immediate improvement is possible by realizing that the comparison A(i) < min is necessary only when A(i) > max is false. Hence the statement 'if A(i) < min' can be kept inside 'else'. Now the best case occurs when the elements are in increasing order. The number of comparisons is n-1. The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons is 2(n-1). On the average number of element comparisons is less than 2(n-1). On the average, A(i) will be greater than max half the time and so the average number of comparisons is 3n/2 -1.

Minmax using Divide and Conquer

A divide and conquer algorithm for this problem would proceed by dividing any instance I = A(1:n) into smaller instances. For example, I1 = A(1:n/2) and I2 = A(n/2+1, n). If MAX(I) and MIN(I) are the maximum and minimum of the elements in I then MAX(I) = the greater of MAX(I1) and MAX(I2), and MIN(I) = smaller of MIN(I1) and MIN(I2). If I contains only one element then the answer can be computed without any splitting. The algorithm for the same can be given as follows.

Algorithm

```
Function minmax (A, i, j, fmax, fmin)
          A → array of elements (1:n)
          i → starting index of the effective part of the array
          j → ending index of the effective part of the array
          fmax → the parameter passed to store the maximum element found
          fmin → the parameter passed to store the minimum element found
integer → mid, gmin, gmax, hmin, hmax
```

```
// min and max are assumed to be built-in functions which require one comparison to
// compute their result
1) start
2) if i = j  then
        a) fmax = fmin = A(i)
        b) return
3) if i = j-1
        a) if A(i) > A(j) then
                i) fmax = A(i)
                ii) fmin = A(j)
         else   i) fmax = A(j)
                ii) fmin = A(i)
        b) return
4) mid = (i + j) /2
5) call minmax (A, i, mid, gmax, gmin)
6) call minmax (A, mid+1, j, hmax, hmin)
7) fmax = max(gmax, hmax)
8) fmin = min(gmin, hmin)
9) return
```

The above procedure is initially invoked by the statement maxmin(1,n,x,y). 'max' and 'min' are functions that find the greater and smaller of two elements respectively. Note that each of these functions uses only one comparison per call.

We can simulate the above procedure using a set of inputs. A good way of keeping a track of recursive calls is to build a tree so that a node is added each time a new call is made.

Implementation

```
import java.util.*;
class Example
{
 public static int max (int a, int b)
 {
  if (a>b)
     return a;
  return b;
 }
public static int min(int a, int b)
{
 if (a<b)
   return a;
 return b;
}
 public static void maxmin(int x[ ], int i, int j, int fmaxmin[])
 {
  int mid,gmaxmin[]=new int [2],hmaxmin[]=new int[2];
  if (i==j)
```

```
     fmaxmin[0]=fmaxmin[1]=x[i];
  else if(i==j-1)
         if (x[i]>x[j])
         {
          fmaxmin[0]=x[i];
          fmaxmin[1]=x[j];
         }
         else
         {
          fmaxmin[0]=x[j];
          fmaxmin[1]=x[i];
         }
     else
     {
      mid=(i+j)/2;
      maxmin(x,i,mid,gmaxmin);
      maxmin(x,mid+1,j,hmaxmin);
      fmaxmin[0]=max(gmaxmin[0],hmaxmin[0]);
      fmaxmin[1]=min(gmaxmin[1],hmaxmin[1]);
     }
}
 public static void main(String args[ ])
 {
  int a[],n,fmaxmin[]=new int[2];
  Scanner src=new Scanner(System.in);
  System.out.println("Enter number of elements");
  n=src.nextInt();
  a=new int[n];
  System.out.println("Enter the elements");
  for (int i=0; i<n; i++)
  a[i]=src.nextInt();
  maxmin(a,0,n-1,fmaxmin);
  System.out.println("MAX = "+fmaxmin[0]);
  System.out.println("MIN = "+fmaxmin[1]);
 }
}
```

Analysis

Now we have to find the number of element comparisons. If T(n) represents this number, then the resulting recurrence relation is

$$T(n) = T(n/2) + T(n/2) + 2 \quad n > 2$$

$$1 \quad n = 2$$

$$0 \quad n = 1$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

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

By solving the above recurrence relation we get the total number of element comparisons as 3n/2 – 2. Compared with the previous straightforward method there is a saving of almost 25% in comparisons. It can be shown that no other algorithm based upon comparisons uses less than 3n/2 – 2 comparisons. So this revised maxmin procedure proves to be optimal. But in terms of storage it is worse than the straightforward algorithm, because it requires stack space for i, j, fmax and fmin. Given n elements there will be log n + 1 levels of recursion and we need to save five values for each recursive call (fifth is the return address).

Merge Sort

Another example of divide and conquer, we are going to see is a sorting algorithm having worst case complexity of O(n log n). This algorithm is called merge sort. For the worst case we have to assume that the set of keys are arranged in non decreasing order. Given sequence of n elements A(1:n), the general idea is to imagine them split into two sets A(1:n/2) and A(n/2+1:n). Each set is individually sorted and the resulting sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of the divide and conquer strategy where the splitting is into two equal size sets and the combining operation is the merging of two sorted sets into one. So we will see two procedures below. The first one is 'mergesort' which describes divide operation using recursion and the other is 'merge' which merges together two sorted sets.

Algorithm

```
Function mergesort (A, low, high)
        A(low : high) → array containing high – low + 1 (>=0) values
        low → index of the first element of the array A
        high → index of the last element of the array A
Integer → mid
1) start
2) if low > = high then
        return
3) mid = (low + high)/2
4) call mergesort (A, low, mid)
5) call mergesort (A, mid+1, high)
6) call merge (A, low, mid, high)
7) return

Function merge (A, low, mid, high)
        A(low : high) → array containing high – low + 1 (>=0) values
        low → index of the first element of the array A
        mid → index of the middle of the array A where it is split for sorting
        high → index of the last element of the array A
integer → i, j, k, temp(low:high)
1) start
2) i = low, j = mid +1, k = 0
3) while i <= mid and j <= high do
        a) if A(i) < A(j) then
                i) temp (k) = A(i)
                ii) j = j + 1
        else    i) temp (k) = A(j)
                ii) j = j+1
        b) k = k +1
4) while i <= mid do
        a) temp (k) = A(i)
        b) i = i +1, k = k +1
5) while j <= high do
        a) temp (k) = A(j)
        b) j = j +1, k = k +1
6) for i = low to high do
        A(i) = temp(i)
7) return
```

Working

Before executing mergesort, the n elements should be placed in A(1:n) and the auxiliary array temp(1:n) should also be declared. Then call mergesort (A, 1, n) will cause the keys to be rearranged into non-decreasing order in A.

Consider the array of nine elements A = (53, 27, 4, 83, 61, 52, 40, 11, 37). Procedure mergesort begins by splitting A into two subfiles of size five and four. The elements in A(1:5) are then split into two subfiles of size three and two. Then the items in A(1:3) are split into subfiles of size two and one. The two values in A(1:2) are split a final time into one elements subfiles and now the merging begins. Note that no actual movement of data has yet taken place. A record of the subfiles is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

(53 | 27 | 4 | 83 61 | 52 40 11 37)

With vertical bars indicating boundaries of subfiles. A(1) and A(2) are merged to yield

(27 53 | 4 | 83 61 | 52 40 11 37)

A(3) is merged with A(1:2) producing

(4 27 53 | 83 61 | 52 40 11 37)

Next, element A(4) and A(5) are merged

(4 27 53 | 61 83 | 52 40 11 37)

followed by merging of A(1:3) and A(4:5) to give

(4 27 53 61 83 | 52 40 11 37)

At this point the algorithm has returned to the first invocation of mergesort and it is about to process the second recursive call. Repeated recursive calls are invoked producing the following subfies.

(4 27 53 61 83 | 52 | 40 | 11 | 37)

A(6) and A(7) are merged similarly A(8) and A(9) are merged producing

(4 27 53 61 83 | 40 52 | 11 37)

then A(6:7) is merged with A(8:9)

(4 27 53 61 83 | 11 37 40 52)

At this point there are two sorted subfiles and the final merge produces the fully sorted result

(4 11 27 37 40 52 53 61 83)

Implementation

```
import java.util.*;
class Merge
{
 int x[ ], size;
```

```
public Merge(int size)
{
 Scanner src=new Scanner(System.in);
 this.size=size;
 x=new int[size];
 System.out.println("Enter "+size+" elements");
 for (int i=0; i<size; i++)
   x[i]=src.nextInt();
}
public void merge(int lb1, int ub1, int ub2)
{
 int temp[ ]=new int[20];
 int i,j,k,d;
 k=0;
 i=lb1;
 j=ub1+1;
 while (i<=ub1 && j<=ub2)
   if (x[i]<x[j])
     temp[k++]=x[i++];
   else
     temp[k++]=x[j++];
 while (i<=ub1) temp[k++]=x[i++];
 while (j<=ub2) temp[k++]=x[j++];
 for (i=lb1,j=0; i<=ub2;i++,j++)
    x[i]=temp[j];
}
public void mergeSort(int lb, int ub)
{
 int mid;
 if (lb<ub)
 {
  mid = (lb+ub)/2;
  mergeSort(lb,mid);
  mergeSort(mid+1,ub);
  merge(lb,mid,ub);
 }
}
public void display()
{
 for (int i=0; i<size; i++)
   System.out.print(x[i]+" ");
 System.out.println();
}
public int getSize()
```

```
 {
  return size;
 }
}
class MergeExp
{
 public static void main(String args[])
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter number of elements");
  int n=src.nextInt();
  Merge obj=new Merge(n);
  System.out.println("The original array");
  obj.display();
  obj.mergeSort(0,obj.getSize()-1);
  System.out.println("\nThe Sorted array");
  obj.display();
 }
}
```

Analysis

In mergesort the splitting continues until sets containing a single element are produced. If the time for the merging operation is proportional to n then the computing time for mergesort is described by the recurrence relation

$T(n) = a$ $\quad$ $n = 1$, a is constant

$\quad 2\,T(n/2) + cn$ $\quad$ $n > 1$, c is constant

When n is a power of two, $n = 2^k$, we can solve this equation by successive substitution. By solving this we get,

$T(n) = an + cn \log n$

Hence mergesort has the time complexity of O(n logn).

In simple way we can summarize as follows.

For n elements we require number of passes as m such that n = 2m.

So, $\log_2 n = m$.

Hence number of passes = m = log n

Number of comparisons in each pass = n.

