GURUNANAK INSTITUTE OF TECHNOLOGY

157/F, Nilgunj Road, Panihati
Kolkata -700114
Website: www.gnit.ac.in
Email: info.gnit@jisgroup.org

Approved by A.I.C.T.E., New Delhi

Affiliated to MAKAUT, West Bengal

Online Course Ware (OCW)

Course: Design & Analysis of Algorithms

Course Level: Undergraduate

Credit: 3

Prepared by:

Ms. Srabani Kundu (CSE)

Mr. Sourish Mitra (CSE)

MODULE 01: Complexity Analysis: [3L]

The word "***Algorithm***" comes from the Persian author ***Abdullah Jafar Muhammad ibn Musa Al-khowarizmi*** in ninth century, who has given the definition of algorithm as follows:

- An Algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- **An Algorithm** is a well defined computational procedure that takes input and produces output.
- An Algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve some particular output.

Any Algorithm must satisfy the following criteria (or Properties)

1. **Input**: It generally requires finite no. of inputs.
2. **Output:** It must produce at least one output.
3. **Uniqueness**: Each instruction should be clear and unambiguous
4. **Finiteness:** It must terminate offer a finite no. of steps.

Characteristics of an algorithm

Every algorithm should have the following five characteristic features

1. Input
2. Output
3. Definiteness
4. Effectiveness
5. Finiteness

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input. In other words, viewed little more formally, an algorithm is a step by step formalization of a mapping function to map input set onto an output set.

"**Analysis of algorithm**" is a field in computer science whose overall goal is an understanding of the complexity of algorithms (in terms of time Complexity), also known as ***execution time*** & storage (or space) requirement taken by that algorithm.

Suppose **M** is an algorithm, and suppose n is the size of the input data. The time and space used by the algorithm M are the two main measures for the efficiency of **M.** The time is measured by counting the number of key operations, for example, in case of sorting and searching algorithms, the number of comparisons is the number of key operations. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.

The ***complexity*** of an algorithm M is the *function f(n)*, which give the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage

space required by an algorithm is simply a multiple of the data size n. In general the term "***complexity***" given anywhere simply refers to the running time of the algorithm. There are 3 cases, in general, to find the complexity function f(n):

1. **Best case:** The minimum value of $f(x)$ for any possible input.
2. **Worst case:** The maximum value of $f(x)$ for any possible input.
3. **Average case:** The value of $f(x)$ which is in between maximum and minimum for any possible input. Generally the Average case implies the ***expected value*** of $f(x)$.

The analysis of the average case assumes a certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The Average case also uses the concept of probability theory. Suppose N_k the numbers $N_1, N_2, \ldots\ldots, N_k$ occur with respective probabilities $P_1, P_2, \ldots\ldots, P_k$

Then the expectation or average value of E is given by $E = P_1 N_1 + P_2 N_2 + \cdots + P_k N_k$

To understand the Best, Worst and Average cases of an algorithm, consider a linear array $A[1 \ldots n]$, where the array A contains n-elements. Students may you are having some problem in understanding. Suppose you want *either* to find the location LOC of a given element (say x) in the given array A or to send some message, such as LOC=0, to indicate that x does not appear in A. Here the linear search algorithm solves this problem by comparing given x, one-by-one, with each element in A. That is, we compare with A[1], then A[2], and so on, until we find LOC such that x=A[LOC].

Algorithm: (**Linear search**)
/* **Input:** A linear list A with n elements and a searching element x
Output: Finds the location LOC of x in the array A (by returning an index)
or return LOC=0 to indicate x is not present in A. */

1. [Initialize]: Set K=1 and LOC=0.
2. Repeat step 3 and 4 *while*$(LOC == 0 \;\&\&\; K \leq n)$
3. *if*$(x == A[K])$
4. {
5. LOC=K
6. $K = K + 1$;
7. }
8. *if*$(LOC == 0)$
9. *printf*(" x is not present in the given array A);
10. *else*
11. *printf*(" x is present in the given array A at location A[LOC]);
12. Exit [end of algorithm]

Analysis of linear search algorithm

The complexity of the search algorithm is given by the number C of comparisons between x and array elements A[K].

Best case: Clearly the best case occurs when x is the first element in the array A that is $x = A[LOC]$. In this case $C(n) = 1$

Worst case: Clearly the worst case occurs when x is the last element in the array A or x is not present in given array A (to ensure this we have to search entire array A till last element). In this case, we have $C(n) = n$

Average case: Here we assume that searched element x appear array A, and it is equally likely to occur at any position in the array. Here the number of comparisons can be any of numbers $1,2,3,\ldots\ldots,n$, and each number occurs with the probability $p = \frac{1}{n}$.

then

$$C(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \cdots\ldots\ldots + n.\frac{1}{n}$$
$$= (1 + 2 + \cdots\ldots\ldots + n).\frac{1}{n}$$
$$= \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2}$$

It means the average number of comparisons needed to find the location of x is approximately equal to half the number of elements in array A. From above discussion, it may be noted that the complexity of an algorithm in the average case is much more complicated to analyze than that of worst case. Unless otherwise stated or implied, we always find and write the complexity of an

There are three basic asymptotic $(i.e.\ n \to \infty)$ notations which are used to express the running time of an algorithm in terms of function, whose domain is the set of natural numbers N={1,2,3,…..}. These are:

- O (Big- Oh): This notation is used to express Upper bound (maximum steps)
- Ω (Big- Omega): This notation is used to express Lower bound i.e. minimum (at least) steps required to solve a problem
- Θ (“Theta”) Notations: Used to express both Upper & Lower bound, also called tight bound

Asymptotic notation gives the *rate of growth*, i.e. performance, of the run time for “*sufficiently large input sizes*” $(i.e.\ n \to \infty)$ and is **not** a measure of the *particular* run time for a specific input size (which should be done empirically). O-notation is used to express the Upper bound (worst case); Ω- notation is used to express the Lower bound (Best case) and Θ-Notations is used to express both upper and lower bound (i.e. Average case) on a function.

Space Complexity

The Space Complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion. The time complexity of an algorithm is given by the no. of steps taken be the algorithm to compute the function it was written for.

Time Complexity

The time t_p, taken by a program P, is the sum of the Compile time & the Run (execution) time.

The Compile time does not depends on the instance characteristics (i.e. no. of inputs, no. of outputs, magnitude of inputs, magnitude of outputs etc.).

Thus we are concerned with the running time of a program only.

1. Algorithm X (a,b,c)

   { $return\ a + b + b * c + \frac{a + b + c}{a} + b$

   }

Here the problem instance is characterized by the specified values of a, b, and c.

2. Algorithm SUM (a, n)

   ```
   {    S:= 0
        For i = 1 to n do
        S = S + a [i];
   }    Return S;
   ```

Here the problem instance is characterized by value of n, i.e., number of elements to be summed.

This run time is denoted by t_p, we have:

$t_p(n) = C_a ADD(n) + C_s SUB(n) + C_m MUL(n) + C_d DIV(n) + \ldots$ where n → instance characteristics. C_a, C_s, C_m, C_d denotes time needed for an Addition, Subtraction, Multiplication, Division and so on.

ADD, SUB, MUL, DIV is a functions whose values are the numbers of performed when the code for P is used on an instance with characteristics n. ***Generally, the time complexity of an algorithm is given by the no. steps taken by the algorithm to complete the function it was written for.*** The number of steps is itself a function of the instance characteristics.

How to calculate time complexity of any program

The number of machine instructions which a program executes during its running time is called its *time complexity.* This number depends primarily on the size of the program's input. Time taken by a program is the sum of the compile time and the run time. In time complexity, we consider run time only. The time required by an algorithm is determined by the number of elementary operations.

The following primitive operations that are independent from the programming language are used to calculate the running time:

- Assigning a value to a variable calling a function Performing an arithmetic operation Comparing two variables
- Indexing into a array of following a pointer reference Returning from a function

The following fragment shows how to count the number of primitive operations executed by an algorithm.

```
int sum(int n)
{
    int i, sum;
1.  sum = 0;                    //add 1 to the time count
2.  for(i = 0; i < n; i++)      //add n + 1 to the time count
    {
3.    sum = sum + i * i;        //add n to the time count
    }
4.  return sum;                 //add 1 to the time count
```

Asymptotic notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mathbf{N} = \{0, 1, 2, ...\}$. Such notations are convenient for describing the worst-case running-time function $T(n)$, which is usually defined only on integer input sizes. It is sometimes convenient, however, to *abuse* asymptotic notation in a variety of ways. For example, the notation is easily extended to the domain of real numbers or, alternatively, restricted to a subset of the natural numbers. It is important, however, to understand the precise meaning of the notation so that when it is abused, it is not *misused*. This section defines the basic asymptotic notations and also introduces some common abuses.

Θ-notation

We found that the worst-case running time of insertion sort is $T(n) = \Theta(n^2)$. Let us define what this notation means. For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$\Theta(g(n)) = \{f(n)$: there exist 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 $n \ge n_0\}$.

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$. Instead, we will usually write "$f(n) = \Theta(g(n))$" to express the same notion. This abuse of equality to denote set membership may at first appears confusing, but we shall see later in this section that it has advantages.

Figure 1.1(a) gives an intuitive picture of functions $f(n)$ and $g(n)$, where we have that $f(n) = \Theta(g(n))$. For all values of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \ge n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

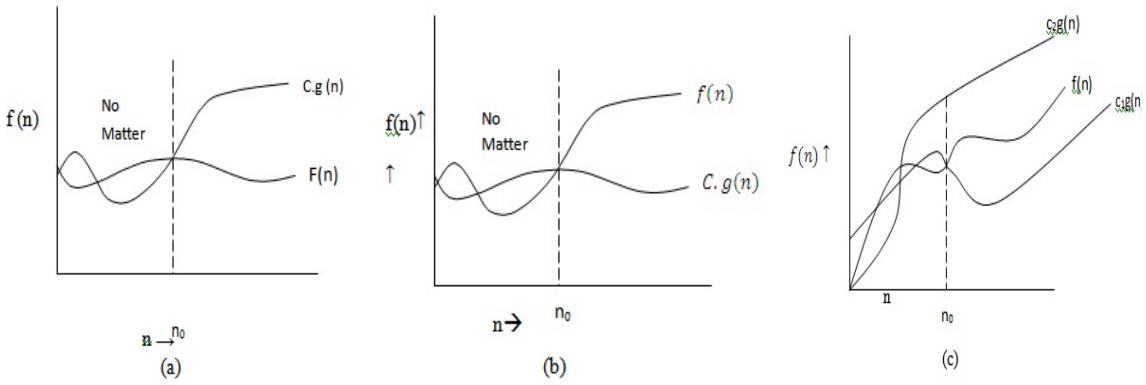


Figure 1.1: Graphic examples of the O, Ω and Θ notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. (*a*) O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. (*b*) Ω- notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$. (*c)* Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.

The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be ***asymptotically nonnegative***, that is, that $f(n)$ be nonnegative whenever n is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within Θ-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

We introduced an informal notion of Θ-notation that amounted to throwing away lower-order terms and ignoring the leading coefficient of the highest-order term. Let us briefly justify this intuition by using the formal definition to show that $1/2n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants c_1, c_2, and n_0 such that $c_1n^2 \le 1/2n^2 - 3n \le c_2n^2$ for all $n \ge n_0$. Dividing by n^2 yields $c_1 \le 1/2 - 3/n \le c_2$. The right-hand inequality can be made to hold for any value of $n \ge 1$ by choosing $c_2 \ge 1/2$. Likewise, the left- hand inequality can be made to hold for any value of $n \ge 7$ by choosing $c_1 \le 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $1/2n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that *some* choice exists. Note that these constants depend on the function $1/2n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants. We can also use the formal definition to verify that $6n^3 \ne \Theta(n^2)$. Suppose for the purpose of

contradiction that c_2 and n_0 exist such that $6n^3 \le c_2 n^2$ for all $n \ge n_0$. But then $n \le c_2/6$, which cannot possibly hold for arbitrarily large n, since c_2 is constant.

Intuitively, the lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n. A tiny fraction of the highest-order term is enough to dominate the lower-order terms. Thus, setting c_1 to a value that is slightly smaller than the coefficient of the highest-order term and setting c_2 to a value that is slightly larger permits the inequalities in the definition of Θ-notation to be satisfied. The coefficient of the highest-order term can likewise be ignored, since it only changes c_1 and c_2 by a constant factor equal to the coefficient.

_O_-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use *O*-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.We use *O*-notation to give an upper bound on a function, to within a constant factor. Figure 1(b) shows the intuition behind *O*-notation. For all values n to the right of n_0, the value of the function $f(n)$ is on or below $g(n)$.

We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than *O*-notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$. Thus, our proof that any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\Theta(n^2)$ also shows that any quadratic function is in $O(n^2)$. What may be more surprising is that any *linear* function $an + b$ is in $O(n^2)$, which is easily verified by taking $c = a + |b|$ and $n_0 = 1$.

Some readers who have seen *O*-notation before may find it strange that we should write, for example, $n = O(n^2)$. In the literature, *O*-notation is sometimes used informally to describe asymptotically tight bounds, that is, what we have defined using Θ-notation. In this book, however, when we write $f(n) = O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is. Distinguishing asymptotic upper bounds from asymptotically tight bounds has now become standard in the algorithms literature.

Using *O*-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm from Chapter 2 immediately yields an $O(n^2)$ upper bound on the worst-case running time: the cost of each iteration of the inner loop is bounded from above by $O(1)$ (constant), the indices i and j are both at most n, and the inner loop is executed at most once for each of the n^2 pairs of values for i and j.

Since *O*-notation describes an upper bound, when we use it to bound the worst-case running

time of an algorithm, we have a bound on the running time of the algorithm on every input. Thus, the $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input. The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\Theta(n^2)$ bound on the running time of insertion sort on *every* input. For example, we saw in Chapter 2 that when the input is already sorted, insertion sort runs in $\Theta(n)$ time.

Technically, it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given n, the actual running time varies, depending on the particular input of size n. When we say "the running time is $O(n^2)$," we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$. Equivalently, we mean that the worst-case running time is $O(n^2)$.

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

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

The intuition behind Ω-notation is shown in Figure 3.1(c). For all values n to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

Theorem 1.1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

As an example of the application of this theorem, our proof that $an^2 + bn + c = \Theta(n^2)$ for any constants a, b, and c, where $a > 0$, immediately implies that $an^2 + bn + c = \Omega(n^2)$ and $an^2 + bn + c = O(n^2)$. In practice, rather than using Theorem 1.1 to obtain asymptotic upper and lower bounds from asymptotically tight bounds, as we did for this example, we usually use it to prove asymptotically tight bounds from asymptotic upper and lower bounds.

Since Ω-notation describes a lower bound, when we use it to bound the best-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well. For example, the best-case running time of insertion sort is $\Omega(n)$, which implies that the running time of insertion sort is $\Omega(n)$.

The running time of insertion sort therefore falls between $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of n and a quadratic function of n. Moreover, these bounds are

asymptotically as tight as possible: for instance, the running time of insertion sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted). It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time. When we say that the *running time* (no modifier) of an algorithm is $\Omega(g(n))$, we mean that *no matter what particular input of size n is chosen for each value of n*, the running time on that input is at least a constant times $g(n)$, for sufficiently large n.

Asymptotic notation in equations and inequalities

We have already seen how asymptotic notation can be used within mathematical formulas. For example, in introducing O-notation, we wrote "$n = O(n^2)$." We might also write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. How do we interpret such formulas?

When the asymptotic notation stands alone on the right-hand side of an equation (or inequality), as in $n = O(n^2)$, we have already defined the equal sign to mean set membership: $n \in O(n^2)$. In general, however, when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n)$ is some function in the set $\Theta(n)$. In this case, $f(n) = 3n + 1$, which indeed is in $\Theta(n)$.

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, we expressed the worst-case running time of merge sort as the recurrence

$T(n) = 2T(n/2) + \Theta(n)$.