Total number of element comparisons $= n\,m$

$= n \log n$

So time complexity of merge sort $= O(n \log n)$

Quick Sort

The divide and conquer approach may be used to arrive at an efficient sorting method different from mergesort. In mergesort, the file A(1:n) was divided at its midpoint into subfiles which are independently sorted and later merged. In quicksort, the division into two subfiles is made such that the sorted subfiles do not need to be later merged. This is accomplished by rearranging the elements in 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. Thus, the elements in A(1:m) and A(m +1 : n) may be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of A, say t = A(s), and then reordering the other elements so that all elements appearing before t in A(1:n) are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as partitioning.

Procedure partition accomplishes an in-place partitioning of the elements. For a list A(1:n), the first element is partitioning element. The partition element is placed in such a way that all the elements on the left of it are though unsorted but lesser than partition element and all elements to the right of it are though unsorted but grater than it. So the array is divided at the position of partition element and the two sub-arrays can be sorted independently. There is no need to merge the sub-arrays. The assumption that the first element is partition element id merely for convenience rather other choices for the partitioning element than the first element in the set will be better in practice.

Algorithm

```
Function partition ( A, lb, ub )
        A → array of elements (lb:ub)
        lb → starting index of the array A (lower bound)
        ub → ending index of the array A (upper bound)
return value → integer (index at which the array should be partitioned)
integer → down, up, temp, value
1) start
2) value = A(lb)
3) down = lb +1  and up = ub
4) while A(down) < value do
        down ++
5) while A(up) > value do
        up - -
6) if  down < up then
        a) interchange A(down) with A(up) as
                temp = A(down)
                A(down) = A(up)
                A(up)= temp
        b) goto step 4
7) A(lb) = A(up)
8) A(up)= value
9) return up (as the partition index)
```

As an example consider the list of following 8 elements.

(37, 18, 40, 82, 69, 20, 30, 56)

The initial partition element is the first element i.e. 37. The steps to keep 37 at its correct place can be shown as follows.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	down	up
37	18	40	82	69	20	30	56	3	7
37	18	30	82	69	20	40	56	4	6
37	18	30	20	69	82	40	56	5	4
20	18	30	**37**	69	82	40	56		

Using a 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 procedure partition two sets S1 and S2 are produced. All elements in S1 are less than or equal to the elements in S2. Hence S1 and S2 may be sorted independently. Each set will be sorted by reusing procedure partition. The complete recursive process which makes use of function partition can be given as follows.

```
Function quicksort( A, lb, ub)
        A → array of elements (lb:ub)
        lb → starting index of the array A (lower bound)
        ub → ending index of the array A (upper bound)
integer → p
1) start
2) if lb < ub then
        a) call partition (A, lb, ub) and store the returned value in p
        b) call quicksort (A, lb, p-1)
        c) call quicksort (A, p+1, ub)
3) return
```

Implementation

```
import java.util.*;
class Quick
{
 int x[], size;
 public Quick(int size)
 {
  Scanner src=new Scanner(System.in);
  this.size=size;
  x=new int[size];
  System.out.println("Enter "+size+" elements");
  for (int i=0; i<size; i++)
   x[i]=src.nextInt();
 }
public int partition (int lb, int ub)
{
 int value= x[lb],down=lb+1,up=ub,temp,i;
 while(true)
 {
  while(x[down]<value)
  down++;
  while (x[up]>value)
   up--;
  if(down<up)
  {
      temp=x[up];
      x[up]=x[down];
      x[down]=temp;
  }
else break;
```

```
 }
 x[lb]=x[up];
 x[up]=value;
 return up;
}
public void quickSort (int lb, int ub)
{
 int p;
 if (lb<ub)
 {
      p=partition(lb,ub);
      quickSort(lb,p-1);
      quickSort(p+1,ub);
 }
}
public void display()
{
 for (int i=0; i<size; i++)
   System.out.print(x[i]+" ");
 System.out.println();
}
public int getSize()
{
 return size;
}
}
class QuickExp
{
 public static void main(String args[])
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter number of elements");
  int n=src.nextInt();
  Quick obj=new Quick(n);
  System.out.println("The original array");
  obj.display();
  obj.quickSort(0,obj.getSize()-1);
  System.out.println("\nThe Sorted array");
  obj.display();
 }
}
```

Analysis

In the average case we assume that the position of the pivot element turns out to be the middle position. Then the complete array of n elements will get divided into 2 arrays of n/2 elements each. Further each of its will get divided which results in 4 arrays of n/4 elements each. This will continue m times where m = $\log_2$ n. So finally there would be n arrays of n/n elements each. So the total number of comparisons can be calculated as follows.

Total = n + 2*(n/2)+4(n/4) + ... + n*(n/n)
= n + n + n ... m times
= n * m
= n log n

Hence Quick Sort gives the efficiency of O(n log n) for average case. But Quick Sort behaves differently when the array is originally sorted or even reverse sorted. When the sort starts with already sorted elements, x[lb] i.e. x[0] will be compared with all the elements which results in n comparisons. As x[lb] is in its correct position, the array will get divided into two sub-arrays of 0 and n-1 elements. The process will repeat resulting in n-1 comparisons in next pass and we will have n such passes. So the total comparisons for the entire sort can be calculated as follows.

Total = n + (n-1) + (n-2) + ... + 1
= n (n-1) /2

The same procedure will be followed when the set of elements are already in the reverse order. Hence it shows that Quick Sort gives efficiency of $O(n^2)$ when the array is already sorted or reverse sorted. So the best case and worst case efficiency of quick sort is $O(n^2)$.

Randomized Quick Sort

The problem arises in quick sort when the elements are already in some order. The reason for that is the procedure for the selection of pivot element. For our convenience we select the first element always as pivot element. This degrades the performance of the sort for ordered elements. The better way to avoid this is to use the randomized version of quick sort. Here the pivot element is selected with the help of randomized function. Obviously, the randomized function if fed with the boundaries of the array as here we have lb and ub. Then the function returns any value between the bounds randomly. So for already sorted array, the chance of array getting divided as 0 and n-1 is minimized.

```
import java.util.*;
class RandomizedQuick
{
 int x[ ], size;
 public RandomizedQuick(int size)
 {
  Scanner src=new Scanner(System.in);
  this.size=size;
  x=new int[size];
  System.out.println("Enter "+size+" elements");
  for (int i=0; i<size; i++)
    x[i]=src.nextInt();
 }
 private int randomized(int lb, int ub)
 {
  Random obj=new Random();
  int z;
  do
  {
```

```
  z=obj.nextInt(ub);
 } while(z<lb);
  return z;
}
public int partition (int lb, int ub)
{
 int value,down=lb,up=ub,temp,i;
 int ran=randomized(lb,ub);
 System.out.println("\nPivot Randomized Index ="+ran);
 value=x[ran];
 while(true)
 {
  while(x[down]<value)
  down++;
  while (x[up]>value)
   up--;
  if(down<up)
  {
       temp=x[up];
       x[up]=x[down];
       x[down]=temp;
  }
 else break;
 }
 return up;
}
public void quickSort (int lb, int ub)
{
 int p;
 if (lb<ub)
 {
 display();
 System.out.println("LB = "+lb+" UB = "+ub);
  p=partition(lb,ub);
  quickSort(lb,p-1);
   quickSort(p+1,ub);
 }
}
public void display()
{
 System.out.println();
 for (int i=0; i<size; i++)
   System.out.print(x[i]+" ");
 System.out.println();
 }

 public int getSize()
 {
  return size;
 }
}
class RandomizedQuickExp
{
 public static void main(String args[])
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter number of elements");
  int n=src.nextInt();
  RandomizedQuick obj=new RandomizedQuick(n);
  System.out.println("The original array");
  obj.display();
  obj.quickSort(0,obj.getSize()-1);
  System.out.println("\nThe Sorted array");
  obj.display();
 }
}
```

Partition Algorithm for Selection Sort

The partition algorithm may also be used to obtain an efficient solution to the selection problem. In this problem, we are given n elements A(1:n) and required to determine the kth smallest element. If the partitioning element v is positioned at A(j0, then j-1 elements are less than or equal to A(j) and n-j elements are greater than or equal to A(j). Hence if k < j then kth smallest element is in A (1 : j-1); if k = j then A(j) is kth smallest element; if k > j then the kth smallest element is in A (j+1 : n). This is the select procedure which places the kth smallest element into position k and partitions the remaining elements such that A(i) <= A(k), 1<=i<=k and A(i) >= A(k), k < i <= n.

Algorithm

```
Function select (A, n, k)
        A → an array of n elements (1:n)
        n → number of elements in A
        k → the value between 1 and n, function has to find and place kth smallest element
        at position k
integer → low, high, p
1) start
2) call partition (A, lb, ub) and store the return value in p
3) if k = p then return
4) if k < p then
        high = p -1
   else low = p +1
5) goto step 2
```

Let us take one example. Suppose we want to find the sixth smallest (k =6) from the list of numbers given below.

(69, 28, 82, 17, 51, 78, 23, 41)

Initial call to partition function picks 69 as partition element and places it as follows.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	down	up
69	28	82	17	51	78	23	41	3	8
69	28	41	17	51	78	23	82	6	7
69	28	41	17	51	23	78	82	7	6
23	28	41	17	51	**69**	78	82		

So partition keeps 69 at position 6 and returns j = 6. Here k=6, hence k=j and we got the kth smallest element. So we need not proceed. This is just the coincidence that we have got kth smallest element in first call itself. Now suppose we were searching for fourth smallest element (k=4) and we have got j = 6. Then as per the algorithm,

k < j, therefore low remains same and high = j -1 =5.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	down	up
23	28	41	17	51				2	4
23	17	41	28	51				3	2
17	**23**	41	28	51					

Now it places 23 at position 2 and returns j as 2. In short it finds second smallest element. Now,

k > j, low = j +1 = 3 and high remains same.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	down	up
		41	28	51				5	4
		28	**41**	51					

Finally it places 41 at position 4 and returns j=4. Now we have found 4th smallest element. In this example the array also gets sorted. So it can be viewed as one more way of performing selection sort or selection sort using partition algorithm. But the value of k should be appropriately selected for array to get sorted. Because here if we have got k =6, then we get that in first partition only and the remaining array is not sorted at that point.