If we are interested only in the asymptotic behavior of $T(n)$, there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. For example, in the expression

$$\sum_{i=1}^{n} O(i),$$

there is only a single anonymous function (a function of i). This expression is thus *not* the same as $O(1) + O(2) + \ldots + O(n)$, which doesn't really have a clean interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation, as in $2n^2 + \Theta(n) = \Theta(n^2)$.
We interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid*. Thus, the meaning of our example is that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In

other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

A number of such relationships can be chained together, as in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

We can interpret each equation separately by the rule above. The first equation says that there is *some* function $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$ for all n. The second equation says that for *any* function $g(n) \in \Theta(n)$ (such as the $f(n)$ just mentioned), there is *some* function $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$ for all n. Note that this interpretation implies that $2n^2 + 3n + 1 = \Theta(n^2)$, which is what the chaining of equations intuitively gives us.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set $o(g(n)) = \{f(n) :$ for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in the o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 . \tag{3.1}$$

Some authors use this limit as a definition of the o-notation; the definition in this book also restricts the anonymous functions to be asymptotically nonnegative.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

For example, $n^2/2 = \omega(n)$, but $n^2/2 \ne \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

Example: $2n^2 = O(n^3)$, with c = 1 and n0 = 2. Examples of functions in $O(n^2)$:

- n^2
- $n^2 + n$
- $n^2 + 1000n$
- $1000n^2 + 1000n$
- $n/1000$
- $n^{1.99999}$
- $n^2/ \lg \lg \lg n$

A recurrence is a function is depending in terms of one or more base cases, and itself, with smaller arguments.
Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a recurrence equation which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

E.g.the worst case running time T (n) of the merge sort procedure by recurrence can be expressed as

$T(n) = \Theta(1)$; if n=1
$= 2T(n/2) + \Theta(n)$;if n>1

whose solution can be found as $T(n)=\Theta(n\log n)$

There are various techniques to solve recurrences.

Substitution method

1. Guess the solution.
2. Use induction to Þnd the constants and show that the solution works.

Recursion trees

Use to generate a guess. Then verify by substitution method.

Master method

Used for many divide-and-conquer recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n) > 0$.

The details are:

1. Substitution method:

The substitution method comprises of 3 steps

i. Guess the form of the solution

ii. Verify by induction

iii. Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name "substitution method". This method is powerful, but we must be able to guess the form of the answer in order to apply it.

e.g. recurrence equation: $T(n)=4T(n/2)+n$

step 1: guess the form of solution

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

$F(n)=4f(n/2)$

⇨ $F(2n)=4f(n)$

⇨ $F(n)=n^2$

So, $T(n)$ is order of n^2 Guess $T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k)<=ck^3$ $T(n)=4T(n/2)+n$

$<=4c(n/2)^3 +n$

$<=cn^3/2+n$

$<=cn^3-(cn^3/2-n)$

$T(n)<=cn^3$ as $(cn^3/2 -n)$ is always positive So what we assumed was true.

⇨ $T(n)=O(n^3)$

Step 3: solve for constants $Cn^3/2-n>=0$

$n>=1, c>=2$

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound

Assume, $T(k)<=ck^2$

so, we should prove that $T(n)<=cn^2$

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

$=4c(n/2)^2+n$

$=cn^2+n$

So,T(n) will never be less than cn^2. But if we will take the assumption of $T(k)=c1\ k^2-c2k$, then we can find that $T(n) = O(n^2)$

2. By iterative method:

e.g. $T(n)=2T(n/2)+n$

$=> 2[2T(n/4) + n/2]+n$

$=>2^2T(n/4)+n+n$

$=> 2^2[2T(n/8)+ n/4]+2n$

$=>2^3T(n/2^3) +3n$

After k iterations ,$T(n)=2^kT(n/2^k)+kn$---------- (1) Sub problem size is 1 after $n/2^k=1$ => $k=logn$

So,after logn iterations ,the sub-problem size will be 1. So, when k=logn is put in equation 1

$T(n)=nT(1)+nlogn$

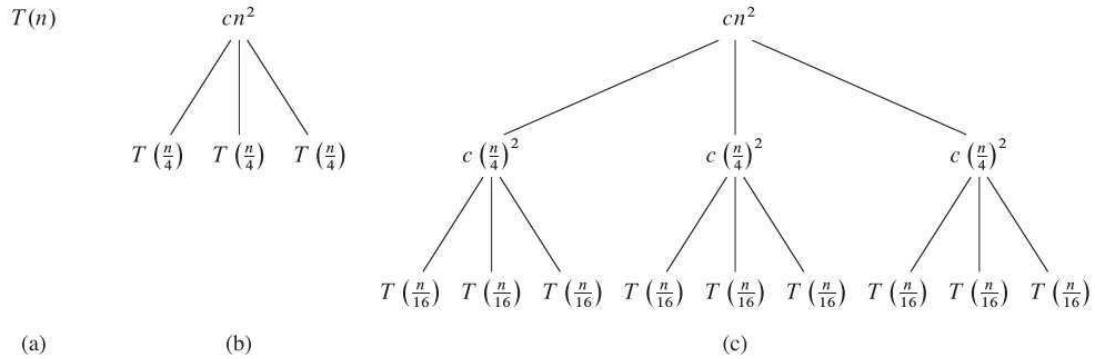
=nc+nlogn (say c=T(1))

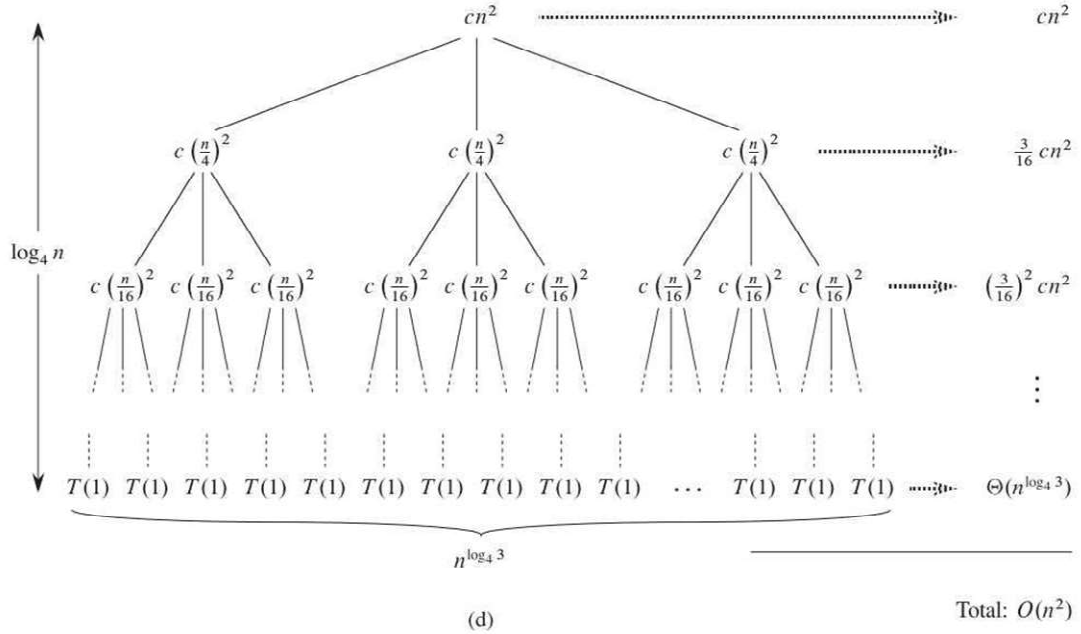
= O(nlogn)

3. By recursion tree method:

In a recursion tree ,each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations .we sum the cost within each level of the tree to obtain a set of per level cost,and then we sum all the per level cost to determine the total cost of all levels of recursion .

Constructing a recursion tree for the recurrence T (n) =3T (n/4) +cn^2





Constructing a recursion tree for the recurrence T (n)= 3T (n=4) + cn^2.. Part (a) shows T (n),

which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height log4n (it has log4n + 1 levels). Sub problem size at depth i =$n/4^i$

Sub problem size is 1 when $n/4^i=1$ => i=log4n So, no. of levels =1+ log4n

Cost of each level = (no. of nodes) x (cost of each node)

No. Of nodes at depth $i=3^i$

Cost of each node at depth $i=c\ (n/4^i)^2$

Cost of each level at depth $i=3^i\ c\ (n/4^i)^2 = (3/16)^i cn^2$ $T(n)= \sum_{i=0}^{\log_4 n} cn^2(3/16)^i$

$T(n)= \sum_{i=0}^{\log_4 n -1} cn^2(3/16)^i$ + cost of last level Cost of nodes in last level $=3^i T(1)$

$=c3^{\log_4 n}$ (at last level i=log n)

$= cn^{\log_4 3}$

$$T(n)= \sum_{i=0}^{\log_4 n -1} cn^2(3/16)^i + c\, n^{\log_4 3}$$

$$= cn^2 \sum_{i=0}^{\infty} (3/16)^i + cn^{\log_4 3}$$

$= cn^2*(16/13)+ cn^{\log\ 3}$ =>T(n)=O(n^2)

4. By master method:

The master method solves recurrences of the form

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

where a>=1 and b>1 are constants and f(n) is a asymptotically positive function .
To use the master method, we have to remember 3 cases:

1. If $f(n)=O(n^{\log_b a - \varepsilon})$ for some constants $\varepsilon >0$,then $T(n)=\Theta(n^{\log_b a})$
2. If $f(n)=\Theta(n^{\log_b a})$ then $T(n)=\Theta(n^{\log_b a}\log n)$
3. If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon>0$,and if a*f(n/b)<=c*f(n) for some constant c<1 and all sufficiently large n,then $T(n)=\Theta(f(n))$

Example:

e.g. (n)=2T(n/2)+nlogn

ans: a=2 b=2

f(n)=nlogn

using 2nd formula $f(n)=\Theta(n^{\log_2 \log^k n})$

$\Rightarrow \Theta(n^1 \log^k n)=n\log^2 n$ $\Rightarrow K=1$

$T(n)=\Theta(n^{\log_2 \log^1 n})$

$\Rightarrow \Theta(n\log^2 n)$

MODULE 01 MCQ and Short type Problem
MCQ:

1. If all c(i, j)'s and r(i, j)'s are calculated, then OBST algorithm in worst case takes one of the following time.

(a) O(n log n) (b) $O(n^3)$ (c) $O(n^2)$ (d) O(log n) (e) $O(n^4)$. Ans : $O(n^3)$

2. The asymptotic notation for defining the average time complexity is
(a)Equivalence
(b)Symmetric
(c)Reflexive
(d)Both (c) and (b) above.
Ans : Equivalence

3. The upper bound on the time complexity of the nondeterministic sorting algorithm is

(a) O(n) (b) O(n log n) (c) O(1) (d) O(log n) (e) $O(n^2)$. Ans: O(n)

4. The worst case time complexity of the nondeterministic dynamic knapsack algorithm is

(a) O(n log n) (b) O(log n) (c) $O(n^2)$ (d) O(n) (e) O(1). Ans :O(n)

5. Recursive algorithms are based on
(a) Divide and conquer approach (b) Top-down approach

(c) Bottom-up approach (d) Hierarchical approach
(e) Heuristic approach.

Ans : Bottom-up approach

6. What do you call the selected keys in the uick sort method?
(a) Outer key (b)Inner Key (c) Partition key(d) **Pivot key** (e) Recombine key. Ans : Pivot key

7. How do you determine the cost of a spanning tree?
(a) By the sum of the costs of the edges of the tree
(b) By the sum of the costs of the edges and vertices of the tree
(c) By the sum of the costs of the vertices of the tree
(d) By the sum of the costs of the edges of the graph
(e) By the sum of the costs of the edges and vertices of the graph.

Ans : By the sum of the costs of the edges of the tree

8. The time complexity of the normal quick sort, randomized quick sort algorithms in the worst case is
(a) $O(n^2)$, O(n log n) (b) $O(n^2)$, $O(n^2)$ (c) O(n log n), $O(n^2)$ (d) O(n log n), O(n log n) (e) O(n log n), $O(n^2 \log n)$.

Ans :$O(n^2)$, $O(n^2)$

9. Let there be an array of length 'N', and the selection sort algorithm is used to sort it, how many times a swap function is called to complete the execution?

(a) N log N times (b) log N times (c) N^2 times
(d) N-1 times (e) N times.

Ans :N-1 times

10. The Sorting method which is used for external sort is

(a) Bubble sort (b) Quick sort (c) Merge sort
(d) Radix sort (e) Selection sort.

Ans :Radix sort

11. In analysis of algorithm, approximate relationship between the size of the job and the amount of work required to do is expressed by using _____
(a) Central tendency (b) Differential equation (c) Order of execution (d) Order of magnitude (e) Order of Storage.

Ans: Order of execution

12. Worst case efficiency of binary search is
(a) log2 n + 1

(b) n
(c) N^2
(d) 2^n
(e) log n.

Ans : log2 n + 1

13. For defining the best time complexity, let f (n) = log n and g (n) = √n,_____
(a)f (n) ∈Ω(g(n)), but g(n) ∉ Ω (f(n)) **(b)** f (n) ∉Ω(g(n)), but g(n) ∈ Ω (f(n))
(c)f (n) ∉Ω(g(n)), and g(n) ∉ Ω (f(n)) (d) f (n) ∈Ω(g(n)), and g(n) ∈ Ω (f(n))

Ans :f (n) ∉Ω(g(n)), but g(n) ∈ Ω (f(n))

14. For analyzing an algorithm, which is better computing time?

(a) O (100 Log N) (b) O (N) (c) O (2^N) (d) O (N logN) (e) O (N^2).
Ans :O (100 Log N)

15. Let f, t: N→R ≥ 0, and t (n) ∈O (f (n)) iff t(n)≤ c.f (n) where cis positive real constant andn≥ no, then no is _____
(a) Upper bound (b)Lower bound (c) Duality value
(d) Threshold value (e) Maximum value.

Ans : Lower bound

SHORT TYPE PROBLEM:

1.Write all the 3 cases of Master method to solve a recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

2. Use Mater Theorem to give the tight asymptotic bounds of the following recurrences:

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

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

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

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

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

f. $T(n) = 8T\left(\frac{n}{2}\right) + 3n^2$

3. Write a condition when Master method fails to solve a recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

4. Can Master Theorem be applied to the recurrence of $T(n) = 4T\left(\frac{n}{2}\right) + n^2 logn$? why and why not? Give an asymptotic upper bound of the recurrence?

5. Write a recurrence relation for the following

recursive functions: a)

```
Fast_Power(x, n)
{   (n == 0)
    return 1;
  elseif (n == 1)
    return x;
  elseif ((n%2) == 0)      //if n is even
    return Fast_power(x, n/2) * Fast_power(x, n/2);
}
else
  return x * Fast_power(x, n/2) * Fast_power(x, n/2);
```

b)

```
Fibnacci(n)
{   (n == 0)
    return 0;
  if (n == 1)
    return 1;
  else
    return fibnacci(n − 1) + fibnacci(n − 2);
}
```

6. Solve the following recurrence using Iteration Method:

a) $T(n) = 3T\left(\frac{n}{2}\right) + n$

b) Recurrence obtained in .1 a) part

c) Recurrence obtained in .1 b) part

7. Solve the following recurrence Using Recursion tree method

a. $T(n) = 4T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$

b.

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

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

MODULE 2: Algorithm Design Techniques:[12]

Divide and Conquer

Divide and Conquer Algorithm

- In this approach ,we solve a problem recursively by applying 3 steps

1. **DIVIDE**-break the problem into several sub problems of smaller size.
2. **CONQUER**-solve the problem recursively.
3. **COMBINE**-combine these solutions to create a solution to the original problem. Control

abstraction for divide and conquer algorithm

```
Algorithm D and C (P)
{
if small(P)
then return S(P)
```

else

{ divide P into smaller instances P_1 ,P_2 .....P_k Apply D and C to each sub problem

Return combine (D and C(P_1)+ D and C(P_2)+.....+D and C(P_k))
}

Rurrence relation is expressed as
T(n)= Θ(1),if n<=C

=aT(n/b) + D(n)+ C(n) ,otherwise

then n=input size a=no. Of sub-problems n/b= input size of the sub-problems

Merge sort

It is one of the well-known divide-and-conquer algorithms. This is a simple and very efficient algorithm for sorting a list of numbers.