Implementation

```
import java.util.*;
class SelectionWithPartition
{
 int x[], size;
 public SelectionWithPartition(int size)
 {
  Scanner src=new Scanner(System.in);
  this.size=size;
  x=new int[size];
  System.out.println("Enter "+size+" elements");
  for (int i=0; i<size; i++)
    x[i]=src.nextInt();
 }
 private int randomized(int lb, int ub)
 {
  Random obj=new Random();
  int z;
  do
  {
    z=obj.nextInt(ub);
  }
  while(z<lb);
    return z;
 }
 public int partition (int lb, int ub)
 {
  int value,down=lb,up=ub,temp,i;
  int ran=randomized(lb,ub);
  System.out.println("\nPivot Randomized Index ="+ran);
  value=x[ran];
  while(true)
  {
   while(x[down]<value)
   down++;
   while (x[up]>value)
    up--;
   if(down<up)
   {
        temp=x[up];
        x[up]=x[down];
        x[down]=temp;
   }
  else break;
  }
  return up;
}
public void place (int lb, int ub, int k)
{
 int p;
 if (lb<ub)
 {
 display();
 System.out.println("LB = "+lb+" UB = "+ub);
      p=partition(lb,ub);
      if (k<p)
          place(lb,p-1,k);
      else if(k>p)
```

```
            place(p+1,ub,k);
         else return;
 }
}
public void selectionSort()
{
 for (int i=0;i<size-1;i++)
    place(0,size-1,i);
}

public void display()
{
 System.out.println();

 for (int i=0; i<size; i++)
   System.out.print(x[i]+" ");
 System.out.println();
}

public int getSize()
{
 return size;
}
}
class SelectionWithPartitionExp
{
 public static void main(String args[ ])
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter number of elements");
  int n=src.nextInt();
  SelectionWithPartition obj=new SelectionWithPartition(n);
  System.out.println("The original array");
  obj.display();
  obj.selectionSort();
  System.out.println("\nThe Sorted array");
  obj.display();
 }
}
```

Analysis

On each successive call to partition, either low increases by at least one or high decreases by at least one. Initially low = 1 and high = n. Hence, at most n calls to partition may be made. Thus the worst case complexity of select is at most O(n^2). This O(n^2) behavior occurs when the input A(1:n) is such that the partitioning element on the ith call is the ith smallest element and k = n. In this, low increases by one following each call to partition and high remains unchanged. Hence n calls are made for a total cost of $O(n^2)$. The average computing time of select is however O(n).

By choosing the partition element more carefully, we can obtain selection algorithm with worst case complexity O(n). In order to obtain such an algorithm, the partition element must be chosen such that at least some fraction of elements will be smaller than the partition element at least some (other) fraction of elements will be grater than the partition element. Such a selection of the partition element can be made using median of medians rule.

Radix Sort

This sort works for multiple digits numbers by using Bucket Sort for individual digits. So it keeps 10 buckets representing 0 to 9 digits. All the numbers are rearranged in those buckets using their least significant digit. Then this process is repeated for the remaining digits one by one till the most significant digit.

Algorithm

```
Function radix_sort( a, n)
        a (1: n) → array containing elements to sort
        n → number of elements
integer i, d, num
1) start
2) num = max (a, n) // to find the maximum element
3) d = $\log_{10}$ (num) +1 // to find the number of digits in the maximum number
4) i = 1
5) if (i > d) then go to step 8
6) bucket (a, n, i)
7) increment i by 1
8) return
Function bucket_sort (a, n, k)
        k → radix for which the radix sort to perform
Integer no(n, n), count (n), dig, i
1) start
2) i=0
3) if i = n then go to step 10
4) p = 10k
5) dig = remainder after dividing a[i] by p
6) place a[i] in nos[dig] at count[dig]
7) increment count[dig] by 1
8) increment i by 1
9) go to step 3
10) assign i = 0, j = 0
11) if i = 10 then go to step 16
12) assign m =0,
13) for m = 0 to count [i] do
        a) a[j] = nos[ i ][m]
        b) increment m and j by 1
14) increment i by 1
15) go to step 11
16) return
```

Implementation

```
import java.util.*;
class RadixSort
{
public static void bucket(int x[ ], int n, int k)
{
 int i,j;
 int no[][]=new int[10][20],count[]=new int[10];
 for (i=0; i<n; i++)
 {
    int dig;
    dig=x[i]/(int)(Math.pow(10,k-1))%10;
    no[dig][count[dig]++]=x[i];
 }
 for (i=0, j=0; i<10; i++)
 {
    int m=0;
    while (m<count[i])
    {
     x[j++]=no[i][m++];
    }
 }
 }

 public static void radixSort(int x[ ], int n)
 {
  int i;
  int max=x[0];
   for (i=1;i<n;i++)
    if (x[i]>max)
       max=x[i];
  int d=(int)(Math.log(max)+1);
  for (i=1; i<=d;i++)
   bucket(x,n,i);
 }

  public static void main(String args[])
  {
   Scanner src=new Scanner(System.in);
   System.out.println("Enter number of elements");
   int n=src.nextInt();
   int x[]=new int[n];
   for (int i=0; i<n;i++)
   x[i]=src.nextInt();
radixSort(x,n);
 System.out.println("\nSorted Array");
 for (int i=0; i<n; i++)
 System.out.print(x[i]+" ");
}
}
```

Analysis

The algorithms takes O(n) for each bucket sort as calculated previously and here we require 'd' bucket sorts where d is the number of digits in the largest number say 'k'. So it can be represented as d = $\log_{10}k$.

Hence total comparisons = n* d

= n* log k

Now if number of inputs is large then n nearly approaches k. Hence number of comparisons can be approximated to nlogn. So the worst case efficiency of Radix Sort is O(n logn). Like Bucket Sort, here also we need to use a huge array which degrades the space efficiency.

Sample Output

Initial Array

```
| 23 | 814 | 925 | 545 | 2547 | 21237 | 7 | 24727 | 87 | 28 |

| 7 | 814 | 23 | 925 | 24727 | 28 | 21237 | 545 | 2547 | 87 |

| 7 | 23 | 28 | 87 | 21237 | 545 | 2547 | 24727 | 814 | 925 |

| 7 | 23 | 28 | 87 | 545 | 814 | 925 | 21237 | 2547 | 24727 |

| 7 | 23 | 28 | 87 | 545 | 814 | 925 | 2547 | 21237 | 24727 |
```

Strassen's Matrix Multiplication

Let A and B be two n x n matrices. The product matrix C = AB is also an n x n matrix whose i, jth element is formed by taking elements in the ith row of A and jth column of B and multiplying to give

C (i,j) = Σ A(i, k) B(k, j), where 1 <= k <= n

for all i and j between 1 and n. To compute C(i, j) using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm (conventional) is $O(n^3)$.

The divide and conquer strategy suggests another way to compute product of two n x n matrices. For simplicity we will assume that n is a power of 2 i.e. that there exists a nonnegative integer k such that $n = 2^k$. In case, n is not a power of two then enough rows and columns may be added to both A and B so that the resulting dimensions are a power of two. Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimensions n/2 x n/2. Then the product AB can be computed by using the above formula for the product of 2 x 2 matrices, namely if AB is

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

then

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$
$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$
$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$
$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

If n=2 then the above formulas are computed using a multiplication operation for the elements of A and B. For n>2 the elements of C can be computed using matrix multiplication and addition operations applied to matrices of size n/2 x n/2. Since n is a power of 2, these matrix products can be recursively computed by the same algorithm we are using for n x n case. This algorithm continue applying itself to smaller size submatrices until n becomes suitably small (n=2) so that the product is computed directly.

In order to compute AB using the above divide and conquer method, we need to perform eight multiplications of n/2 x n/2 matrices and four additions of n/2 x n/2 matrices. Since two n/2 x n/2 matrices may be added in time cn^2 for some constant c, the overall computing time T(n) of the resulting divide and conquer algorithm id given by the recurrence relation

$$T(n) = \begin{cases} b, & n <= 2 \\ 8T(n/2) + cn^2, & n > 2 \end{cases}$$

where b and c are constants.

By solving this recurrence relation we get $T(n) = O(n^3)$. Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ vs $O(n^2)$) one may attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij}s given above using only 7 multiplications and 18 additions or subtractions. The method involves first computing the seven n/2 x n/2 matrices P, Q, R, S, T, U, V as given below. Then C_{ij}s are computed using the formulas given below. For computing these 7 n/2 x n/2 matrices we require 7 matrix multiplication and 10 matrix addition. Then finally C_{ij}s are computed using additional 8 matrix additions or subtractions.

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

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

$R = A_{11} (B_{12} - B_{22})$

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

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

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

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

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

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

The resulting recurrence relation for T(n) is

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

where a and b are constants.

Solving this recurrence relation we get,

$$T(n) = O(n^{\log 2\, 7}) \approx O(n^{2.81})$$

The bound $O(n^{2.81})$ may be further reduced if we could find a way to multiply 2 x 2 matrices using less than 7 multiplications. But theories have shown that 7 multiplications are necessary.