We are given a sequence of n numbers which we will assume is stored in an array A [1...n]. The objective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

- **Divide: Split A down the middle into two sub-sequences, each of size roughly n/2 . Conquer: Sort each subsequence (by calling MergeSort recursively on each).**
- **Combine: Merge the two sorted sub-sequences into a single sorted list.**

The dividing process ends when we have split the sub-sequences down to a single item. A sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage, which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The “divide” phase is shown on the left. It works top-down splitting up the list into smaller sublists. The “conquer and combine” phases are shown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.

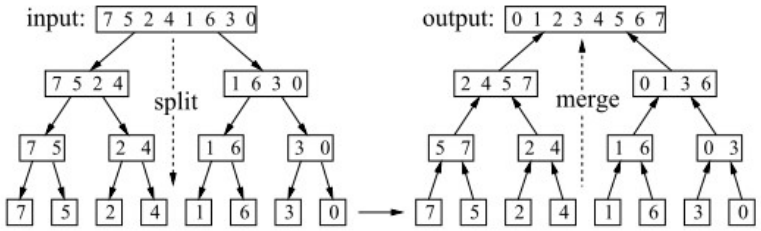


Figure 2.1 Merge Sort

Designing the Merge Sort algorithm top-down. We’ll assume that the procedure that merges two sorted list is available to us. We’ll implement it later. Because the algorithm is called recursively on sub lists, in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the subarray that we are to sort. The call MergeSort(A, p, r) will sort the sub-arrayA [p..r] and return the sorted result in the same subarray.

Here is the overview. If r = p, then this means that there is only one element to sort, and we may return immediately. Otherwise (if p < r) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely q = (p + r) / 2 (rounded down to then earest integer). Then we split the array into subarrays A [p..q] and A [q + 1 ..r] . Call Merge Sort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) which merges these two subarrays into a single sorted array.

```
MergeSort(array A, int p, int r) {
if  (p < r) {      // we have at least 2 items  q = (p + r)/2
MergeSort(A, p, q) // sort A[p..q]
MergeSort(A, q+1, r) // sort A[q+1..r]
```

```
Merge(A, p, q, r)        // merge everything together
}                                                                    }
```

Merging: All that is left is to describe the procedure that merges two sorted lists. Merge(A, p, q, r)assumes that the left sub array, A [p..q] , and the right sub array, A [q + 1 ..r] , have already been sorted. We merge these two sub arrays by copying the elements to a temporary working array called B. For convenience, we will assume that the array B has the same index range A, that is, B [p..r] . We have to indices i and j, that point to the current elements of each sub array. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A.

```
Merge(array A, int p, int q, int r) {          // merges A[p..q] with A[q+1..r]
array B[p..r]
i = k = p                                      //initialize pointers
j = q+1
while (i <= q and j <= r) {                    // while both subarrays are nonempty
if (A[i] <= A[j]) B[k++] = A[i++]              // copy from left subarray
else B[k++] = A[j++]                           // copy from right subarray
}
while (i <= q) B[k++] = A[i++]                 // copy any leftover to B
while (j <= r) B[k++] = A[j++]
for i = p to r do A[i] = B[i]          // copy B back to A                    }
```

Analysis: What remains is to analyze the running time of MergeSort. First let us consider the running time of the procedure Merge(A, p, q, r). Let n = r − p + 1 denote the total length of both the left and right sub arrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. Thus the running time to Merge n items is $\Theta (n)$. Let us write this without the asymptotic notation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are

strictly smaller than n. Finally, a recurrence has some basis values (e.g. for $n = 1$), which are defined explicitly.

Let's see how to apply this to Merge Sort. Let $T(n)$ denote the worst case running time of Merge Sort on an array of length n. For concreteness we could count whatever we like: number of lines of pseudo code, number of comparisons, number of array accesses, since these will only differ by a constant factor. Since all of the real work is done in the Merge procedure, we will count the total time spent in the Merge procedure.

First observe that if we call Merge Sort with a list containing a single element, then the running time is a constant. Since we are ignoring constant factors, we can just write $T(n) = 1$. When we call Merge Sort with a list of length $n > 1$, e.g. Merge(A, p, r), where $r - p + 1 = n$, the algorithm first computes $q = (p + r) / 2$. The sub array $A[p..q]$, which contains $q - p + 1$ elements. You can verify that is of size $n/2$. Thus the remaining sub array $A[q + 1..r]$ has $n/2$ elements in it. How long does it take to sort the left sub array? We do not know this, but because $n/2 < n$ for $n > 1$, we can express this as $T(n/2)$. Similarly, we can express the time that it takes to sort the right sub array as $T(n/2)$.

Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

$T(n) = 1$ if $n = 1$,

$2T(n/2) + n$ otherwise.

Solving the above recurrence we can see that merge sort has a time complexity of $\Theta(n \log n)$.

QUICKSORT

- Worst-case running time: $O(n2)$.
- Expected running time: $O(n \lg n)$.
- ***Sorts in place.***

Description of quicksort

Quicksort is based on the three-step process of divide-and-conquer.

- To sort the subarray$A[p \ldots r]$:

Divide: Partition $A[p \ldots r]$, into two (possibly empty) subarrays$A[p \ldots q-1]$ and $A[q+1 \ldots r]$, such that each element in the Þrstsubarray$A[p \ldots q-1]$ is $\leq A[q]$ and **A[q] *is* ≤ *each element in the second subarray* A[q + *1* . . r *]*.**

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

QUICKSORT *(A, p, r)*

if $p < r$

then $q \leftarrow$PARTITION*(A, p, r)*

QUICKSORT *(A, p, q − 1)* QUICKSORT *(A, q* + 1*, r)*

Initial call is QUICKSORT *(A,* 1*, n)*

Partitioning

Partition subarray$A$ $[p \ldots r]$ by the following procedure:

PARTITION *(A, p, r)*

$x \leftarrow A[r]$

$i \leftarrow p-1$

for $j \leftarrow p$ **to** $r-1$

do if $A[j] \leq x$

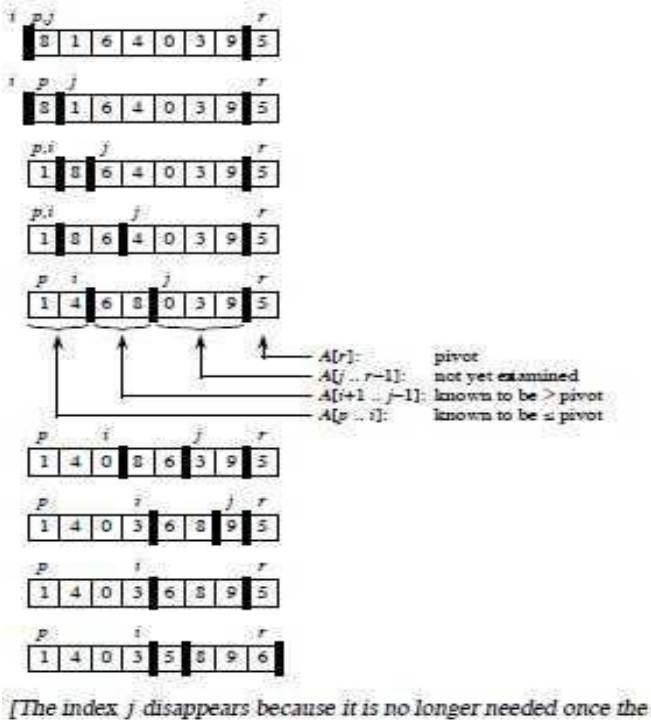
then $i \leftarrow i + 1$

exchange$A[i] \leftrightarrow A[j]$

exchange$A[i+1] \leftrightarrow A[r]$

return*i* + 1

- ***PARTITION always selects the last element* A*[r] in the subarray*A*[p . . r] as the*** pivot ***the element around which to partition.***
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:



[The index j disappears because it is no longer needed once the for loop is exited.]

Figure 2.2 Quick sort

Performance of Quick sort

The running time of Quick sort depends on the partitioning of the sub arrays:

- If the sub arrays are balanced, then Quick sort can run as fast as merge sort.
- If they are unbalanced, then Quick sort can run as slowly as insertion sort.

Worst case

- Occurs when the sub arrays are completely unbalanced.
- Have 0 elements in one sub array and $n - 1$ elements in the other sub array.

• Get the recurrence

$T(n) = T(n-1) + T(0) + \Theta(n)$

$= T(n-1) + \Theta(n)$

$= O(n2)$.

• ***Same running time as insertion sort.***

• In fact, the worst-case running time occurs when Quick sort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Best case

• Occurs when the sub arrays are completely balanced every time.

• Each sub array has $\leq n/2$ elements.

• Get the recurrence

$T(n) = 2T(n/2) + \Theta(n) = O(n \lg n)$.

Balanced partitioning

• ***QuickPort's average running time is much closer to the best case than to the worst case.***

• Imagine that PARTITION always produces a 9-to-1 split.

• Get the recurrence

$T(n) \leq T(9n/10) + T(n/10) + \_(n) = O(n \lg n)$.

• ***Intuition: look at the recursion tree.***

• It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$.

• ***Except that here the constants are different; we get log10*** **n** ***full levels and log10/9*** **n** ***levels that are nonempty.***

• As long as it's a constant, the base of the log doesn't matter in asymptotic notation.

• Any split of constant proportionality will yield a recursion tree of depth $O(\lg n)$.

<u>HEAPSORT</u>

- In place algorithm
- Running Time: O(n log n)
- Complete Binary Tree

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is

completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

The root of the tree is A[1], and given the index *i* of a node, we can easily compute the indices of its parent, left child, and right child:

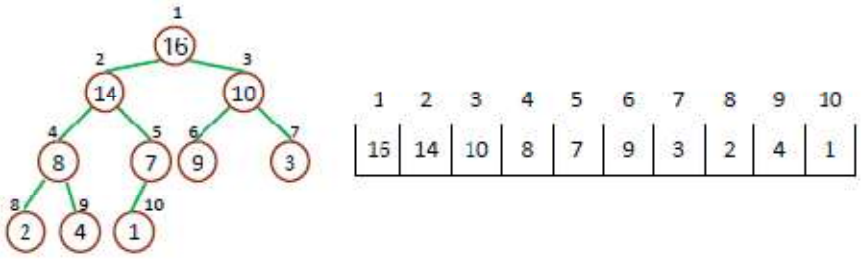


Figure 2.3 (a) Max Heap (b) heap data structure

- PARENT (i) => ***return*** [i / 2]
- LEFT (i) => ***return*** 2*i*
- RIGHT (i) => ***return*** 2i+ 1

On most computers, the LEFT procedure can compute 2i in one instruction by simply shifting the binary representation of i left by one bit position.

Similarly, the RIGHT procedure can quickly compute 2i + 1 by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit.The PARENT procedure can compute *[i/2]* by shifting i right one bit position. Good implementations of heapsort often implement these procedures as "macros" or "inline" procedures.

There are two kinds of binary heaps: **max-heaps** and **min-heaps**.

- ***In a*** max-heap,***the*** max-heap property ***is that for every node*** **i** ***other than the root,*** *A[PARENT(i)] >= A[i]* ,***that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.***
- A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node i other than the root, **A[PARENT(i)<=A[i] ,**

The smallest element in a min-heap is at the root.

✓ The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf and

- ✓ The height of the heap is the height of its root.
- ✓ Height of a heap of n elements which is based on a complete binary tree is **O(log n).**

Maintaining the heap property

MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

MAX-HEAPIFY(A,i)

1. l LEFT(i)
2. r RIGHT(i)
3. if A[l] > A[i]
4. largest l
5. if A[r] > A[largest]
6. Largest r
7. if largest != i
8. Then exchange A[i] A[largest]
9. MAX-HEAPIFY(A,largest)

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in *largest.* If A[i] is largest, then the sub tree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max-heap property. The node indexed by *largest,* however, now has the original value A[i], and thus the sub tree rooted at *largest* might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that sub tree.

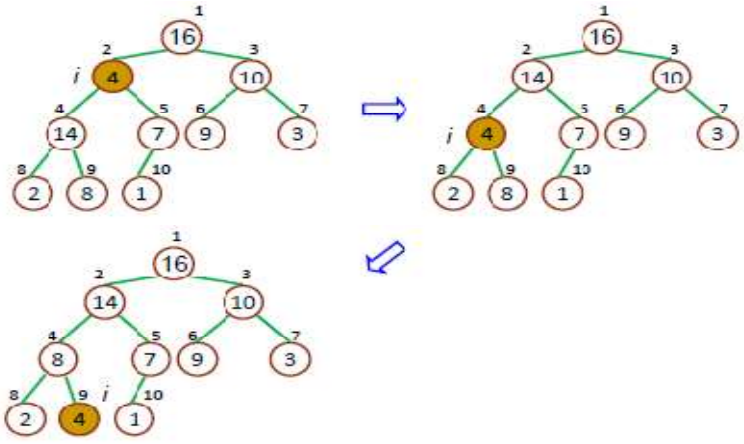


Figure 2.4 The action of MAX-HEAPIFY (A, 2), where *heap-size* = 10. **(a)** The initial configuration, with A [2] at node i = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging A [2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY (A,4) Now has i = 4. After swapping A[4] with A[9], as shown in **(c),** node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.

The running time of MAX-HEAPIFY by the recurrence can be described as T (n) < = T (2n/3)

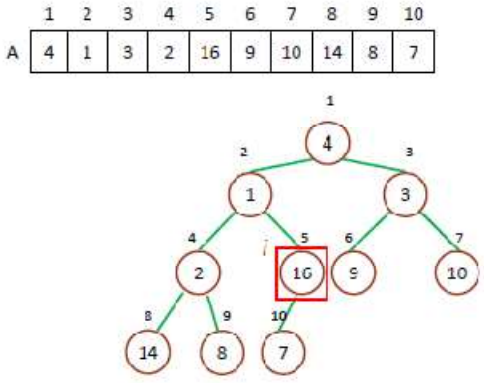
+ O (1)

The solution to this recurrence is T(n)=O(log n)

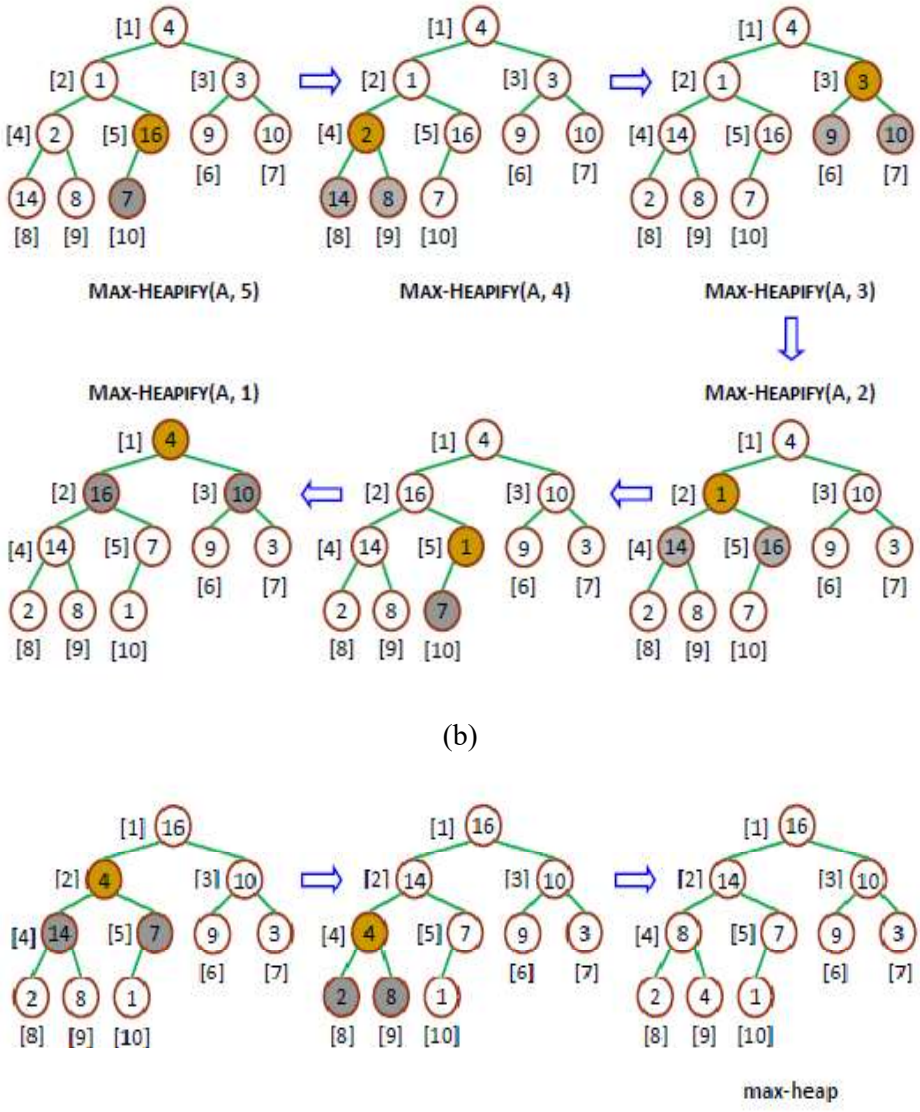
Building a heap

Build-Max-Heap(A)

1. for i [n/2] to 1
2. do MAX-HEAPIFY(A,i)



(a)



(b)

(c)

Figure 2.5 Max Heap

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height [log n] and at most $[n/2^{h+1}]$ nodes of any height h.

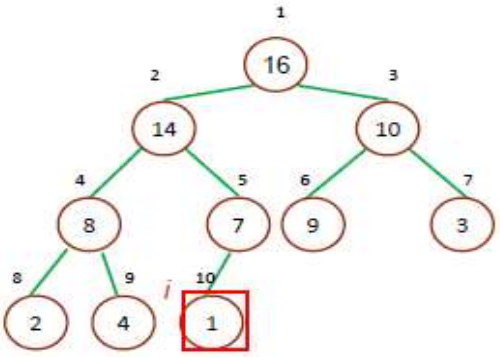
The total cost of BUILD-MAX-HEAP as being bounded is T(n)=O(n)

The HEAPSORT Algorithm

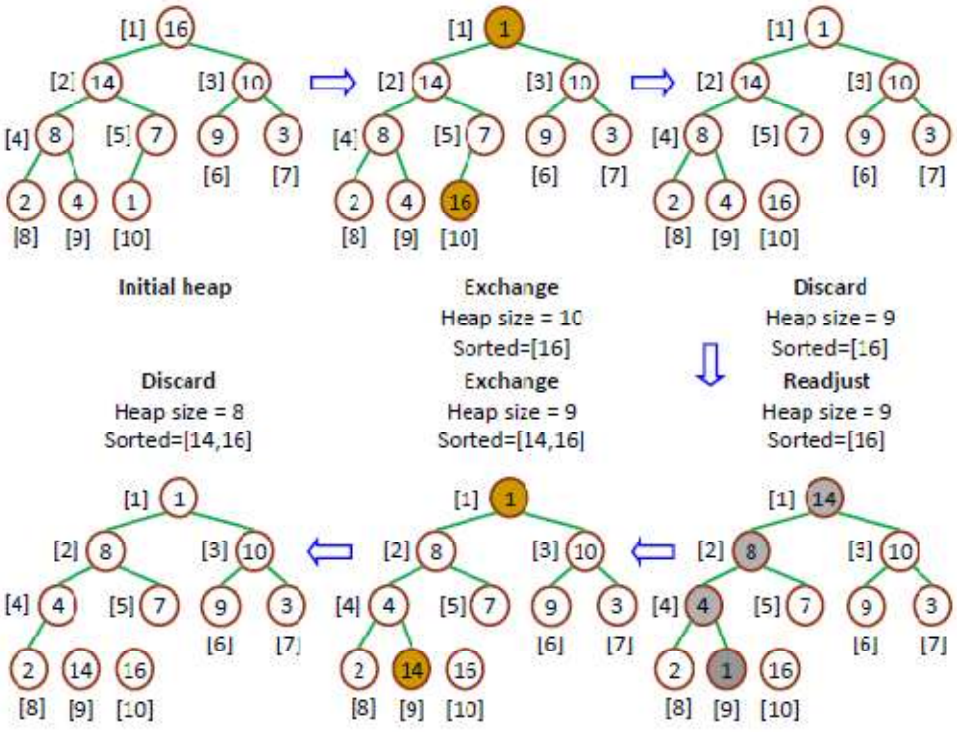
HEAPSORT(A)

1. BUILD MAX-HEAP(A)
2. for i=n to 2

3. exchange A[1] with A[i]
4. MAX-HEAPIFY(A,1)



	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16



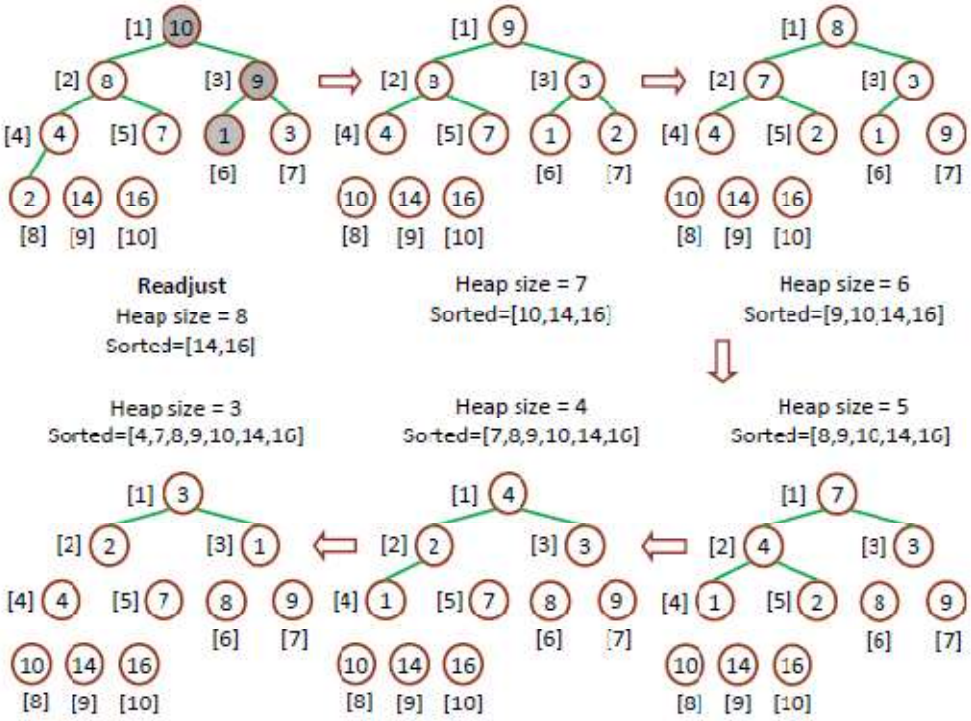


Figure 2.6 Heap Sort

TheHEAPSORT procedure takes time O(n log n), since the call to BUILD-MAX- HEAP takes time O(n) and each of the n - 1 calls to MAX-HEAPIFY takes time O(log n).

MODULE 2: Algorithm Design Techniques:[12]
Dynamic Programming

The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. The DP in closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. The DP differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these sub- problems.
The steps of Dynamic Programming technique are:

- **Dividing the problem into sub-problems**: The main problem is divided into smaller sub- problems. The solution of the main problem is expressed in terms of the solution for the smaller sub-problems.

- **Storing the sub solutions in a table**: The solution for each sub-problem is stored in a table so that it can be referred many times whenever required.

- **Bottom-up computation**: The DP technique starts with the smallest problem instance

The strategy can be used when the process of obtaining a solution of a problem can be viewed as a sequence of decisions. The problems of this type can be solved by taking an optimal sequence of decisions. An optimal sequence of decisions is found by taking one decision at a time and never making an erroneous decision. In Dynamic Programming, an optimal sequence of decisions is arrived at by using the principle of optimality. The principle of optimality states that whatever be the initial state and decision, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting form the first decision.
A fundamental difference between the greedy strategy and dynamic programming is that in the greedy strategy only one decision sequence is generated, wherever in the dynamic programming, a number of them may be generated. Dynamic programming technique guarantees the optimal solution for a problem whereas greedy method never gives such guarantee.

Let, we have three matrices A1, A2 and A3, with order (10 x 100), (100 x 5) and (5 x 50) respectively.

Then the three matrices can be multiplied in two ways.

(i) First, multiplying A2 and A3, then multiplying A1 with the resultant matrix i.e. A1(A2 A3).
(ii) First, multiplying A1 and A2, and then multiplying the resultant matrix with A3 i.e. (A1A2) A3.

The number of scalar multiplications required in case 1 is 100 * 5 * 50 + 10 * 100 * 50 = 25000 + 50,000 = 75,000 and the number of scalar multiplications required in case 2 is 10 * 100 * 5 + 10 * 5 * 50 = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesize the expression in every possible fashion and count each time how many scalar multiplications are required. Thus the matrix chain multiplication problem can be stated as "find the optimal parenthesisation of a chain of matrices to be multiplied such that the number of scalar multiplications is minimized".

Dynamic Programming Approach for Matrix Chain Multiplication

Let us consider a chain of n matrices $A1, A2 \ldots\ldots\ldots A_n$, where the matrix Ai has dimensions $P[i\text{-}1]$ x $P[i]$. Let the parenthesisation at k results two sub chains $A1 \ldots\ldots A_k$ and $A_k{+}1 \ldots\ldots A_n$. These two sub chains must each be optimal for $A1 \ldots\ldots A_n$ to be optimal. The cost of matrix chain $(A1 \ldots A_n)$ is calculated as $cost(A1 \ldots\ldots A_k) + cost(A_k{+}1 \ldots\ldots A_n)$ + cost of multiplying two resultant matrices together i.e.

***cost*(*A1……An*)= *cost*(*A1……Ak*) + *cost*(*Ak*+1…...*An*) + cost of multiplying two resultant matrices together.**

Here, the cost represents the number of scalar multiplications. The sub chain $(A1 \ldots A_k)$ has a dimension $P[0]$ x $P[k]$ and the sub chain $(A_k{+}1 \ldots\ldots A_n)$ has a dimension $P[k]$ x $P[n]$. The number of scalar multiplications required to multiply two resultant matrices is $P[0]$ x $P[k]$ x $P[n]$.

Let $m[i, j]$ be the minimum number of scalar multiplications required to multiply the

matrix chain $(A_i \ldots\ldots\ldots A_j)$. Then

(i) $m[i, j] = 0$ if $i = j$

(ii) $m[i, j]$ = minimum number of scalar multiplications required to multiply $(A_i \ldots A_k)$ + minimum number of scalar multiplications required to multiply $(A_{k+1} \ldots A_n)$ + cost of multiplying two resultant matrices i.e.

$$m[i, j] = m[i, k] + m[k, j] + P[i-1] \times P[k] \times P[j]$$

Therefore, the minimum number of scalar multiplications required to multiply n matrices $A_1 A_2 \ldots\ldots A_n$ is

$$m[1, n] = \min_{1 \le k < n} \{m[1, k] + m[k, n] + P[0] \times P[k] \times P[n]\}$$

The dynamic programming approach for matrix chain multiplication is presented in Algorithm

Algorithm MATRIX-CHAIN-MULTIPLICATION (P)

// P is an array of length $n+1$ i.e. from $P[0]$ to $P[n]$. It is assumed that the matrix A_i has the dimension $P[i-1] \times P[i]$.

```
{
    for(i = 1; i<=n;
        i++) m[i, i] =
        0;
    for(l = 2; l<=n; l++){
        for(i = 1; i<=n-(l-1);
            i++){ j = i + (l-
            1);
           m[i, j] = ∞;
        for(k = i; k<=j-1; k++)
              q = m[i, k] + m[k+1, j] + P[i-1] P[k] P[j] ;
              if (q<m [i, j]){
                 m[i, j] = q;
```

```
                    s[i, j] = k;
                }
            }
        }
    }
    return m and s.
}
```

.

MODULE 2: Algorithm Design Techniques: [12]
Dynamic Programming

Now let us discuss the procedure and pseudo code of the matrix chain multiplication. Suppose, we are given the number of matrices in the chain is n i.e. $A1, A2 \ldots\ldots A_n$ and the dimension of matrix A_i is $P[i-1] \times P[i]$. The input to the matrix-chain-order algorithm is a sequence $P[n+1] = \{P[0], P[1], \ldots\ldots P[n]\}$. The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \ldots\ldots n$ in lines 2-3. Then, the algorithm computes $m[i, j]$ for $j - i = 1$ in the first step to the calculation of $m[i, j]$ for $j - i = n - 1$ in the last step. In lines 3 – 11, the value of $m[i, j]$ is calculated for $j - i = 1$ to $j - i = n - 1$ recursively. At each step of the calculation of $m[i, j]$, a calculation on $m[i, k]$ and $m[k+1, j]$ for $i \le k < j$, are required, which are already calculated in the previous steps.

To find the optimal placement of parenthesis for matrix chain multiplication $A_i, A_{i+1}, \ldots A_j$, we should test the value of $i \le k < j$ for which $m[i, j]$ is minimum. Then the matrix chain can be divided from $(A1 \ldots\ldots A_k)$ and $(A_k+1 \ldots\ldots A_j)$.

Let us consider matrices A1,A2……A5 to illustrate MATRIX-CHAIN-MULTIPLICATION algorithm. The matrix chain order $P = \{P_0, P_1, P_2, P_3, P_4, P_5\} = \{5, 10, 3, 12, 5, 50\}$. The objective is to find the minimum number of scalar multiplications required to multiply the 5 matrices and also find the optimal sequence of multiplications.

The solution can be obtained by using a bottom up approach that means first we should calculate m_{ii} for $1 \le i \le 5$. Then m_{ij} is calculated for $j - i = 1$ to $j - i = 4$. The value of m_{ii} for $1 \le i \le 5$ can be filled as 0 that means the elements in the first row can be assigned 0. Then

For $j - i = 1$

$m_{12} = P_0\ P_1\ P_2 = 5 \times 10 \times 3 = 150$

$m_{23} = P_1\ P_2\ P_3 = 10 \times 3 \times 12 = 360$

$m_{34} = P_2\ P_3\ P_4 = 3 \times 12 \times 5 = 180$

$m_{45} = P_3\ P_4\ P_5 = 12 \times 5 \times 50 = 3000$

For $j - i = 2$

$m_{13} = \min \{m_{11} + m_{23} + P_0\ P_1\ P_3, m_{12} + m_{33} + P_0\ P_2\ P_3\}$

$= \min \{0 + 360 + 5 * 10 * 12, 150 + 0 + 5*3*12\}$

$= \min \{360 + 600, 150 + 180\} = \min \{960, 330\}$

$= 330$ $m_{24} = \min \{m_{22} + m_{34} + P_1\ P_2\ P_4, m_{23} +$

$m_{44} + P_1\ P_3\ P_4\}$

$= \min \{0 + 180 + 10*3*5, 360 + 0 + 10*12*5\}$

$= \min \{180 + 150, 360 + 600\} = \min \{330, 960\}$

$= 330$ $m_{35} = \min \{m_{33} + m_{45} + P_2\ P_3\ P_5, m_{34} +$

$m_{55} + P_2\ P_4\ P_5\}$

$= \min \{0 + 3000 + 3*12*50, 180 + 0 + 3*5*50\}$

$= \min \{3000 + 1800 + 180 + 750\} = \min \{4800, 930\} = 930$

For $j - i = 3$

$m_{14} = \min \{m_{11} + m_{24} + P_0\ P_1\ P_4, m_{12} + m_{34} + P_0\ P_2\ P_4, m_{13} + m_{44} + P_0\ P_3\ P_4\}$

$= \min \{0 + 330 + 5*10*5, 150 + 180 + 5*3*5, 330 + 0 + 5*12*5\}$

$= \min \{330 + 250, 150 + 180 + 75, 330 + 300\}$

$= \min \{580, 405, 630\} = 405$

$m_{25} = \min \{m_{22} + m_{35} + P_1\ P_2\ P_5, m_{23} + m_{45} + P_1\ P_3\ P_5, m_{24} + m_{55} + P_1\ P_4\ P_5\}$

$= \min \{0 + 930 + 10*3*50, 360 + 3000 + 10*12*50, 330 + 0 + 10*5*50\}$

$= \min \{930 + 1500, 360 + 3000 + 6000, 330 + 2500\}$

$= \min \{2430, 9360, 2830\} = 2430$

For $j - i = 4$

m15 = min{m11+ m25+ P0 P1 P5, m12+m35+ P0 P2 P5, m13 + m45 +P0 P3 P5, m14+m55+P0 P4 P5 }

= min{0+2430+5*10*50, 150+930+5*3*50,

330+3000+5*12*50, 405+0+5*5*50}

= min {2430+2500, 150+930+750, 330+3000+3000, 405+1250}

= min {4930, 1830, 6330, 1655} = 1655

Hence, minimum number of scalar multiplications required to multiply the given five matrices is 1655.