Implementation

```
import java.util.*;
class Strassen
{

 public static void read(int x[][], int n)
```

```
{
 Scanner src=new Scanner(System.in);
 System.out.println("Enter matrix");
  for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
    x[i][j]=src.nextInt();
}
public static void display(int x[][], int n)
{
  for (int i=0; i<n; i++)
  {  for (int j=0; j<n; j++)
      System.out.print(x[i][j]+" ");
    System.out.println();
  }
}
public static int[][] add(int x[][], int y[][], int n)
{
 int z[][]=new int[n][n];
  for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
   {
     z[i][j]=x[i][j]+y[i][j];
   }
 return z;
}

public static int[][] prod(int x[][], int y[][])
{
 int z[][]=new int[2][2];
 int P,Q,R,S,T,U,V;
 P=(x[0][0]+x[1][1])*(y[0][0]+y[1][1]);
 Q=(x[1][0]+x[1][1])*y[0][0];
 R=x[0][0]*(y[0][1]-y[1][1]);
 S=x[1][1]*(y[1][0]-y[0][0]);
 T=(x[0][0]+x[0][1])*y[1][1];
 U=(x[1][0]-x[0][0])*(y[0][0]+y[0][1]);
 V=(x[0][1]-x[1][1])*(y[1][0]+y[1][1]);
 z[0][0]=P+S-T+V;
 z[0][1]=R+T;
 z[1][0]=Q+S;
 z[1][1]=P+R-Q+U;
 return z;
}
public static void store (int p[][],int res[][],int is,int ie,int js,int je)
```

```
{
 for (int i=is,i1=0; i<ie; i++,i1++)
   for (int j=js,j1=0; j<je; j++,j1++)
     res[i][j]=p[i1][j1];
}
public static int [][] mult (int x[][], int y[][], int n)
{
 if (n==2)
   return (prod(x,y));
 int x1[][]=new int[n/2][n/2];
 int x2[][]=new int[n/2][n/2];
 int x3[][]=new int[n/2][n/2];
 int x4[][]=new int[n/2][n/2];
 int y1[][]=new int[n/2][n/2];
 int y2[][]=new int[n/2][n/2];
 int y3[][]=new int[n/2][n/2];
 int y4[][]=new int[n/2][n/2];
 for (int i=0; i<n/2; i++)
   for (int j=0; j<n/2; j++)
   {
     x1[i][j]=x[i][j];
     y1[i][j]=y[i][j];
   }
 for (int i=0,i1=0; i<n/2; i++,i1++)
   for (int j=n/2,j1=0; j<n; j++,j1++)
   {
     x2[i1][j1]=x[i][j];
     y2[i1][j1]=y[i][j];
   }
 for (int i=n/2,i1=0; i<n; i++,i1++)
   for (int j=0,j1=0; j<n/2; j++,j1++)
   {
     x3[i1][j1]=x[i][j];
     y3[i1][j1]=y[i][j];
   }
 for (int i=n/2,i1=0; i<n; i++,i1++)
   for (int j=n/2,j1=0; j<n; j++,j1++)
   {
     x4[i1][j1]=x[i][j];
     y4[i1][j1]=y[i][j];
   }
   int z[][][]=new int[8][][];
    z[0]=mult(x1,y1,n/2);
    z[1]=mult(x2,y3,n/2);
```

```
      z[2]=mult(x1,y2,n/2);
      z[3]=mult(x2,y4,n/2);
      z[4]=mult(x3,y1,n/2);
      z[5]=mult(x4,y2,n/2);
      z[6]=mult(x3,y2,n/2);
      z[7]=mult(x4,y4,n/2);
     int partial[][][]=new int[4][][],j=0;
     for (int i=0; i<n; i+=2)
     {
       partial[j]= add(z[i],z[i+1],n/2);
       j++;
     }
     int result[][]=new int[n][n];
     store (partial[0],result,0,n/2,0,n/2);
     store (partial[1],result,0,n/2,n/2,n);
     store (partial[0],result,n/2,n,0,n/2);
     store (partial[0],result,n/2,n,n/2,n);
     return result;
 }
 }
 class StrassenExp2
 {
  public static void main(String args[])
  {
   Scanner src=new Scanner(System.in);
   System.out.println("Enter the order of matrix");
   int n=src.nextInt();
   int a[][]=new int[n][n];
   int b[][]=new int[n][n];
   Strassen.read(a,n);
   Strassen.read(b,n);
   int c[][]=Strassen.mult(a,b,n);
   System.out.println("Strrasen Matrix Multiplication\nResult Matrix");
   Strassen.display(c,n);
  }
 }
```

■ ■ ■

Chapter 3 • Greedy Method

The General Method

The greedy method can be termed as the most straight forward design of all and it can be applied to variety of problems. Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called feasible solution. We are required to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution. There is usually an obvious way to determine a feasible solution, but not necessarily an optimal solution.

The greedy method suggests that one can devise an algorithm which works in stages, considering one element at a time. At each stage, a decision is made regarding whether a particular input is feasible or not to be a part of optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If adding the current input results into infeasible solution then that input is not added but permanently discarded. This is continued till either the optimal solution is achieved or all the inputs are exhausted.

Greedy algorithms are most straight forward. As the name suggests, they are very short sighted in their approach. They take decisions on the basis of information immediately available at hand without worrying about the effects of these decisions in future. Hence they are easy to implement, and efficient if they work. But daily life problems are rarely such simple to be solved with such a straight forward approach. These algorithms are used to solve optimization problems like shortest path, maximum profit etc. It works by selecting the strongest alternative at each step. It never reconsiders its decision. Hence it doesn't need any book keeping mechanism.

Characteristics (Components)

i) to construct solution, we have set of candidates currently available.

ii) as it proceeds, the candidates are classified in two sets. One contains the candidates which are considered and selected while other set contains considered and rejected.

iii) it contains one function to check whether the recent step taken has arrived till the solution or not i.e. whether the candidates selected so far are enough to construct an optimal solution or not.

iv) the other function is feasible function that checks whether the solution after adding the current candidate would be feasible or not. If the solution would not be feasible then the current candidate is not chosen but rejected.

v) one more function can be used as 'selection' function which decides the strongest candidate to be considered amongst the remaining candidates depending upon some optimization criteria.

vi) the last function can be 'objective' function which gives the value of the solution i.e. whatever we were trying to optimize. Unlike the above functions, this function does not appear in Greedy Algorithm.

The procedure can be summarized briefly as follows :

a) the solution set is empty initially.

b) each time the strongest option is selected for consideration.

c) if adding that candidate creates a feasible solution then the candidate is selected.

d) if the solution would not be feasible then the candidate is rejected.

e) the procedure is repeated till the required solution is obtained (success) or all the candidates are considered (failure).

Note that Greedy Algorithm never reconsiders the candidate that has been rejected once. Even the candidate that has been selected once will not be rejected or removed from the solution set. Hence once the candidate is there it's always there. So when a Greedy Algorithm works correctly, the first solution obtained is always optimal.

Function Greedy (C set)

```
C is the set of candidates.
S ← Ø (S is the solution set which is initially empty)
While ( C ≠ Ø and solution (S) ≠ true ) do
        x ← select (C)        // the strongest candidate is selected using select function
        C ← C \ { x }         // x is removed from C
        If ( feasible ( S U {x}) = true ) then
        S ← S U {x}           // x is added in solution set
        If ( solution (S) = true) then
                return S
return failure (solution not found).
```

Various algorithms can be of type Greedy for example a coin change problem, a knapsack problem, minimum spanning trees etc. Let us see the simplest one in the beginning i.e. coin and change problem.

Coin and Change Problem

In our daily life we have coins for 100, 50, 25, 20, 10, 5 and 1 paise. The coin and change problem is to pay the change of given amount to customer using the smallest possible number of coins.

3.39 Rs. = 3 *100 + 1 * 20 + 1* 10 + 1 * 5 + 4 * 1

This is a straight forward greedy approach. We will start with the highest feasible denomination and would continue till adding that denomination becomes infeasible. Then we would go ahead with the second highest and feasible denomination. For this problem Greedy starts with an empty set and at each step it selects the coin with highest denomination such that it does not exceed the amount to pay. The particular features of the problem can be as follows.

1) The candidates are coins of various denominations as in our example 100, 50, 25, 10, 5 and 1 units, with sufficient number of coins for each denomination that will never run out.
2) The solution function checks whether the value of the coins chosen so far is exactly t he amount to be paid.
3) The current coin is feasible if the total value of coins selected so far does not exceed the amount to be paid.
4) The selection function chooses the highest valued coin remaining in the set of candidates.
5) The objective function counts the number of coins used in the solution.

Knapsack Problem

Now let us try to apply the greedy method to solve more complex problem. This problem is the knapsack problem. We are given n objects and a knapsack. Object i has a weight Wi and profit Pi whereas the knapsack has the maximum capacity M. If a fraction Xi, 0 <= Xi <=1, of object i is placed into the knapsack then a profit of Pi Xi is earned. The objective is obtain a filling of the knapsack that maximizes the total profit earned. Since the total knapsack capacity is M, we require the total weight of all chosen objects to be at most M. Formally the problem may be stated as,

maximize Σ PiXi for 1 <= i <=n

subject to Σ WiXi <= M for 1 <= i <= n

and 0 <= Xi <= 1, 1 <= i <= n

The feasible solution is any set (X1, X2, ... Xn) satisfying both of the above constraints. An optimal solution is a feasible solution for which the value of Σ PiXi is maximum.

Consider the following instance of the knapsack problem where n=3, M=20, (P1, P2, P3) = (25, 24, 15) and (W1,W2,W3) = (18, 15, 10). Four feasible solutions are:

	(X1, X2, X3)	ΣWiXi	ΣPiXi
i)	(1/2, 1/3, 1/4)	16.5	24.25
ii)	(1, 2/15, 0)	20	28.2
iii)	(0, 2/3, 1)	20	31
iv)	(0, 1, 1/2)	20	31.5

We can see that the fourth solution yields maximum profit and we will soon see that this solution is optimal for the given problem instance.

In case the sum of all weights is <= M, then clearly Xi = 1, 1 <= i <= n is an optimal solution. But the optimal solution should fill the knapsack exactly to its capacity. This is because we can always increase by a fractional amount the contribution of some object i until the total weight is exactly M. The variety of feasible solutions whose sum is identically M can be considered. We may have following approaches.

One approach is to select the next candidate as the object with the largest profit. At last the object which doesn't fit completely can be taken in fraction. In short, we should see that at each time when the next object is included in the knapsack then there should be largest possible increase in profit. This has one drawback. It may happen that if only a fraction of the last object is included then it is possible to get a bigger increase by using a different object. For example if we are left with only two units of weight and we have the following two options (Pi=4, Wi=4) and (Pj=3, Pw=2) left in order. It is better to have complete j than to have half of i. This approach yields solution (ii) which is clearly sub-optimal.

The other approach is to select the next candidate as the object with the lowest weight. Earlier we were greedy for larger profit so now let us try to be greedy with capacity and use it up as slowly as possible. This would require us to consider objects in increasing weights Wi. This approach yields solution (iii) and it is too sub-optimal. This time though the capacity is used slowly, profits were not coming rapidly. Hence our next approach would be to achieve a balance between the rate at which the profit increases and the rate at which the capacity is used. We want more increase in profit at low increase in weight hence it would be advisable to consider the profit and weight ration. So the next candidate to be selected will be the object having maximum profit per unit of capacity used. The objects will be considered in the decreasing order of the ration Pi/Wi. Solution (iv) would be produced by this strategy. Let us put that in stepwise manner. The table arranges all the objects in the required order.

Sr.No	Pi	Wi	Pi/Wi
2	24	15	1.6
3	15	10	1.5
1	25	18	1.39

Initial P=0, W=0.

Step 1

Object 2 is next candidate. As W2 = 15 which is less than 20,

Hence Object 2 is selected completely.

W = 0 + 15 = 15

P = 0 + 24 = 24

Capacity left = 5

Step 2

Object 3 is next candidate. But W3 = 10 which is greater than 5.

Hence Object 3 is selected in fraction.

W = 15 + 10 /10 * 5 = 20

P = 24 + 15 /10 * 5 = 31.5

This is the optimal solution for the given problem. Disregarding the time to initially sort the objects, each of the three strategies outlined above requires only O(n) time.