To find the optimal parenthesization of A1…….A5, we find the value of k is 4 for which m15 is minimum. So the matrices can be splitted to (A1….A4) (A5). Similarly, (A1….A4) can be splitted to (A1A2) (A3 A4) because for $k = 2$, m14 is minimum. No further splitting is required as the subchains (A1A2) and (A3 A4) has length 1. So the optimal paranthesization of A1 …….A5 in ((A1 A2) (A3 A4)) (A5).

Time complexity of multiplying a chain of *n* matrices

Let $T(n)$ be the time complexity of multiplying a chain of n matrices.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(1) + \sum_{k=1}^{n-1} [T(k) + T(n-k) + \Theta(1)] & \text{if } n > 1 \end{cases}$$

$$\Rightarrow T(n) = \Theta(1) + \sum_{k=1}^{n-1} [T(k) + T(n-k) + \Theta(1)] \quad \text{if } n > 1$$

$$= \Theta(1) + \Theta(n-1) + \sum_{k=1}^{n-1} [T(k) + T(n-k)]$$

$$\Rightarrow T(n) = \Theta(n) + 2[T(1) + T(2) + \quad + T(n-1)] \qquad (7.1)$$

Replacing n by n-1, we get

$$T(n-1) = \Theta(n-1) + 2[T(1) + T(2) + \quad + T(n-2)] \qquad (7.2)$$

Subtracting equation 7.2 from equation 7.1, we have

$$T(n) - T(n-1) = \Theta(n) - \Theta(n-1) + 2T(n-1)$$

$$\Rightarrow T(n) = \Theta(1) + 3T(n-1)$$

$$= \Theta(1) + 3[\Theta(1) + 3T(n-2)] = \Theta(1) + 3\Theta(1) + 3^2 T(n-2)$$

$$= \Theta(1)[1 + 3 + 3^2 + \quad + 3^{n-2}] + 3^{n-1} T(1)$$

$$= \Theta(1)[1 + 3 + 3^2 + \quad + 3^{n-1}]$$

n

$$= \frac{3\ -1}{2} = O(2^n)$$

MODULE 2: Algorithm Design Techniques: [12]
Dynamic Programming

The longest common subsequence (LCS)

The longest common subsequence (LCS) problem can be formulated as follows "Given two sequences $X = \langle x1, x2 \ldots\ldots\ldots xn \rangle$ and $Y = \langle y1, y2 \ldots\ldots\ldots yn \rangle$ and the objective is to find the LCS $Z = \langle z1, z2 \ldots\ldots\ldots zn \rangle$ that is common to x and y"

Given two sequences X and Y, we say Z is a common sub sequence of X and Y if Z is a subsequence of both X and Y. For example, X = ⟨ A, B, C, B, D, A, B⟩ and Y = ⟨ B, D, C, A, B, A⟩, the sequence ⟨B, C, A⟩ is a common subsequence. Similarly, there are many common subsequences in the two sequences X and Y. However, in the longest common subsequence problem, we wish to find a maximum length common subsequence of X and Y, that is ⟨ B, C, B, A⟩ or ⟨ B, D, A, B⟩. This section shows that the LCS problem can be solved efficiently using dynamic programming.

Dynamic programming for LCS problem Theorem (Optimal Structure of an LCS)

Let $X = \langle x_1, x_2 \ldots\ldots\ldots x_n \rangle$ and $Y = \langle y_1, y_2 \ldots\ldots\ldots y_n \rangle$ be sequences and let $Z = \langle z_1, z_2 \ldots\ldots\ldots z_n \rangle$ be any LCS of X and Y.

Case 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an *LCS* of X_{m-1} and Y_{n-1}.

Case 2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an *LCS* of X_{m-1} and Y.

Case 3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof The proof of the theorem is presented below for all three cases.

Overlapping Sub-problems

From theorem, it is observed that either one or two cases are to be examined to find an LCS of $X_m$and Y_n. If $x_m = y_n$, then we must find an LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must find an LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1}. The LCS of X and Y is the longer of these two LCSs.

Let us define $c[m, n]$ to be the length of an LCS of the sequences X_m and Y_n. The optimal structure of the LCS problem gives the recursive formula

$$c[m,n] = \begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ c[m-1, n-1] + 1 & \text{if } x_m = y_n \\ \max\{c[m-1,n], c[m,n-1]\} & \text{if } x_m \neq y_n \end{cases} \quad \text{.....(1)}$$

Generalizing equation 7.1, we can formulate

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } x_i \neq y_j \end{cases} \quad \text{.....(2)}$$

Module 2 Algorithm Design Techniques: [12]
Lecture 10 Dynamic Programming

Computing the length of an LCS

Based on equation (1), we could write an exponential recursive algorithm but there are only $m*n$ distinct problems. Hence, for the solution of $m*n$ distinct subproblems, we use dynamic programming to compute the solution using bottom up approach.

The algorithm LCS_length (X, Y) takes two sequences $X = \langle x1, x2 \ldots\ldots\ldots x_n \rangle$ and $Y = \langle y1, y2 \ldots\ldots\ldots y_n \rangle$ as inputs and find $c[m, n]$ as the maximum length of the subsequence in X and Y. It stores $c[i, j]$ and $b[i, j]$ in tables $c[m, n]$ and $b[m, n]$ respectively, which simplifies the construction of optimal solution.

AlgorithmLCS_LENGTH (X, Y)

```
{
    m=length [X]
    n=length [Y]
    for( i =1; i<=m;
        i++) c[i,0] =
        0;
```

```
    for(j=0; j<n;
        j++) c[0, j]=
        0;
    for(i=1;  i< m; i++){
     for(j = 1; j <= n;  j++){
            if(x[i] = = y[j]) {
                c[i, j] = 1 + c[i-1, j-
                1]; b[i,jj] ← ‘    ’;
            }
        else{
            if(c[ i-1, j] ≥ c[i, j-1] )
                c[i, j] = c[i-1, j];
                b[i,j] = ‘↑’;
            else
                c[i,j] = c[i, j-
                1]; b[i, j] =
                ‘←’
        }
    return c and b ;
}
```

Constructing an LCS

The algorithm LCS_LENGTH returns *c* and *b* tables. The *b* table can be used to construct the LCS of *X* and *Y* quickly.

```
Algorithm PRINT_LCS (b, X, i, j)
{if (i == 0 || j == 0) return;
   if (b[i, j  ] = =↖  ‘) {
```

```
        PRINT_LCS (b, X, i-1, j-1)
        Print xi
    }
    else if (b[i, j] = = '↑')
        PRINT_LCS (b, X, i-1, j)
    else
        PRINT_LCS (b, X, i, j-1)
}
```

Let us consider two sequences $X = \langle C, R, O, S, S \rangle$ and $Y = \langle R, O, A, D, S \rangle$ and the objective is to find the LCS and its length. The longest common subsequence of X and Y is $\langle R, O, S \rangle$ and the length of LCS is 3.

MODULE 2: Algorithm Design Techniques: [12]

Greedy Approach

Introduction

Greedy algorithms are typically used to solve an optimization problem. An Optimization problem is one in which, we are given a set of input values, which are required to be either maximized or minimized w. r. t. some constraints or conditions. Generally an optimization problem has n inputs (call this set as input domain or Candidate set, C), we are required to obtain a subset of C (call it solution set, S where $S \subseteq C$) that satisfies the given constraints or conditions. Any subset S, $S \subseteq C$, which satisfies the given constraints, is called a feasible solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that does this is called an optimal solution.

A greedy algorithm proceeds step–by-step, by considering one input at a time. At each stage, the decision is made regarding whether a particular input (say x) chosen gives an optimal solution or not. Our choice of selecting input x is being guided by the selection function (say select). If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set. On the other hand, if the inclusion of that input x results in an infeasible solution, then this input x is not added to the partial solution. The input we tried and rejected is never considered again. When a greedy algorithm works correctly, the first solution found in this way is always optimal.

In brief, at each stage, the following activities are performed in greedy method:

1. First we select an element, say X , from input domain C.
2. Then we check whether the solution set S is feasible or not. That is we check whether x can

be included into the solution set S or not. If yes, then solution set $S \leftarrow S \cup \{X\}$. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.

3. Continue until S is filled up (i.e. optimal solution found) or C is exhausted whichever is earlier.

(Note: From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function (either maximize or minimize, as the case may be), is called **optimal solution.**

In this Chapter, we will discuss those problems for which greedy algorithm gives an optimal solution such as Knapsack problem, Minimum cost spanning tree (MCST) problem and Single source shortest path problem.

Objective

After going through this Unit, you will be able to:

1. Understand the basic concept about Greedy approach to solve Optimization problem.
2. Understand how Greedy method is applied to solve any optimization problem such as Knapsack problem, Minimum-spanning tree problem, Shortest path problem etc.

Fractional Knapsack Problem

Let there are n number of objects and each object is having a weight and contribution to profit. The knapsack of capacity M is given. The objective is to fill the knapsack in such a way that profit shall be maximum. We allow a fraction of item to be added to the knapsack.

Mathematically, we can write

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{Subject to} \quad \sum_{i=1}^{n} w_i x_i \le M$$

$$1 \le i \le n \text{ and } 0 \le x_i \le 1.$$

Where pi and wi are the profit and weight of ith object and xi is the fraction of ith object to be selected. Note that the value of x_i will be any value between 0 and 1 (inclusive). If any object is completely placed into a knapsack then its value is 1 (i.e. x_i=1) , if we do not pick (or select) that object to fill into a knapsack then its value is 0 (i.e. x_i=0). Otherwise if we take a fraction of any object then its value will be any value between 0 and 1.

For example

Given n = 3, (p1, p2, p3) = {25, 24, 15}

(w1, w2, w3) = {18, 15, 10} M = 20

Solution

Some of the feasible solutions are shown in the following table.

Solution No	x1	x2	x3	∑wi xi	∑pi xi
1	1	2/15	0	20	28.2
2	0	2/3	1	20	31.0
3	0	1	1/2	20	31.5

These solutions are obtained by different greedy strategies.

Greedy strategy I: In this case, the items are arranged by their profit values. Here the item with maximum profit is selected first. If the weight of the object is less than the remaining capacity of the knapsack then the object is selected full and the profit associated with the object is added to the total profit. Otherwise, a fraction of the object is selected so that the knapsack can be filled exactly. This process continues from selecting the highest profitable object to the lowest profitable object till the knapsack is exactly full.

Greedy strategy II: In this case, the items are arranged by fair weights. Here the item with minimum weight in selected first and the process continues like greedy strategy-I till the knapsack is exactly full.

Greedy strategy III: In this case, the items are arranged by profit/weight ratio and the item with maximum profit/weight ratio is selected first and the process continues like greedy strategy-I till the knapsack is exactly full.

Therefore, it is clear from the above strategies that the Greedy method generates optimal solution if we select the objects with respect to their profit to weight ratios that means the object with maximum profit to weight ratio will be selected first. Let there are n objects and the object i is associated with

```
Greedy Fractional-Knapsack (P[1..n],W[1..n],X[1..n], M)
        /* P[1..n]and W[1..n]contains the profit and weight of then-objects ordered
        such that X[1..n]is a solution set and M is the capacity of KnapSack*/
{           For i← 1to n do
                X[i]← 0
:

                profit← 0       //Total profit of item filled in Knapsack
                weight←         //Total weight of items packed in
                0i←1
        While(Weight<M)//M is the Knap sack
        Capacity
        {
                if(weight +W[i]≤M)
                        X[i]= 1
                        weight= weight+ W[i]
                else
                X[i]= (M-
wright)/w[i]weight= M
        Profit =profit+ p[i]*X[i]i++;
}//end of while
}//end of Algorithm
```

Running time of Knapsack (fractional) problem:

Sorting of n items (or objects) in decreasing order of the ratio profit/weight takes $O(n \log n)$ time.

Since this is the lower bound for any comparison based sorting algorithm. Line 6 of Greedy Fractional-Knapsack takes $O(n)$ time. Therefore, the total time including sort is $O(n \log n)$.

Example: 1: Find an optimal solution for the knapsack instance n=7 and M=15,

Solution:

Greedy algorithm gives a optimal solution for knapsack problem if you elect the object in decreasing of the ratio profit/weight. That is we select those object first which has maximum value of the ratio . This ratio is also called profit per unit weight . Since $\left(\frac{p_1}{w_1}, \frac{p_2}{w_2}, \ldots, \frac{p_7}{w_7}\right) = (5, 1.67, 3, 1, 6, 4.5, 3)$. Thus we select 5th object first , then 1st object, then 3rd (or 7th) object, and so on.

Approach	$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$		
Selection Object in decreasing order of the ratio		1+2+4+5+1+2 =15	6+10+18+15+3+3.33 =55.33

Kruskal's Algorithm

This minimum spanning tree algorithm was first described by Kruskal in 1956 in the same paper where he rediscovered Jarnik's algorithm. This algorithm was also rediscovered in 1957 by Loberman and Weinberger, but somehow avoided being renamed after them. The basic idea of the Kruskal's algorithms is as follows: scan all edges in increasing weight order; if an edge is safe, keep it (i.e. add it to the set A).

Overall Strategy

Kruskal's Algorithm, as described in CLRS, is directly based on the generic MST algorithm. It builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm consider each edge in turn, order by increasing weight. If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree on the other hand, if an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

A little more formally, given a connected, undirected, weighted graph with a function $w : E \rightarrow R$.

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Data Structure

Before formalizing the above idea, lets quickly review the disjoint-set data structure from Chapter 21.

Make_SET(v): Create a new set whose only member is pointed to by v. Note that for this operation v must already be in a set.

FIND_SET(v): Returns a pointer to the set containing v.

UNION(u, v): Unites the dynamic sets that contain u and v into a new set that is union of these two sets.

Algorithm

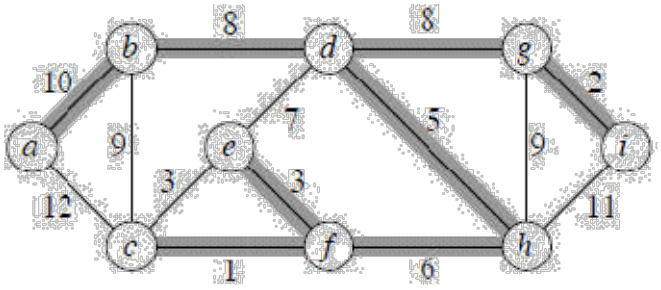
Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in the graph.

```
KRUSKAL(V, E, w)

A ← { }          // Set A will ultimately contains the edges of the MST
for each vertex v in V
   do MAKE-SET(v)
sort E into nondecreasing order by weight w
for each (u, v) taken from the sorted list
   do if FIND-SET(u) = FIND-SET(v)
      then A ← A ∪ {(u, v)}
         UNION(u, v)
return A
```

Illustrative Examples

Lets run through the following graph quickly to see how Kruskal's algorithm works on it:



We get the shaded edges shown in the above figure.