Algorithm

```
Function Greedy_Knapsack ( P, W, M, X, n)
        P → list storing profits of n objects
        W → list storing weights of n objects
        M → maximum capacity of knapsack
        Xp → solution set storing profits
        Xw → solution set storing weights
        n → number of objects
real → rcap (remaining capacity of knapsack)
integer → i
1) start
2) Xp = Xw = empty
3) rcap = M
4) for i = 1 to n do
        a) if W(i) > rcap then return
        b) if (Xw + W(i) <= M) then
                i) Xw = Xw + W(i)
                ii) Xp = Xp + P(i)
        else    i) Xw = Xw + (M - W(i))
                ii) Xp = Xp + P(i) / W(i) * (M – W(i))
        c) if (Xw >= M ) then return
```

While greedy based algorithms using the first two measures do not guarantee optimal solutions for the knapsack problem, the third strategy always obtains an optimal solution. Let us take one example.

Pi = { 18, 5, 9, 5, 12, 4}

Wi = { 7, 2, 3, 5, 3, 1}

Here n = 6 and Max Weight = 13.

Sr. No.	Profit (Pi)	Weight (Wi)	P/W
1.	18	7	2.57
2.	5	2	2.5
3.	9	3	3
4.	5	5	1
5.	12	3	4
6.	4	1	4

So the Greedy Algorithm will proceed with the following steps.

Step 1

Object 5 is next candidate

W5 = 3 which is less than 13, hence Object 5 is selected completely

W = 0 + 3 = 3

P= 0 + 12 = 12

Capacity left = 13 – 3 =10

Step 2

Object 6 is next candidate

W6 = 1 which is less than 10, hence Object 6 is selected completely

W = 3 + 1 = 4

P= 12 + 4 = 16

Capacity left = 10 – 1 = 9

Step 3

Object 3 is next candidate

W3 = 3 which is less than 9, hence Object 3 is selected completely

W = 4 + 3 = 7

P = 16 + 9 = 25

Capacity left = 9 – 3= 6

Step 4

Object 1 is next candidate

But W1 = 7 > 6, hence Object 1 cannot be selected completely but it can be selected in fraction so that the maximum capacity can be reached

We need more W = 6 i.e. 7 * 6/7 = 6

Hence Profit = 18 * 6/7 = 15.42

W = 7 + 6 = 13

P= 25 + 15.42 = 40.42 which is the maximum profit.

Implementation

```
import java.util.*;

class KnapSack
{
 double x[][];
 int size;
 double sackCap;
 public KnapSack()
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter the number of items");
  size=src.nextInt();
  x=new double[size][3];
  System.out.println("Enter the profits and weights");
  for (int i=0; i<size; i++)
  {
   x[i][0]=src.nextDouble();
   x[i][1]=src.nextDouble();
   x[i][2]=x[i][0]/x[i][1];
  }
  System.out.println("Enter the capacity of the sack");
  sackCap=src.nextDouble();
 }
 public void sort()
 {
  for (int i=0; i<size-1; i++)
    for (int j=0; j<size-1; j++)
      if (x[j][2]<x[j+1][2])
      {
       double t[];
       t=x[j];
       x[j]=x[j+1];
       x[j+1]=t;
      }
 }

 public void display()
 {
  System.out.println("The items are");
  for (int i=0; i<size; i++)
   System.out.println(x[i][0]+" -- "+x[i][1]+" = "+x[i][2]);
  System.out.println();
 }
 public void fillKnapSack()
 {
  int i=0;
  double currentProfit=0, currentWeight=0;
  while(currentWeight<sackCap)
  {
   if (currentWeight+x[i][1]<=sackCap)
   {
    currentWeight+=x[i][1];
    currentProfit+=x[i][0];
    System.out.println("Object "+(i+1)+" : "+x[i][0]+" -- "+x[i][1]);
   }
   else
   {
    currentProfit+=x[i][0]/x[i][1]*(sackCap-currentWeight);
    currentWeight+=x[i][1]/x[i][1]*(sackCap-currentWeight);
    System.out.println("Object "+(i+1)+"(Partially) : "+x[i][0]+" -- "+x[i][1]);
   }
   i++;
  }
  System.out.println("Total Profit "+currentProfit);
 }
}
class KnapSackExp
```

```
{
 public static void main(String args[])
 {
  KnapSack obj=new KnapSack();
  obj.display();
  obj.sort();
  obj.display();
  obj.fillKnapSack();
 }
}
```

0/1 Knapsack

In the above knapsack problem we have partitioned the last object so as to fill the sack completely. But accordingly the profit is also fractioned. It may be possible to go ahead and find an object with weight less than or equal to the capacity left. It may have positive effect on the total profit. So the other method of knapsack does not allow the fractioning of objects and goes ahead till the capacity is reached or all the items are considered. This may even have reverse effect on profit that the object with manageable weight may not be found. So the fraction that would have been added is also not possible now. This type of method is known as 0/1 knapsack. It says either do not take the object (0) or take it completely(1).

```
import java.util.*;


class KnapSack01
{
 double x[][];
 int size;
 double sackCap;
 public KnapSack01()
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter the number of items");
  size=src.nextInt();
  x=new double[size][3];
  System.out.println("Enter the profits and weights");
  for (int i=0; i<size; i++)
  {
   x[i][0]=src.nextDouble();
   x[i][1]=src.nextDouble();
   x[i][2]=x[i][0]/x[i][1];
  }
  System.out.println("Enter the capacity of the sack");
  sackCap=src.nextDouble();
 }
 public void sort()
 {
  for (int i=0; i<size-1; i++)
```

```
    for (int j=0; j<size-1; j++)
      if (x[j][2]<x[j+1][2])
      {
       double t[];
       t=x[j];
       x[j]=x[j+1];
       x[j+1]=t;
      }
 }

 public void display()
 {
  System.out.println("The items are");
  for (int i=0; i<size; i++)
   System.out.println(x[i][0]+" -- "+x[i][1]+" = "+x[i][2]);
  System.out.println();
 }
 public void fillKnapSack()
 {
    double currentProfit=0, currentWeight=0;
  for (int i=0; i<size; i++)
  {
   if (currentWeight+x[i][1]<=sackCap)
   {
    currentWeight+=x[i][1];
    currentProfit+=x[i][0];
    System.out.println("Object "+(i+1)+" : "+x[i][0]+" -- "+x[i][1]);
   }
   if (currentWeight==sackCap)
     break;
  }
  System.out.println("Total Profit "+currentProfit);
  System.out.println("Total Weight "+currentWeight);
 }
}

class KnapSack01Exp
{
 public static void main(String args[])
 {
  KnapSack01 obj=new KnapSack01();
  obj.display();
  obj.sort();
  obj.display();
 obj.fillKnapSack();
}
}
```

Minimum Spanning Trees

Let G=(V,E) be an undirected connected graph. A sub graph T=(V',E') is a spanning tree of G, iff T is a tree. Spanning trees can be used to obtain an independent set of circuits equations for an electrical network. After getting a spanning tree suppose B is a set of edges not present in the spanning tree then adding one edge from B will form a electrical circuit or cycles. The cycles obtained in this way are independent. Hence the circuit equations so obtained are also independent.

Spanning tree has many other applications as well. A spanning tree is a minimal subgraph G' of G such that V(G')=V(G) and G' is connected by a minimal sub graph. Any connected graph with n vertices must have at least n – 1 edges and all connected graphs with n -1 edges are trees. If the nodes of G represents cities and the edges represents possible communication links connecting to cities. Then the minimum number of links needed to connect the n cities is n – 1. The spanning trees of G will represent all feasible choices. In any practical situation however the edges will have weights assigned to them. These weights might represent the cost of construction, the length of the link etc. Given such a weighted graph one would then wish to select for construction a set of communication link that would connect all the cities and have minimum total cost or be of minimum total length. In either case the links selected will have to form a tree (assuming all weights are positive). In case this is not so, then the selction of links contains a cycle. Removal of any one of the links on this cycle will result in a link selection of less cost connecting all cities. We are therefore interest in finding a spanning tree of G with minimum cost (the cost of a spanning tree is the sum of the cost of edges in that tree).

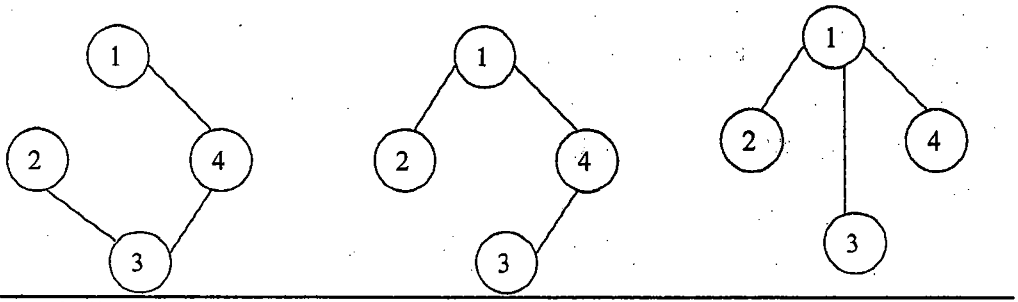
There is a possible interpretation of the optimization criteria mentioned earlier where the edges of the graph are considered in non-decreasing order of the costs. This interpretation is that the set T of edges so far selected for the spanning tree be such that it is possible to complete T into a tree. Thus T may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges T can be completed into a tree iff there are no cycles in T. This interpretation of greedy method also results in minimum cost spanning tree. This method is called as Kruskal's algorithm.

A second greedy method to obtain a minimum cost spanning tree would build this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest such criterion would be to choose an edge that result in minimum increase in the sum of the cost of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected form a tree. Thus, is A is a set of edges selected so far, then A forms a tree. The next edge (u,v) to be included in A is a minimum cost edge not in A with the property that A U {(u,v)} is also a tree. The algorithm following this way is known as Prim's algorithm.

Spanning Tree

Any tree consisting of edges of a graph and including all the vertices of the graph is known as a 'Spanning Tree'.

The figure shows a graph with four vertices and the following are three of its spanning tree.



Minimum Cost Spanning Tree

Given a connected weighted graph G, it is often desired to create a spanning tree T for G such that the sum of the weights of the tree edges in T is as small as possible. Such a tree is called a minimum cost spanning tree and represents the cheapest way of connecting all the nodes in G.

A greedy method to obtain a minimum cost spanning tree would be to build this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest criterion would be to choose an edge that results in a minimum increase in the sum of the costs of edges so far included. These are two ways to interpret this criterion.

1) In the first way the edges of the graph are considered in non-decreasing order of cost. The set T of edges so far selected for the spanning tree should be such that it is possible to complete T into a tree. Thus T may not be a tree at all stages in the algorithm. T can be completed into a tree if and only if there are no cycles in T. This method is known as Kruskal's Algortihm.

2) The set of edges so far selected should form a tree. Now if A is the set of edges selected so far, then A has to form a tree. The next edge (u,v) to be included in A is a minimum cost edge not in A with the property that A U {(u,v)} also results in a tree. This selection criterion will result in a minimum cost spanning tree. This method is known as Prim's Algorithm.

Spanning Tree Algorithms

1) Kruskal's Algorithm

Kruskal's algorithm follows the greedy approach as it always considers the edge with the minimum cost as the next candidate. It does not care about whether or not the current candidates selected form a tree or not. In this all the edges of the graph are grouped together in such a way that one may determine that the two vertices we are trying to connect say v and w are already connected by the earlier selection of edges. In case they are, then the edge (v, w) is to be discarded. If they are not, then (v, w) is to be added to the solution. One possible grouping is to place all the vertices in the same connected component of T into a set. Then two vertices v and w are connected in T (solution set) if and only if they are not in same set. Take an example

{1, 2, 3, 4, 5, 6} is the initial component set

Edge 1 :- {1, 2} → it will be connected and we get.

{1, 2}

Edge 2 :- {3, 6} → even this will be connected to give

{1, 2} . {3, 6}

Edge 3 :-{4, 6} → 4 is still there in component set so edge will be connected

{1, 2} {3, 4, 6} → as 3 and are already connected

Edge 4 :- {2, 6} → the edge is connected

{1, 2, 3, 4, 6}

Edge 5 :- {1, 4} → edge is rejected as both vertices are in solution set and belong to same subset, so they are already connected.

Edge 6 :- {3, 5} → edge is selected as vertex 5 is still in the component set.

So we finally we get {1, 2, 3, 4, 5, 6}

Summarizing we get the following rules.

1) even if at least one vertex is in the component set, the edge can be selected.

2) if both the vertices are in the selection set then they must belong to the different sub set then only the edge can be selected.

3) both the vertices are in solution set and belong to same sub set then the edge can not be selected, as they are already connected and connecting them will form a cycle.

Algorithm

```
Function Kruskal( Weight, n)
        Weight → a matrix representation of weighted graph
        n → number of vertices
e → array of structure to store edge represented with Vs, Ve and Wt.
parent → Array of integers storing the parent of each node
1) start
2) read the weight matrix and construct a list of edges in e.
3) sort the edges in e in the ascending order of their weights
4) initialize nodecount to i, k to 0 and parent array to -1
5) take the edge at e[k] and assign i=e[k].Vs, j = e[k].Ve
6) perform i = parent [i] till parent[i] > -1
7) perform j = parent [j] till parent[j] > -1
8) if i=j then goto step
9) display the edge as e[k].Vs to e[k].Ve
10) assign parent[j] = i
11) increment k by 1
12) increment nodecount by 1
13) if (nodecount < n) then goto step 5
14) return
```

Implementation

```
import java.util.*;
/* KRUSKAL'S ALGORITHM For Minimum Cost Spanning Tree */
class Edge
{
 int v1,v2,wt;
}
class KruskalsExp
{
public static void kruskals(int weight[][], int max)
{
int parent[ ]=new int[max];
Edge e[ ]=new Edge[15];
int i,j,k,size=0,nodecount;
for (i=0; i<max; i++)
parent[i]=-1;
for (i=0; i<max-1; i++)
 for (j=i+1; j<max; j++)
  if (weight[i][j]!=100)
  {
   e[size]=new Edge();
   e[size].v1=i;
   e[size].v2=j;
   e[size].wt=weight[i][j];
```

```
    size++;
   }
   for (i=0;i<size; i++)
   System.out.println((e[i].v1+1)+"--"+(e[i].v2+1)+"="+e[i].wt);
   for (i=0; i<size-1; i++)
    for (j=i+1; j<size; j++)
       if (e[i].wt>e[j].wt)
       {
         Edge temp=e[i];
         e[i]=e[j];
         e[j]=temp;
       }
   System.out.println();
   for (i=0;i<size; i++)
   System.out.println((e[i].v1+1)+"--"+(e[i].v2+1)+"="+e[i].wt);
   System.out.println("Spanning Tree\n");
   nodecount=1;
   for (k=0; nodecount<max; k++)
   {
    i=e[k].v1;
    j=e[k].v2;
    while (parent[i]>-1) i= parent[i];
    while (parent[j]>-1) j= parent[j];
    if (i!=j)
    {
     parent[j]=i;
     System.out.println((e[k].v1+1)+"--"+(e[k].v2+1)+"="+e[k].wt);
     nodecount++;
    }
   }
}
public static void main(String args[ ])
{
 Scanner src=new Scanner(System.in);
 System.out.println("Enter number of nodes");
 int n=src.nextInt();
 int visited[]=new int[n];
 int x[][]=new int[n][n];
 System.out.println("Enter the weight matrix");
 for (int i=0; i<n;i++)
   for (int j=0;j<n; j++)
     x[i][j]=src.nextInt();
 kruskals(x,n);
}
```

}

Analysis

It is clearly seen that the Kruskal's algorithm requires edges in sorted sequence. Actually it is not essential to sort all the edges so long as next edge for consideration can be determined easily. If the edges are maintained as a min heap then the next edge can be obtained in O(log e) time as G has e edges. The construction of the heap itself takes O(e) time. So the worst case complexity of Kruskal's algorithm is O(e log e). The complexity widely depends on the selection of sorting algorithm for sorting the edges.

2) Prim's Algorithm

Prim's algorithm also aims at finding the minimum cost spanning tree. But extra condition is that the edges selected so far at every step should form a tree. So it has to do lot of book keeping. Starting from a given vertex, it first computes the smallest possible distances from the current vertex to all other vertex. Then the closest vertex out of them is selected as a next current. The algorithm continues till all the nodes are connected without cycle. In this there is no extra checking require for cycles as the node once selected as current will be marked as visited and will never be considered again.

Algorithm

```
Function Prims( Weight, n, node)
        Weight → a matrix representation of weighted graph
        n → number of vertices
        node → starting node
visited → Array of integers storing record whether a particular node is visited or not
distance → Array storing the distance to reach upto a particular node
precede → Array storing the record of the node preceding the specific node
newdist → variable to store distance in intermediate calculations
smalldist → variable to store the smallest distance from the current node
1) start
2) initialize visited array to 0, distance array to infinity and precede array to -1
3) Initialize visited[node]=1, distance[node]=0 and current=node
4) if all the nodes are visited then go to step
5) smalldist = infinity
6) initialize i to 0
7) if visited[i] = 0 then
        a) newdist = weigth[current][i]
        b) if newdist < distance[i] then
                i) distance[i]=newdist
                ii) precede[i]=current
        c) if dist[i] < smalldist then
                i) smalldist = dist[i]
                ii) k =i
8) increment i by 1
9) if i<n then go to step 7
10) current = k
11) visited[current] = 1
12) go to step 4
13) return
```

Implemetation

```
import java.util.*;
/* PRIM'S ALGORITHM For Minimum Cost Spanning Tree */
class PrimsExp
{
public static boolean all(int visited[ ],int max)
{
int i;
for (i=0; i<max; i++)
if (perm[i]==0)
  return false;
return true;
}
public static void prim(int node,int weight[ ][ ],int precede[ ],int distance[ ],int max)
{
int visited[ ]=new int[max];
int current,i,k=0,smalldist,newdist;
visited[node-1]=1;
distance[node-1]=0;
current = node-1;
while (all(visited,max)==false)
{
 smalldist=100;
 for (i=0; i<max; i++)
 if (visited[i]==0)
 {
  newdist=weight[current][i];
  if (newdist<distance[i])
  {
   distance[i]=newdist;
   precede[i]=current;
  }
  if (distance[i]<smalldist)
  {
   smalldist=distance[i];
   k=i;
  }
 }
 current=k;
 visited[current]=1;
 }
}
public static void main(String args[])
{
```

```
Scanner src=new Scanner(System.in);
System.out.println("Enter number of nodes");
int n=src.nextInt();
int weight[ ][ ]=new int[n][n];
int precede[ ]=new int[n];
int distance[ ]=new int[n];
for (int i=0;i<n;i++)
  distance[i]=100;
System.out.println("Enter the weight matrix");
for (int i=0; i<n;i++)
  for (int j=0;j<n; j++)
    weight[i][j]=src.nextInt();
System.out.println("Enter the starting node");
int node=src.nextInt();
prim(node,weight,precede,distance,n);
System.out.println();
for (int i=1; i<n; i++)
System.out.println((precede[i]+1)+"--"+(i+1)+"="+distance[i]);
}
}.
```

Analysis

We have seen that the Prim's algorithm starts with a tree than includes a minimum cost edge of G. Then, edges will be added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included and the cost of (i, j) is minimum among all edges (k, l) such that vertex k is in the tree and vertex l not in the tree. The time required by procedure Prim is readily seen to be O(n^2) where n is the number of vertices of graph G.

Single Source Shortest Path

Graphs may be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges may be then assigned weights which might be either the distance between the two cities connected by the edge or the average time to drive along that section highway. A motorist wishing to drive from city A to city B would be interested in answers to the following questions.

(i) Is there a path from A to B?

(ii) If there is more than one path from A to B, which is the shortest path?

The problems defined by (i) and (ii) above are special cases of he path problems. The length of the path is now defined to be the sum of weights of the edges on that path. The starting vertex of the path will be referred as the source and the last vertex the destination. The graphs will be digraphs to allow for one way streets. In the problem we shall consider, we are given a directed graph G = (V,E), a weighting function c(e) for the edges of G and a source vertex v0. The problem is to determine the shortest paths from v0 to all the remaining vertices of G. It is assumed that all the weighs are positive.

In formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also conceive of an optimization measure. One possibility is to build the shortest path one by one. As an optimization measure we can use the sum of the lengths of all the paths so far generated. In order for this measure to be minimized, each individual path must be of minimum length. Using this optimization measure, if we have already constructed i shortest paths then the next path to be constructed should be the next shortest minimum length path. The greedy way (and also a systematic way) to generate the shortest paths from v0 to the remaining vertices would be to generate these paths in non-decreasing order of path length. First a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated and so on.