Edge (c, f) : safe
Edge (g, i) : safe

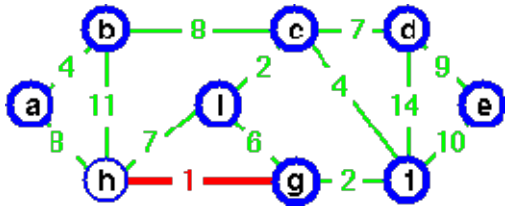
Edge (e, f) : safe
Edge (c, e) : reject
Edge (d, h) : safe
Edge (f, h) : safe
Edge (e, d) : reject
Edge (b, d) : safe
Edge (d, g) : safe
Edge (b, c) : reject
Edge (g, h) : reject
Edge (a, b) : safe

At this point, we have only one component, so all other edges will be rejected. [We could add a test to the main loop of KRUSKAL to stop once $|V| - 1$ edges have been added to A.]

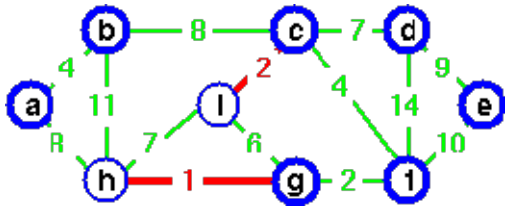
Note Carefully: Suppose we had examined (c, e) before (e, f). Then would have found (c, e) safe and would have rejected (e, f).

Example (CLRS) Step-by-Step Operation of Kurskal's Algorithm.

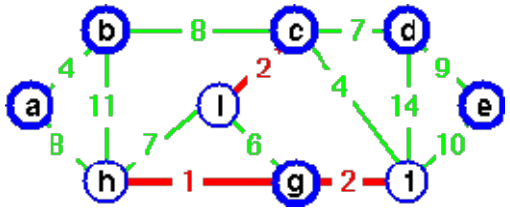
Step 1. In the graph, the Edge(*g*, *h*) is shortest. Either vertex *g* or vertex *h* could be representative. Lets choose vertex g arbitrarily.



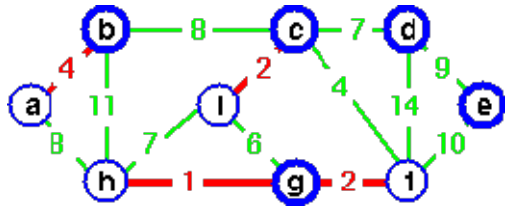
Step 2. The edge (*c*, *i*) creates the second tree. Choose vertex *c* as representative for second tree.



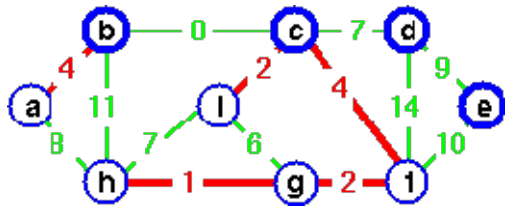
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



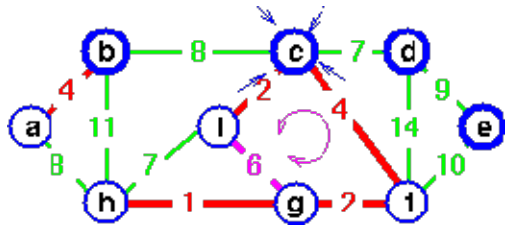
Step 4. Edge (a, b) creates a third tree.



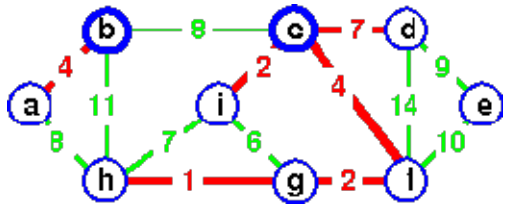
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



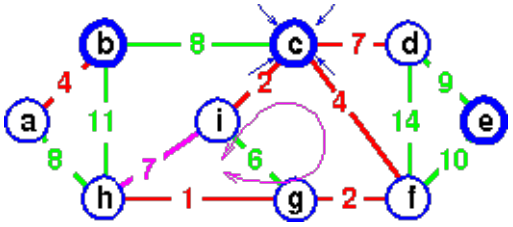
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



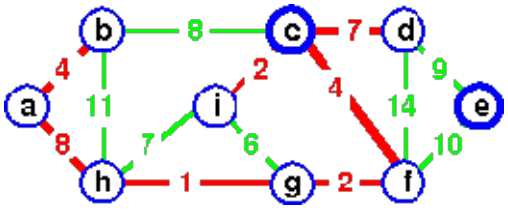
Step 7. Instead, add edge (c, d).



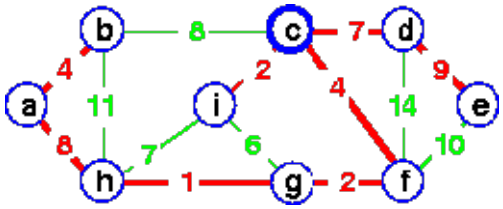
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).



Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



Analysis

Initialize the set A: O(1)

First for loop: |V| MAKE-SETs

Sort E: O(E lg E)

Second for loop: O(E) FIND-SETs and UNIONs

Assuming the implementation of disjoint-set data structure, already seen in Chapter 21, that uses union by rank and path compression: $O((V + E)\,\alpha(V)) + O(E \lg E)$

Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.

$\alpha(|V|) = O(\lg V) = O(\lg E)$.

Therefore, total time is $O(E \lg E)$.

$|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.

Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\, \alpha(V))$, which is almost linear.)

MODULE 2: Algorithm Design Techniques:[12]

Backtracking

A backtracking algorithm tries to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it simply tries all possible options recursively.

- It is one of the most general algorithm design techniques.
- Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.
- To apply backtracking method, tne desired solution must be expressible as an n-tuple (x1…xn) where xi is chosen from some finite set Si.
- The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn).
- The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (m_{i+1}……..m_n) possible test vectors may be ignored entirely.
- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.
- These constraints are classified as:

i) Explicit constraints.
ii) Implicit constraints.

Explicit constraints:

Explicit constraints are rules that restrict each Xi to take values only from a given set.

Some examples are,

$Xi \geq 0$ or Si = {all non-negative real nos.}
Xi =0 or 1 or Si={0,1}.
$Li \leq Xi \leq Ui$ or Si= {a: $Li \leq a \leq Ui$}

- All tupules that satisfy the explicit constraint define a possible solution space for I.

Implicit constraints:

The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

Algorithm:

```
Algorithm I Backtracking (n)
// This schema describes the backtracking procedure .All solutions are generated in X[1:n]
//and printed as soon as they are determined.
 {
   k=1;
   While (k ≠ 0) do
   {
     if (there remains all untried
     X[k] ∈ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then
     {
       if(X[1],……X[k] )is the path to the answer node)
      Then write(X[1:k]);
      k=k+1;              //consider the next step.
    }
 else k=k-1;                  //consider backtracking to the previous set.
 }
}
```

- All solutions are generated in X[1:n] and printed as soon as they are determined.
- T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],……..X[k-1] have already been chosen.
- B_k(X[1]……X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.
- Certain problems which are solved using backtracking method are,

    1. **Sum of subsets.**
    2. **Graph coloring.**
    3. **Hamiltonian cycle.**
    4. **N-Queens problem.**

Sum of subsets:

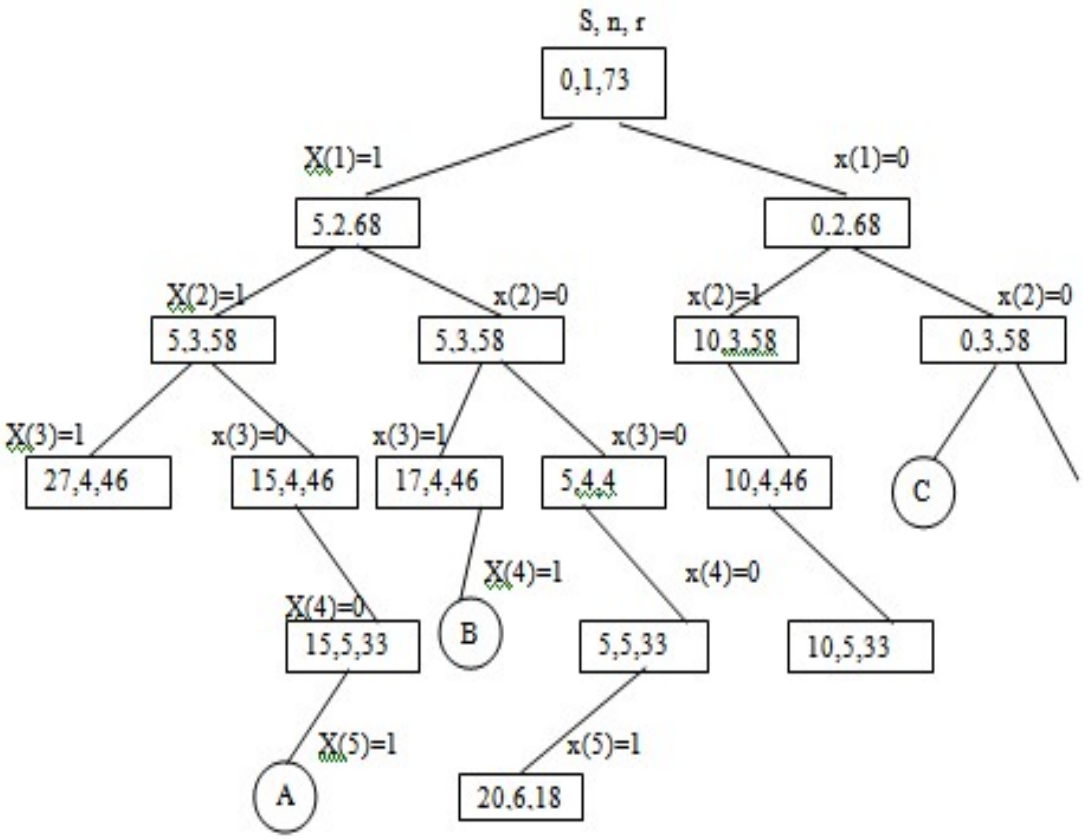
- We are given ‘n’ positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.
- If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.
- If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and right to X(i)=0.

Example:

- Given n=6,M=30 and W(1…6)=(5,10,12,13,15,18).We have to generate all possible combinations of subsets whose sum is equal to the given value M=30.

- In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets,'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.

- The state space tree for the given problem is,



I[st] solution is **A** -> 1 1 0 0 1 0
II[nd] solution is **B** -> 1 0 1 1 0 0
III [rd] solution is **C** -> 0 0 1 0 0 1

- In the state space tree, edges from level 'i' nodes to 'i+1' nodes are labeled with the values of Xi, which is either 0 or 1.

- The left sub tree of the root defines all subsets containing Wi.

- The right subtree of the root defines all subsets, which does not include Wi.

Generation of state space tree:

- Maintain an array X to represent all elements in the set.
- The value of Xi indicates whether the weight Wi is included or not.
- Sum is initialized to 0 i.e., s=0.
- We have to check starting from the first node.
- Assign X(k)<- 1.
- If S+X(k)=M then we print the subset b'coz the sum is the required output.
- If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to check for the next k.
- After generating the left sub tree we have to generate the right sub tree, for this we have to check S+W(k+1)<=M.B'coz W(k) is omitted and W(k+1) has to be selected.
- Repeat the process and find all the possible combinations of the subset.

Algorithm:

```
Algorithm sumofsubset(s,k,r)
{
//generate the left child. note s+w(k)<=M since Bk-1 is true.
X{k]=1;
If (S+W[k]=m) then write(X[1:k]); // there is no recursive call here as W[j]>0,1<=j<=n.
Else if (S+W[k]+W[k+1]<=m) then sum of sub (S+W[k], k+1,r- W[k]);
//generate right child and evaluate Bk.
If ((S+ r- W[k]>=m)and(S+ W[k+1]<=m)) then
{
  X{k]=0;
  sum of sub (S, k+1, r- W[k]);
}
}
```

Hamiltonian cycles:

- Let G=(V,E) be a connected graph with 'n' vertices. A HAMILTONIAN CYCLE is a round trip path along 'n' edges of G which every vertex once and returns to its starting position.
- If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertex are visited in the order of V1,V2…….Vn+1,then the edges are in E,1<=I<=n and the Vi are distinct except V1 and Vn+1 which are equal.
- Consider an example graph G1.

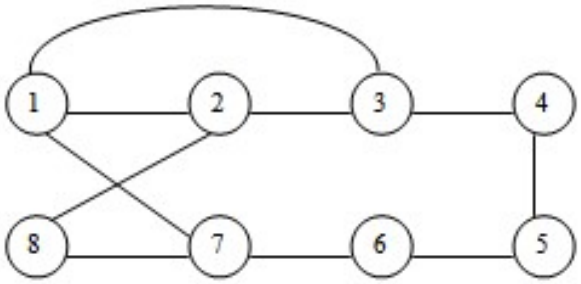


Figure 2. 6 The graph G1 has Hamiltonian cycles:

->1, 3, 4,5,6,7,8,2,1 and
->1, 2, 8,7,6,5,4,3,1.

The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

Procedure:

1. Define a solution vector X(Xi……..Xn) where Xi represents the I th visited vertex of the proposed cycle.

2. Create a cost adjacency matrix for the given graph.

3. The solution array initialized to all zeros except X(1)=1,b'coz the cycle should start at vertex '1'.

4. Now we have to find the second vertex to be visited in the cycle.
5. The vertex from 1 to n are included in the cycle one by one by checking 2 conditions,
   1.There should be a path from previous visited vertex to current vertex.
   2.The current vertex must be distinct and should not have been visited earlier.

6. When these two conditions are satisfied the current vertex is included in the cycle, else the next vertex is tried.

7. When the nth vertex is visited we have to check, is there any path from nth vertex to first 8 vertex. if no path, the go back one step and after the previous visited node.

8. Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm: (Finding all Hamiltonian cycle)

Algorithm Hamiltonian (k)

```
{
 Loop
        Next value (k)
If (x (k)=0) then return;
{
  If k=n then
     Print (x)
Else
Hamiltonian (k+1);
End if

}
Repeat
}

Algorithm Nextvalue (k)
{
 Repeat
{
  X [k]=(X [k]+1) mod (n+1); //next vertex
  If (X [k]=0) then return;
  If (G [X [k-1], X [k]] ≠ 0) then
{
  For j=1 to k-1 do if (X [j]=X [k]) then break;
  // Check for distinction.
  If (j=k) then          //if true then the vertex is distinct.
    If ((k<n) or ((k=n) and G [X [n], X [1]] ≠ 0)) then return;
}
} Until (false);
}
```

8-queens problem:

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector X (X1…Xn) represents a solution in which Xi is the column of the th row where I th queen is placed.
- First, we have to check no two queens are in same row.
- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in it diagonal.

- Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same value.

- Also, every element on the same diagonal that runs from lower right to upper left has the same value.

- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

Steps to generate the solution:

- ❖ Initialize x array to zero and start by placing the first queen in k=1 in the first row.
- ❖ To find the column position start from value 1 to n, where 'n' is the no. Of columns or no. Of queens.
- ❖ If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have to check whether there is any queen in the same column or diagonal.
- ❖ For this considers the previous position, which had already, been found out. Check whether
  X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.
- ❖ If any one of the conditions is true then return false indicating that k th queen can't be placed in position X (k).
- ❖ For not possible condition increment X (k) value by one and precede d until the position is found.
- ❖ If the position X (k) ≤ n and k=n then the solution is generated completely.
- ❖ If k<n, then increment the 'k' value and find position of the next queen.
- ❖ If the position X (k)>n then k th queen cannot be placed as the size of the matrix is 'N*N'.
- ❖ So decrement the 'k' value by one i.e. we have to back track and after the position of the previous queen.