Dijkastra's Algorithm

Algortihm

```
Function dijkastras( Weight, n, node)
        Weight → a matrix representation of weighted graph
        n → number of vertices
        node → starting node
visited → Array of integers storing record whether a particular node is visited or not
distance → Array storing the distance to reach upto a particular node
precede → Array storing the record of the node preceding the specific node
newdist → variable to store distance in intermediate calculations
smalldist → variable to store the smallest distance from the current node
dc → distance for the current node
1) start
2) initialize visited array to 0, distance array to infinity and precede array to -1
3) Initialize visited[node]=1, distance[node]=0, dc =0 and current=node
4) if all the nodes are visited then go to step
5) smalldist = infinity and dc= distance[current]
6) initialize i to 0
7) if visited[i] = 0 then
        a) newdist = dc + weigth[current][i]
        b) if newdist < distance[i] then
                i) distance[i]=newdist
                ii) precede[i]=current
        c) if dist[i] < smalldist then
                i) smalldist = dist[i]
                ii) k =i
8) increment i by 1
9) if i<n then go to step 7
10) current = k
11) visited[current] = 1
12) go to step 4
13) return
```

Implementation

```
import java.util.*;
class DijkastrasExp
{
public static boolean all(int perm[],int max)
{
int i;
for (i=0; i<max; i++)
if (perm[i]==0)
```

```
  return false;
return true;
}
public static void dijkastras(int node,int weight[][],int precede[],int distance[],int max)
{
int perm[]=new int[max],dc;
int current,i,k=0,smalldist,newdist;
perm[node-1]=1;
distance[node-1]=0;
current = node-1;
while (all(perm,max)==false)
{
 smalldist=100;
 dc=distance[current];
 for (i=0; i<max; i++)
 if (perm[i]==0)
 {
  newdist=dc+weight[current][i];
  if (newdist<distance[i])
  {
   distance[i]=newdist;
   precede[i]=current;
  }
  if (distance[i]<smalldist)
   {
    smalldist=distance[i];
    k=i;
   }
 }
 current=k;
 perm[current]=1;
 }
}
 public static void main(String args[])
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter number of nodes");
  int n=src.nextInt();
  int visited[]=new int[n];
  int weight[][]=new int[n][n];
  int precede[]=new int[n];
  int distance[]=new int[n];
  for (int i=0;i<n;i++)
    distance[i]=100;
```

```
System.out.println("Enter the weight matrix");
for (int i=0; i<n;i++)
  for (int j=0;j<n; j++)
    weight[i][j]=src.nextInt();
System.out.println("Enter the starting node");
int node=src.nextInt();
dijkastras(node,weight,precede,distance,n);
System.out.println();
System.out.println("Shortest distance from node "+node);
for (int i=0; i<n; i++)
if (i!=node-1)
System.out.println("For Node "+(i+1)+"="+distance[i]);
}
}
```

Analysis

The time taken by the algorithm on a graph with n vertices is O(n^2). To see this note that the algorithm takes O(n) time as there are n vertices. For each vertex finding the closet vertex also requires O(n) time. So the total time for this procedure is O(n^2). Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in he shortest path. Hence the minimum possible time for such an algorithm would be O(e). Since cost adjacency matrices were used to represent graph, it takes O(n^2) just to determine that which edges are in G and so any shortest path algorithm using this representation must take O(n^2). So the algorithm is optimal for the given representation. If the adjacency list is used then the overall time can be brought down to O(e) but the total time required would remain O(n^2).

Job Sequencing with Deadlines

We are given a set of n jobs. Associated with job i is an integer deadline di >= 0 and a profit pi >= 0. For any job i the profit pi is earned if and only if the job is completed by its deadline. In order to complete a job one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J or Σ Pj for all j's belong to J. An optimal solution is a feasible solution with maximum profit value.

Let n = 4, (p1, p2, p3, p4) = (100, 10, 15, 27) and (d1, d2, d3, d4) = (2, 1, 2, 1). The feasible solutions and their values are as follows.

	Feasible solution	**Processing sequence**	**Value**
(i)	(1, 2)	2, 1	110
(ii)	(1, 3)	1, 3 or 3, 1	115
(iii)	(1, 4)	4, 1	127
(iv)	(2, 3)	2, 3	25
(v)	(3, 4)	4, 3	42
(vi)	(1)	1	100
(vii)	(2)	2	10
(viii)	(3)	3	3
(ix)	(4)	4	27

Solution (iii) is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order: job 4 followed by job 1. Thus the processing of job 4 begins at time zero and completed with the deadline and that of job 1 is completed at time 2.

In order to formulate a greedy algorithm to obtain an optimal solution we must formulate an optimization measure to determine how the next job will be chosen. As a first attempt we can choose the objective function Σpi where i belong to J as our optimization measure. Using this measure, the next job to include will be the one that increases Σpi the most subject to the constraint that the resulting J is a feasible solution. This requires us to consider the jobs in non-decreasing order of the profits. Let us apply this criterion to the data of the above example. We begin with J = empty and total profit = 0.

Step 1

Job 1 is the first candidate with highest profit = 100 and deadline = 2

So job 1 is added and J = {1} creates feasible solution.

Profit = 0 + 100 = 100

Step 2

Job 4 is the next candidate with profit = 27 and deadline = 1

So job 4 is added and J = { 1, 4 } creates feasible solution

Step 3

Job 3 is next candidate with profit = 15 and deadline = 2

As we already have two jobs in sequence we can't meet the deadline of job 3

So J = { 1, 3, 4} is not a feasible solution hence job 3 is rejected.

Step 4

Job 2 is next candidate with profit = 10 and deadline = 1

As we already have two jobs in sequence we can't meet the deadline of job 2

So J = { 1, 2, 4 } is not a feasible solution hence job 2 is rejected.

Finally we get J = { 1, 4} with the sequence (4, 1) and profit = 127 which is the optimal solution.

```
Function greedy_job (D, J, n)
            J → output variable which is the set of jobs to be completed by their deadlines
            D → the deadlines for the jobs
            n → total number of jobs
integer → i
1) start
2) J ← {1}
3) for i = 2 to n do
      a) if (J U {i}) can be completed by their deadlines then
            J ← J U {i}
4) return
```

Now let us see how to represent J and how to carry out the test in the algorithm to check whether the jobs selected so far can be completed by their deadlines or not. We can avoid sorting the jobs in J each time by keeping the jobs in J ordered by deadlines. J itself may be represented by a one dimensional array(1:k) such that J(r), 1<= r <=k are the jobs in J and D(J(1)) <= D(J(2)) <= ... <= D(J(k)). To test if J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that D(J(r)) <= r for 1<= r <= k+1. If job i is to be inserted at position l then only the position of jobs J(l), J(l+1), ..., J(k) is changed after the insertion. Hence, it is necessary to verify only that these jobs (and also job i) do not violate their deadlines following the insertion. We will discuss the algorithm which results from this discussion. The algorithm assumes that the jobs are already sorted in the non-decreasing order of the profits. Further it assumes n >= 1 and that the deadline D(i) of job i is at least 1. Note that no job with D(i) < 1 can ever be finished by its deadline.

Algorithm

```
Function Job_Sequencing (D, J, n, k)
        J → output variable which is the set of jobs to be completed by their deadlines
        D → the deadlines for the jobs
        n → total number of jobs
        k → number of jobs placed on output (J)
integer → i, l
1) start
2) k =1 and J(1) = 1          // job 1 is placed
3) for i = 2 to n do
        a) r = k
        b) while D(J(r)) > D(i) and D(J(r)) ≠ r do
                r = r – 1
        c) if D(J(r)) <= D(i) and D(i) > r then
                i) for l = k down to r +1 do
                        J(l+1) = J(l)
                ii) J(r+1) = i
                iii) k = k+1
4) return
```

Implementation

```
import java.util.*;
class JobSequence
{
 int x[][];
 int d[][],noOfJobs,maxDeadLine=0;
 public JobSequence()
 {
  Scanner src=new Scanner(System.in);
  System.out.println("Enter the number of jobs");
  noOfJobs=src.nextInt();
  x=new int[noOfJobs][2];
  System.out.println("Enter the profits and deadlines");
  for (int i=0; i<noOfJobs; i++)
  {
   x[i][0]=src.nextInt();
   x[i][1]=src.nextInt();
   if (x[i][1]>maxDeadLine)
     maxDeadLine=x[i][1];
  }
 d=new int[maxDeadLine+1][2];
}
public void sort()
{
 for (int i=0; i<noOfJobs-1; i++)
```

```
   for (int j=0; j<noOfJobs-1; j++)
     if (x[j][0]<x[j+1][0])
     {
      int t[];
      t=x[j];
      x[j]=x[j+1];
      x[j+1]=t;
     }
}
public int check(int z)
{
 int val=-1;
 for (int i=1; i<=z; i++)
   if (d[i][0]==0)
       val=i;
 return val;
}
public void display()
{
 System.out.println("The items are");
 for (int i=0; i<noOfJobs; i++)
  System.out.println(" Job "+i+" : "+x[i][0]+" -- "+x[i][1]);
 System.out.println();
}
public void assignJobs()
{
 for (int i=0; i<d.length; i++)
   d[i][0]=0;
 for (int i=0; i<noOfJobs; i++)
 {
   int avail=check(x[i][1]);
   if(avail!=-1)
   {
     d[avail][0]=1;
     d[avail][1]=i;
     System.out.println("Job "+i+" is assigned to slot "+avail);
   }
   else System.out.println("Job "+i+" cannot be assigned for given deadline");
 }
 int profit=0;
 System.out.println("\n\nJob in sequence of execution\n");
 for (int i=1; i<d.length; i++)
 {
  System.out.println("Job "+d[i][1]+" Profit "+x[d[i][1]][0]+" Deadline "+x[d[i][1]][1]);
```

```
   profit+=x[d[i][1]][0];
  }
  System.out.println("Total Profit "+profit);
 }
}
class JobSequenceExp
{
 public static void main(String args[])
 {
  JobSequence obj=new JobSequence();
  System.out.println("Original Input");
  obj.display();
  obj.sort();
  System.out.println("\nJob in decreasing order of profit");
  obj.display();
  obj.assignJobs();
 }
}
```

Complexity of JS Algorithm

For JS there are two possible parameters in terms of which complexity may be measured. We can use n, the number of jobs and s, the number of jobs included in solution J. The above loop is iterated at most k times each one taking O(1) time. If the condition thereafter is true then the next loop is iterated which requires O(k – r) time to insert job i. Hence the total time is O(k). This loop is iterated n – 1 times. If s is the final value of k i.e., s is the number of jobs in the final solution then the total time required can be presented as O(sn). But since s<=n, the worst case time, as the function of n. The space needed for the algorithm is O(s) which is the space needed for J.

A Faster Implementation

The computing time of JS can be reduced from O(n2) to nearly O(n) by using a different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs then we can determine processing times for each of the jobs using the rule: if job i has not been assigned a processing time then assign it to the slot [α – 1, α] where α is the largest integer r such that 1<= r <= di, and the slot [α – 1, α] is free. This rule delays the processing of the job i as much as possible. Consequently when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots in order to accommodate the new job. If for the new job being considered there is no α as defined above then it cannot be included in J. Let's take one example.

n = 5, {p1, p2, p3, p4,p5} = { 20, 15, 10, 5, 1} and {d1, d2, d3, d4, d5} = {2, 2, 1, 3, 3}. Using the above feasibility rule we have:

J	Assigned slots	Job being considered	Action
Nil	None	1	Assigned to [1, 2]
{1}	[1, 2]	2	Assign to [0, 1]
{1, 2}	[0,1], [1, 2]	3	Cannot fit; reject
{1, 2}	[0,1], [1,2]	4	Assign to [2,3]
{1, 2, 4}	[0,1], [1,2], [2, 3]	5	Cannot fit; reject

The optimal solution is {1, 2, 4}

Optimal Storage on Tapes

There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length l_i, 1<= i <= n. Clearly, all programs can be stored on tape if and only if the sum of the lengths of the programs is at most L. We shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order I = i_1, i_2, ..., i_n, the time t_j needed to retrieve program i_j is proportional to Σl_{ik} for 1 <= k <= j. If all the programs are retrieved equally often then the expected or mean retrieval time (MRT) is 1/n Σ t_j for 1<= j <=n. In the optimal storage on tape problem, we are required to find a permutation for n programs so that when they are stored on the tape in this order the MRT is minimized. Minimizing the MRT is equivalent to minimizing D(I) = Σ Σ l_{ik} where for outer summation 1<= i <=n and for inner summation 1 <= k <= j. Let n = 3 and (l_1, l_2, l_3) = (5, 10, 3). There are n! = 6 possible orderings. These orderings and their respective D values are as given below.

Ordering I	D(I)
1, 2, 3	5 + 5 + 10 + 5 + 10 + 3 = 38
1, 3, 2	5 + 5 + 3 + 5 + 3 + 10 = 31
2, 1, 3	10 + 10 + 5 + 10 + 5 + 3 = 43
2, 3, 1	10 + 10 + 3 + 10 + 3 + 5 = 41
3, 1, 2	3 + 3 + 5 + 3 + 5 + 10 = 29
3, 2, 1	3 + 3 + 10 + 3 + 10 + 5 = 34

The optimal ordering is 3, 1, 2.

The greedy approach to build the required permutation would choose the next program based upon some optimization measure. One possible measure would be the D value of the permutation constructed so far. The next program to be store on the tape would be one which minimizes increase in D. If we have already constructed the permutation i_1, i_2, ... i_r, then appending program j given the permutation i_1, i_2, ..., i_r, i_{r+1} (program j). This increase the D value by Σ l_{ik} + i_j for 1<= k <= r. Since Σ l_{ik} for 1 <= k <= r is fixed and independent of j, the increase in D is minimized if the next program chosen is the one with the least length from among the remaining program.

The greedy algorithm resulting from the above discussion is very simple and straight forward. The greedy method simply requires us to store the program in non-decreasing (increasing) order of their lengths. This ordering can be carried out in O(n log n) time using an efficient sorting algorithm like heap sort. This ordering will result in minimization of MRT.

The tape storage problem can be extended to several tapes. If there are m tapes (m>1) as T_0 to T_{m-1}, then the programs are to be distributed on these tapes. For each tape a storage permutation is to be provided. If Ij is the storage permutation for the subset of programs stored on tape j then D(Ij) is as defined earlier. The total retrieval time (TD) is Σ D(Ij) for 0<=j<=m. The objective is to store the programs in such a way as to minimize TD.

The obvious generalization of the solution for the one tape case would be to consider the programs in non-decreasing order of li's. The program currently being considered is placed on the tape which results in the minimum increase in TD. This tape will be the one with the least amount of tape used so far. If there is more than one tape with this property then the one with the smallest index can be used. If the programs are initially ordered such that l1 <= l2 <= .... <= ln then the first m programs will be assigned to tapes T_0, ..., T_{m-1} respectively. The next m programs will be assigned to tapes T_0, ..., T_{m-1} respectively. The general rule is that program i is stored on tape $T_{i \bmod m}$. On any given tape the programs are stored in non-decreasing order of their lengths. The algorithm given below presents this rule. It assumes that the programs are ordered as above. It has computing time of O(n) and does not need to know the actual program lengths.

Algorithm

```
Function Store (n, m)
        n → number of programs to be stored
        m → number of tapes
integer → i, j
```

```
1) start
2) j = 0        // j gives tape number which starts with the lowest index
3) for i = 1 to n do
        a) display ("append program ", i, " to permutation for tape ", j)
        b) j = ( j + 1) mod m
4) return
```

The resulting storage pattern by this algorithm is optimal.

Syllabus

University of Mumbai			
Class: S.E.	Branch: Computer Engineering	Semester: IV	
Subject: Analysis Of Algorithm & Design (Abbreviated as AOAD)			
Periods per Week (each 60 min)	Lecture	04	
	Practical	02	
	Tutorial	--	
		Hours	Marks
Evaluation System	Theory	03	100
	Practical and Oral	02	25
	Oral	---	--
	Term Work	---	25
	Total	05	150

Pre-requisites: Students should familiar with data structure concept. discrete structure and Programming Language such as C++ or JAVA.

Module	Contents	Hours
1	Introduction to analysis of algorithm • Design and analysis fundamentals. • Performance analysis .space and time complexity. • Growth of function – Big-Oh. Omega. theta notation. • Mathematical background for algorithm analysis. • Randomized and recursive algorithm.	05
2	Divide and Conquer .Genaral method , Binary search, finding the min and max. .Merge sort analysis. .Quick sort, performance measurement. .Randomized version of quick sort and analysis. .Partitioned algorithm selection sort, radix sort, efficiency considerations. .Strassen's matrix multiplication.	08
3	Greedy Method .General mehod. .Knapsack problem. .Minimum cost spanning tree- kruskal and primal algo, performanance analysis. .Single source shorted path . .Job sequencing with deadlines. .Optimal storage on tapes.	08
4	Dynamic Programming . The general method	07

	. Multistage graphs. all pair shortest paths. single source shortest paths .Optimal BST .0 1 knapsack .TSP. flow shop scheduling	
5	Backtracking .The general method. .8 queen problem .sum of subsets. .Graph coloring.hamltonian cycles. . Knapsack problem.	07
6	Branch and Bound .The method. LC search. .15 puzzle:An example. . Bounding and FIFO branch and bound . . LC branch and bound . . 0/1 knapsack problem. .TP efficiency considerations.	07
7	Internet algorithm .Strings and patterns matching algorithm : .Tries. .Text compression. .Text similarity testing.	06

TERM WORK

Term work should consist of graded answer papers of the test and 12 implementations using c++/java. Students are expected to calculate complexities for all methods. Each student is to appear for at least one written test during the Term. Each implementation must consist of Problem Statement, Brief Theory, complexity calculation and Conclusion.

Topics for Implementation:

1. Implementation based on divide and conquer method.
2. Implementation on greedy approach .
3. Implementation on dynamic programming .
4. Implementation of backtracking methods
5. Implementation of Branch and Bound concept
6. Implementation of internet algorithm.

Text Books:

1. Ellis horowitz,Sarataj Sahni, S. Rajsekaran."Fundamentals of computer Algorithms" University press.
2. Anany V. Levitin "Introduction to the Design and Analysis of Algorithms" Pearson Education publication, Second Edition.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms", 2nd Edition, MIT Press/McGraw Hill, 2001
4. Michael Goodrich & Roberto Tamassia, "Algorithm design foundation,analysis and internet examples", Second Edition, Wiley student Edition.

Reference Books:

1.S. Baase, S and A. Van Gelder, "Computer Algorithms: Introduction to Design and Analysis",3rd edition. Addison Wesley, 2000

2.Kenneth berman,Jerome Paul "Algorithm:sequential,parallel and distributed" Cengage Learning

3.Mark Allen Weiss, "Data Structure & Algorithm Analysis in C++", Third Edition, Pearson Education.

	. Multistage graphs. all pair shortest paths. single source shortest paths .Optimal BST .0 1 knapsack .TSP. flow shop scheduling	
5	Backtracking .The general method. .8 queen problem .sum of subsets. .Graph coloring.hamltonian cycles. . Knapsack problem.	07
6	Branch and Bound .The method. LC search. .15 puzzle:An example. . Bounding and FIFO branch and bound . . LC branch and bound . . 0/1 knapsack problem. .TP efficiency considerations.	07
7	Internet algorithm .Strings and patterns matching algorithm : .Trics. .Text compression. .Text similarity testing.	06

TERM WORK

Term work should consist of graded answer papers of the test and 12 implementations using c++/java. Students are expected to calculate complexities for all methods. Each student is to appear for at least one written test during the Term. Each implementation must consist of Problem Statement, Brief Theory, complexity calculation and Conclusion.

Topics for Implementation:

1. Implementation based on divide and conquer method.
2. Implementation on greedy approach .
3. Implementation on dynamic programming .
4. Implementation of backtracking methods
5. Implementation of Branch and Bound concept
6. Implementation of internet algorithm.

Text Books:

1. Ellis horowitz,Sarataj Sahni, S. Rajsekaran."Fundamentals of computer Algorithms" University press.
2. Anany V. Levitin "Introduction to the Design and Analysis of Algorithms" Pearson Education publication, Second Edition.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms", 2nd Edition, MIT Press/McGraw Hill, 2001
4. Michael Goodrich & Roberto Tamassia, "Algorithm design foundation,analysis and internet examples", Second Edition, Wiley student Edition.

Reference Books:

1.S. Baase, S and A. Van Gelder, "Computer Algorithms: Introduction to Design and Analysis",3rd edition. Addison Wesley, 2000

2.Kenneth berman,Jerome Paul "Algorithm:sequential,parallel and distributed" Cengage Learning

3.Mark Allen Weiss, "Data Structure & Algorithm Analysis in C++", Third Edition, Pearson Education.