Algorithm:
Algorithm place (k,I)
//return true if a queen can be placed in k th row and I th column. otherwise it returns //
//false .X[] is a global array whose first k-1 values have been set. Abs® returns the //absolute value of r.

```
{
 For j=1 to k-1 do
   If ((X [j]=I)            //two in same column.
   Or (abs (X [j]-I)=Abs (j-k)))
Then return false;
Return true;
}
```

***n*-Queens:**

The prototypical backtracking problem is the classical ***n Queens Problem***, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym "Schachfreund") for the standard 8× 8 board and by François-Joseph Eustache Lionnet in 1869 for the more general *n n* board. The problem is to place *n* queens on an *n n* chessboard, so that no two queens can attack

each other. For readers not familiar with the rules of chess, this means that no two queens are in the same row, column, or diagonal.

Obviously, in any solution to the *n*-Queens problem, there is exactly one queen in each row. So we will represent our possible solutions using an array $Q[1 .. n]$, where $Q[i]$ indicates which square in row i contains a queen, or 0 if no queen has yet been placed in row i. To find a solution, we put queens on the board row by row, starting at the top. A *partial* solution is an array $Q[1 .. n]$ whose first $r-1$ entries are positive and whose last $n-r+1$ entries are all zeros, for some integer r.

The following recursive algorithm, essentially due to Gauss (who called it "methodical groping"), recursively enumerates all complete *n*-queens solutions that are consistent with a given partial solution. The input parameter r is the first empty row. Thus, to compute all *n*-queens solutions with no restrictions, we would call RecursiveNQueens($Q[1 .. n]$,1).

Algorithm Nqueen (k,n)

```
//using backtracking it prints all possible positions of n queens in 'n*n' chessboard. So
 //that they are non-tracking.
{
   For I=1 to n do
      {
        If place (k,I) then
          {
            X [k]=I;
             If (k=n) then write (X [1:n]);
              Else nquenns(k+1,n)    ;
         }
    }
}
```

Example: 4 queens.
Two possible solutions are

	Q		
			Q
Q			
		Q	

Solutin-1
(2 4 1 3)

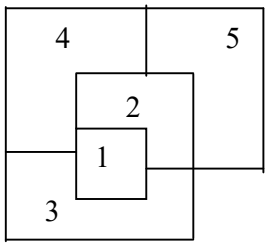
		Q	
Q			
			Q
	Q		

Solution 2
(3 1 4 2)

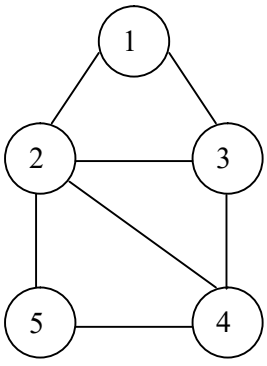
Graph coloring:

- Let ‘G’ be a graph and ‘m’ be a given positive integer. If the nodes of ‘G’ can be colored in such a way that no two adjacent nodes have the same color. Yet only ‘M’ colors are used. So it’s called M-color ability decision problem.
- The graph G can be colored using the smallest integer ‘m’. This integer is referred to as chromatic number of the graph.
- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.
2 is adjacent to 1, 3, 4, 5
3 is adjacent to 1, 2, 4
4 is adjacent to 1, 2, 3, 5
5 is adjacent to 2, 4



Steps to color the Graph:

- First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.

- The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),……..,X(n) ,X(index) is the color, index is the node.

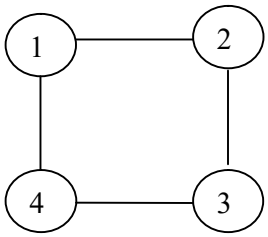
- He formula is used to set the color is,
    - X(k) = (X(k)+1) % (m+1)

- First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.

- Repeat the procedure until all possible combinations of colors are found.

- The function which is used to check the adjacent nodes and same color is,
    - If((Graph (k,j) == 1) and X(k) = X(j))

Example:



N= 4
M= 3

Adjacency Matrix:

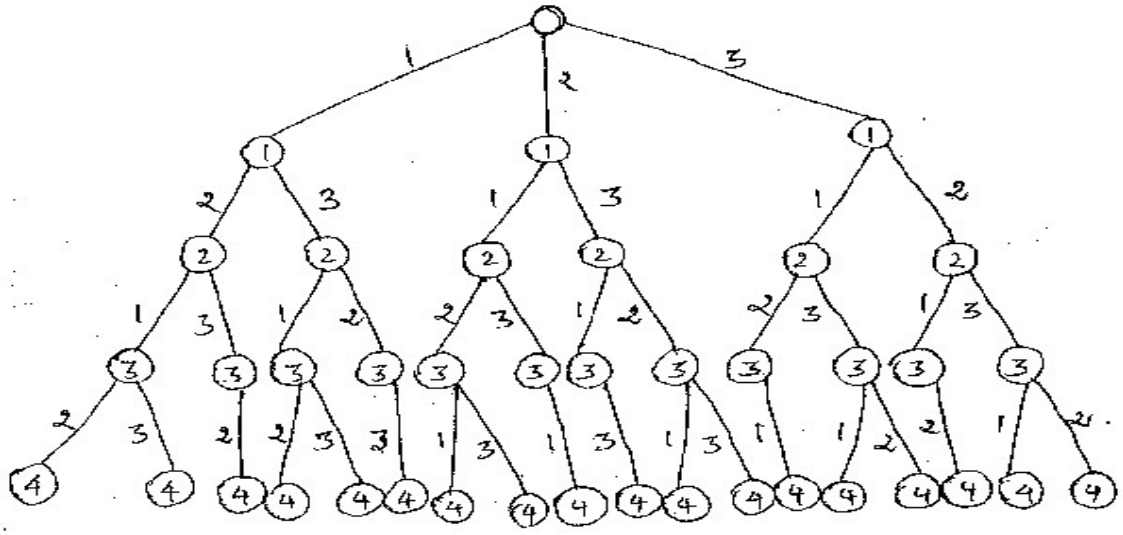
$$\begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix}$$

→ Problem is to color the given graph of 4 nodes using 3 colors.

→Node-1 can take the given graph of 4 nodes using 3 colors.

→ The state space tree will give all possible colors in that ,the numbers which are inside the circles are nodes ,and the branch with a number is the colors of the nodes.

State Space Tree:



Algorithm:

Algorithm mColoring(k)

// the graph is represented by its Boolean adjacency matrix G[1:n,1:n] .All assignments //of 1,2,…….,m to the vertices of the graph such that adjacent vertices are assigned //distinct integers are printed. 'k' is the index of the next vertex to color.

```
{
repeat
{
   // generate all legal assignment for X[k].
  Nextvalue(k);    // Assign to X[k] a legal color.
       If (X[k]=0) then return;          // No new color possible.
      If (k=n) then                 // Almost 'm' colors have been used to color the 'n' vertices
              Write(x[1:n]);
     Else mcoloring(k+1);
}until(false);
}
```

Algorithm Nextvalue(k)

// X[1],……X[k-1] have been assigned integer values in the range[1,m] such that //adjacent values have distinct integers. A value for X[k] is determined in the //range[0,m].X[k] is assigned the next highest numbers color while maintaining //distinctness form the adjacent vertices of vertex K. If no such color exists, then X[k] is 0.

```
{

  repeat
   {
        X[k] = (X[k]+1)mod(m+1);    // next highest color.
       If(X[k]=0) then return;          //All colors have been used.
         For j=1 to n do
         {
```

```
        // Check if this color is distinct from adjacent color.
      If((G[k,j] ≠ 0)and(X[k] = X[j]))
        // If (k,j) is an edge and if adjacent vertices have the same color.
      Then break;
      }

    if(j=n+1) then return;     //new color found.
  } until(false);    //otherwise try to find another color.
}
```

→ The time spent by Nextvalue to determine the children is θ(mn)

→Total time is = $\theta(m^n n)$.

Knapsack Problem using Backtracking:

- The problem is similar to the zero-one (0/1) knapsack optimization problem is dynamic programming algorithm.

- We are given 'n' positive weights Wi and 'n' positive profits Pi, and a positive number 'm' that is the knapsack capacity, the is problem calls for choosing a subset of the weights such that,

$$\sum_{1 \le i \le n} WiXi \le m \text{ and } \sum_{1 \le i \le n} PiXi \text{ is Maximized.}$$

Xi →Constitute Zero-one valued Vector.

- The Solution space is the same as that for the sum of subset's problem.

- Bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node.

- The profits and weights are assigned in descending order depend upon the ratio.

(i.e.) Pi/Wi ≥ P(I+1) / W(I+1)

Solution :

- After assigning the profit and weights, we have to take the first object weights and check if the first weight is less than or equal to the capacity, if so then we include that object (i.e.) the unit is 1.(i.e.) K→ 1.

- Then We are going to the next object, if the object weight is exceeded that object does not fit. So unit of that object is '0'.(i.e.) K=0.
- Then We are going to the bounding function, this function determines an upper bound on the best solution obtainable at level K+1.

- Repeat the process until we reach the optimal solution.

Algorithm:

Algorithm Bknap(k,cp,cw)

// 'm' is the size of the knapsack; 'n' → no.of weights & profits. W[]&P[] are the //weights & weights. P[I]/W[I] ≥ P[I+1]/W[I+1].
//fw→Final weights of knapsack.
//fp→ final max.profit.
//x[k] = 0 if W[k] is not the knapsack,else X[k]=1.

```
{
     // Generate left child.
      If((W+W[k] ≤ m) then
      {
            Y[k] =1;
             If(k<n) then Bnap(k+1,cp+P[k],Cw +W[k])
               If((Cp + p[w] > fp) and (k=n)) then

                 {
                   fp = cp + P[k];
                   fw = Cw+W[k];
                  for j=1 to k do X[j] = Y[j];
                 }
   }

 if(Bound(cp,cw,k) ≥ fp) then
 {
      y[k] = 0;
    if(k<n) then Bnap (K+1,cp,cw);
  if((cp>fp) and (k=n)) then
     {
        fp = cp;
         fw = cw;
           for j=1 to k do X[j] = Y[j];
      }
   }
}
```

Algorithm for Bounding function:

Algorithm Bound(cp,cw,k)
// cp→ current profit total.
//cw→ current weight total.
//k→the index of the last removed item.
//m→the knapsack size.

```
{
   b=cp;
   c=cw;
```

```
    for I =- k+1 to n do
  {
        c= c+w[I];
       if (c<m) then b=b+p[I];
          else return b+ (1-(c-m)/W[I]) * P[I];
}
return b;
}
```

Example:

M= 6 Wi = 2,3,4 4 2 2

N= 3 Pi = 1,2,5 Pi/Wi (i.e.) 5 2 1

Xi = 1 0 1
The maximum weight is 6

The Maximum profit is (1*5) + (0*2) + (1*1)
→ 5+1
→ 6.

Fp = (-1)

- $1 \leq 3$ & $0+4 \leq 6$
  cw = 4,cp = 5,y(1) =1
  k = k+2

- $2 \leq 3$ but 7>6
  so y(2) = 0

- So bound(5,4,2,6)

B=5
C=4
I=3 to 3
C=6
$6 \neq 6$
So return 5+(1-(6-6))/(2*1)

- 5.5 is not less than fp.
  So, k=k+1 (i.e.) 3.
  3=3 & $4+2 \leq 6$
  cw= 6,cp = 6, y(3)=1.
  K=4.
- If 4> 3 then
  Fp =6,fw=6,k=3 ,x(1) 1 0 1
  The solution Xi → 1 0 1

  Profit → 6
  Weight →6.

MODULE: 02 MCQ and Short type Problem
MCQ:

1. For 0/1 KNAPSACK problem, the algorithm takes____amount of time for memory table, and_____time to determine the optimal load, for N objects and W as the capacity of KNAPSACK.

a . O(N+W), O(NW) (b) O(NW),O(N+W) (c)O(N),O(NW) (d) O(NW),O(N)
Ans :(b) O(NW),O(N+W)

2. .The divide and conquer merge sort algorithm's time complexity can be defined as
a. O(long n)
b. O(n)
c. Ω (n log n)
d. O(n log n)
Ans: O(n log n)

3. Sorting is not possible by using which of the following methods?
Insertion (b) Selection (c) Deletion (d) Exchange
Ans :Deletion
4. What is the type of the algorithm used in solving the 8 Queens problem?
Backtracking (b) Dynamic (c) Branch and Bound (d) D and C
Ans :Backtracking
5. The following are the statements regarding the NP problems. Chose the right option from the following options:
All NP-complete problems are not NP-hard.
SomeNP-hard problems are not known to be NP-complete.
Both (I) and (II) are true
Both (I) and (II) are false
Only (I) is true
Only (II) is true

Ans :Only (II) is true

6. Let G be a graph with 'n' nodes and let 'm' be the chromatic number of the graph. Then the time taken by the backtracking algorithm to color it is
a. O(nm) (b) O(n+m) (c) O(mnm) (d) O(nmn).
Ans :O(nmn).
7. The time complexity of the shortest path algorithm can be bounded by
a. O(n2) (b) O(n4) (c) O(n3) (d) O(n)
Ans : O(n3)
8. Read the following statements carefully and pick the correct option:
I. The worst time complexity of the Floyd's algorithm is O(n3).
II. The worst time complexity of the Warshall's algorithm is O(n3).

(a) (I) is false but (II) is true
(b) (I) is true but (II) is false
(c)Both (I) and (II) are true
(d) (I) is true and (II) is not true always
(e) Both (I) and (II) are false.
Ans :Both (I) and (II) are true

9. For the bubble sort algorithm, what is the time complexity of the best/worst case?(assume that the computation stops as soon as no more swaps in one pass)
   a. best case: $O(n)$ worst case: $O(n*n)$
   b. best case: O(n) worst case: O(n*log(n))
   c. best case: O(n*log(n)) worst case: O(n*log(n))
   d. best case: O(n*log(n)) worst case: O(n*n)

Ans : best case: $O(n)$ worst case: $O(n*n)$

10. For the quick sort algorithm, what is the time complexity of the best/worst case?
   a. best case: O(n) worst case: O(n*n)
   b. best case: O(n) worst case: O(n*log(n))
   c. best case: O(n*log(n)) worst case: O(n*log(n))
   d. best case: O(n*log(n)) worst case: O(n*n)

Ans :best case: $O(n*log(n))$ worst case: $O(n*n)$

11. In an arbitrary tree (not a search tree) of order M. Its size is N, and its height is K.
The computation time needed to find a data item on T is
   a. O(K*K)
   b. O(M*M)
   c. O(N)
   d. O(K)

Ans : $O(N)$

12. Which of the following belongs to the algorithm paradigm?
   a. Minimum & Maximum problem
   b. Knapsack problem
   c. Selection problem
   d. Merge sort
   e. Quick sort.

Ans : Knapsack problem

13. The time taken by NP-class sorting algorithm is
   a. O(1)
   b. O(log n)
   c. O(n2)
   d. O(n)

Ans :O(n)

14. Find the odd one out from the following categories of algorithms.
    a. TVSP
    b. N-Queens
    c. 15-Puzzle
    d. Bin-Packing.

Ans : Bin-Packing.

15. The time complexity of binary search in best, worst cases for an array of size N is
    a. N, N2
    b. 1, Log N
    c. Log N, N2
    d. 1, N log N

Ans : 1, Log N

16. Which of following algorithm scans the list by swapping the entries whenever pair of adjacent keys are out of desired order?
    a. Insertion sort
    b. Quick sort
    c. Shell sort
    d. Bubble sort

Ans : Bubble sort

Sample Questions

1 What is Tower of Hanoi Problem?

2 Write a recursive algorithm to solve the problem and find out the time complexity.

3 Find time complexity of merge sort. State its best case, worst case complexity.

4 State the general greedy algorithm in problem solving.

5 Write a Greedy Algorithm to find the optimal solution of the Knapsack Problem. What is the time complexity?

6 What do you mean by recursion tree? Derive the worst case complexity of quick sort using recursion tree.

7 What is difference between Dynamic Programming and Greedy Algorithm?

8 Explain the effect of negative weight edges in finding the shortest path from a source vertex to all the other vertices reachable from the source vertex.

9 Write any Algorithm for single source shortest path problem of a directed graph.

10 Show the working of the algorithm on the graph. Consider A as the source vertex. Find the single pair shortest path?

11 Explain the Backtracking approach with an example.

12 Distinguish between backtracking approach and branch and bound technique.

13 Write an algorithm for N-Queen problem. Explain 4-Queen problem

14 Write an algorithm to find out longest common subsequence

15 Find time complexity of merge sort. State its best case, worst case complexity.

16 Write an algorithm to find fibonacci series which running time is linear.

17 Explain the Backtracking approach with an example.

18 Write an algorithm for N-Queen problem. Explain 4-Queen problem.

19 Find time complexity of merge sort. State its best case, worst case complexity.

20 Find an optimal solution of *parenthesization* of a matrix-chain product whose sequence of dimension <3,5,4,5,4>, also show the m-table and s-table and the parse tree.

MODULE 03: Network Flow: [3L]

When we concerned ourselves with shortest paths and minimum spanning trees, we interpreted the edge weights of an undirected graph as distances. In this lecture, we will ask a question of a different sort. We start with a directed weighted graph G with two distinguished vertices s (the source) and t (the sink). We interpret the edges as unidirectional liquid pipes, communication channel, with an edge's capacity indicated by its weight. The maximum flow problem then asks, how to maximize the flow of liquid as possible from s to t.

Definition. A **flow network** is a directed graph G=(V , E) with distinguished vertices s (the source) and t (the sink), in which each edge (u, v) ∈ E has a nonnegative capacity c(u, v). We require that E never contain both (u, v) and (v, u) for any pair of vertices u, v (so in particular, there are no loops). Also, if u, v ∈ V with (u, v) ∈ E, then we define c(u, v) to be zero. (See Figure .3.1).

In these notes, we will always assume that our flow networks are finite. Otherwise, it would be quite difficult to run computer algorithms on them.

Definition. Given a flow network G= (V , E), a **flow** in G is a function f : V×V → **R** satisfying

1. Capacity constraint: 0 ·<f (u, v) <= c(u, v) for each u, v∈ V

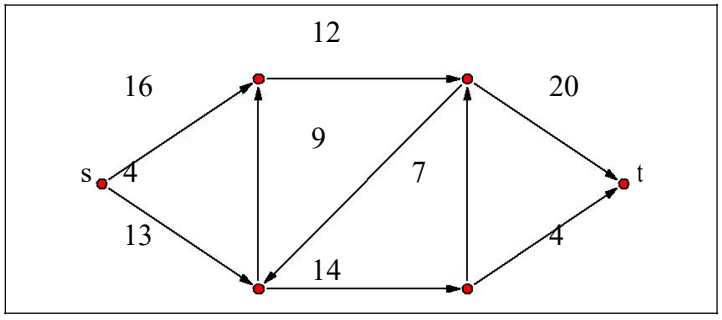


Figure 3.1. A flow network

Network Flow:
Flow network is a directed graph G=(V,E) such that each edge has a non-negative capacity c(u,v)≥0. Flow in a network is an integer-valued function f defined on the edges of G satisfying 0≤f(u,v)≤c(u,v), for every edge (u,v) in E.

- Each edge (u,v) has a non-negative capacity c(u,v).
- If (u,v) is not in E assume c(u,v)=0.
- We have source s and sink t.
- Assume that every vertex v in V is on some path from s to t.
- For each edge (u,v) in E, the flow f(u,v) is a real valued function
- that must satisfy following three conditions :

Capacity Constraint :$\forall$ u,v $\in$V, f(u,v) $\leq$ c(u,v
Skew Symmetry : $\forall$ u,v $\in$V, f(u,v)= -f(v,u)
Flow Conservation: $\forall$ u $\in$V – {s,t} $\Sigma_{v \in V}$ f(s,v)=0

The Value of a Flow:
The value of a flow is given by $| f | = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t)$
The flow into the node is same as flow going out from the node and thus the flow is conserved. Also the total amount of flow from source s = total amount of flow into the sink t.
The Maximum Flow Problem
Given a Graph G (V,E) such that:
$x_{i,j}$ = flow on edge (i,j)
$u_{i,j}$= capacity of edge (i,j)
s = source node
t = sink node
Maximize v
Subject To $\Sigma_j x_{ij} - \Sigma_j x_{ji} = 0$ for each i $\neq$s,t
$\Sigma_j x_{sj} = v$
$0 \leq x_{ij} \leq u_{ij}$ for all (i,j) $\in$ E.

The Ford–Fulkerson Algorithm

The **Ford–Fulkerson algorithm** is an elegant solution to the maximum flow problem. Fundamen-tally, it works like this:

The Ford-Fulkerson's Algorithm
Ford-Fulkerson (G,s,t)

1. For each edge e $\in$ E

2. initialize f(e) ← 0
3. G_f ← residual graph
4. WHILE (there exists augmenting path P)
5. f ← augment(f, P)
6. update G_f
7. RETURN f

Residual Networks and Augmenting Paths

The Ford–Fulkerson algorithm begins with a flow f (initially the zero flow) and successively improves
f by pushing more water along some path p from s to t. Thus, given the current flow f , we need

Augmenting Paths:

Definition:

An augmenting path p is a simple path from s to t on a residual network that is an alternating sequence of vertices and edges of the form $s,e_1,v_1,e_2,v_2,...,e_k,t$ in which no vertex is repeated and no forward edge is saturated and no backward edge is free.

Characteristics of augmenting paths

More flow from s to t through p is possible. The edges of residual network are the edges on which residual capacity is positive. The maximum capacity by which we can increase the flow on p the residual capacity of p is defined as $c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$.

Illustration:

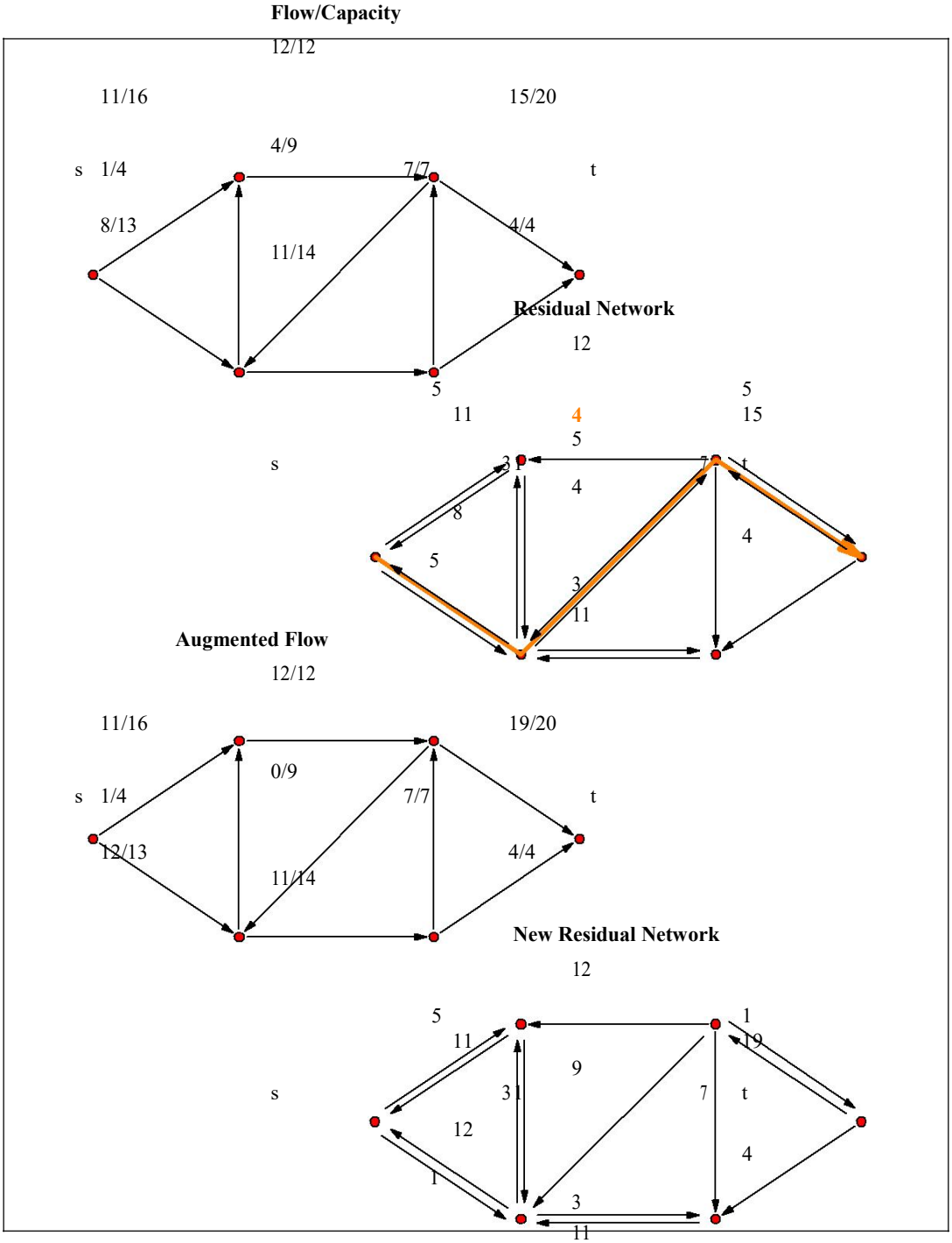


Figure 3.2. We begin with a flow network G and a flow f : the label of an edge (u, v) is "a/b," where a = f (u, v) is the flow through the edge and b = c(u, v) is the capacity of the edge. Next, we highlight an augmenting path p of capacity 4 in the residual network G_f. Next, we augment f by the augmenting path p. Finally, we obtain a new residual network in which there happen to be no more augmenting paths. Thus, our new flow is a maximum flow.

The Max Flow–Min Cut Equivalence

Definition. A **cut** (S, T Æ V \ S) of a flow network G is just like a cut (S, T) of the graph G except that we require s ∈S and t ∈T. Thus, any path from s to t must cross the cut (S, T). Given a flow f in G, the **net flow** f (S, T) across the cut (S, T) is defined as

Cuts of Flow Networks:

A Cut in a network is a partition of V into S and T (T=V-S) such that s (source) is in S and t (target) is in T and Capacity of Cut (S,T) is defined as $c(S,T) = \sum_{u \in S, v \in T} c(u,v)$

\

Theorem: (Max Flow–Min Cut Equivalence)**.** Given a flow network G and a flow f , the following are equivalent:

(i) f is a maximum flow in G.

(ii) The residual network G_f contains no augmenting paths.

(iii) jf j = c(S, T) for some cut (S, T) of G.

If one (and therefore all) of the above conditions hold, then (S, T) is a minimum cut.

MCQ:

1. Which one is called capacity constrain
   a. ∀ u,v ∈V, f(u,v) ≤ c(u,v
   b. ∀ u,v ∈V, f(u,v)= -f(v,u)
   c. ∀ u ∈V – {s,t} Σ f(s,v)=0
   d. None of these
2. Which one is called skew symmetry constrain
   e. ∀ u,v ∈V, f(u,v) ≤ c(u,v
   f. ∀ u,v ∈V, f(u,v)= -f(v,u)
   g. ∀ u ∈V – {s,t} Σ f(s,v)=0
   h. None of these
3. Which one is called flow constrain
   i. ∀ u,v ∈V, f(u,v) ≤ c(u,v
   j. ∀ u,v ∈V, f(u,v)= -f(v,u)
   k. ∀ u ∈V – {s,t} Σ f(s,v)=0
   l. None of these

Question:

1. What is Network Flow ? What do mean by residual capacity? Explain the Ford-Fulkerson Algorithm. Define CUT(S,T). Explain augmenting path and augmenting network.

MODULE 03: Disjoint set manipulation: [2L]

When implementing Kruskal's algorithm i, we built up a minimum spanning tree T by adding in one edge at a time. Along the way, we needed to keep track of the connected components of T; this was achieved using a disjoint-set data structure. In this lecture we explore disjoint-set data structures in more detail.

A **disjoint-set data structure** is a data structure representing a dynamic collection of sets **S** = $\{S_1,..., S_r\}$. Given an element u, we denote by S_u the set containing u. We will equip each set S_i with a representative element rep. This way, checking whether two elements u and v are in the same set amounts to checking whether $rep[S_u] = rep[S_v]$. The disjoint-set data structure supports the following operations:

MAKE-SET(u): Creates a new set containing the single element u.

– u must not belong to any already existing set

– of course, u will be the representative element initially

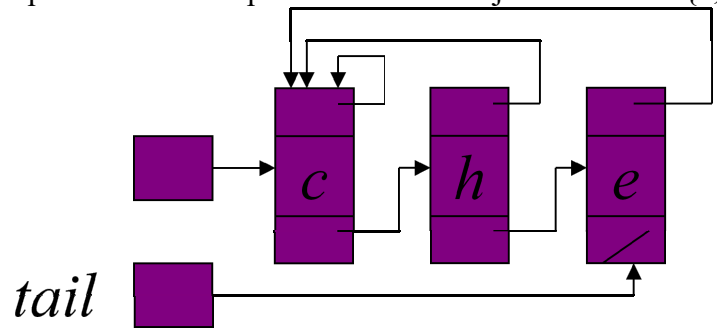
FIND-SET(u): Returns the representative $rep[S_u]$.

UNION(u, v): Replaces S_u and S_v with S_u [S_v in **S**. Updates representative elements as appropriate.

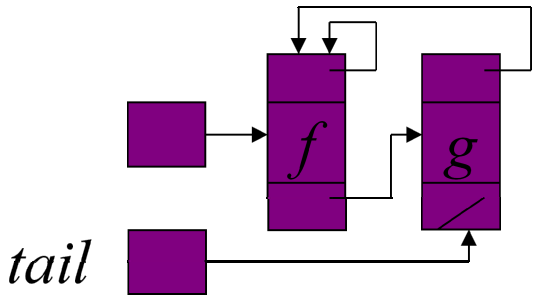
In Lecture 4, we looked at two different implementations of disjoint sets: doubly-linked lists and trees. In this lecture we'll improve each of these two implementations, ultimately obtaining a very efficient tree-based solution.

Linked-List Implementation

Linked-lists for two sets are as follows:
Simple linked-list implementation of disjoint sets : Set {*c*,*h*,*e*}



Another linked-list implementation of disjoint sets : Set {*f*, *g*}



MAKE SET(S,u) –	initialize as a lone node	O(1)	
FIND-SET(S, u) –	walk left from u until you reach the head of S_u	O(n)	worst-case
UNION(S, u, v) –	walk right (towards the tail) from u and left (towards the head) from v. Reassign pointers so that the tail of S_u and the head of S_v become neighbors. The representative is updated automatically.	O(n)	worst-case

An Application of Disjoint-Set
Algorithms ddetermine the connected components of an undirected graph.

CONNECTED-COMPONENTS(G)
1. **for** each vertex $v \in$V[G]
2. **do** MAKE-SET(v)
3. **for** each edge $(u,v) \in$E[G]
4. **do if** FIND-SET(u) ≠ FIND-SET(v)
5. **then** UNION(u,v)

SAME-COMPONENT(u,v)
1. **if** FIND-SET(u)=FIND-SET(v)
2. **then return** TRUE
3. **else return** FALSE

Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer

UNION Implementation
- A simple implementation: UNION(x,y) just appends x to the end of y, updates all back-to-representative pointers in x to the head of y.
- Each UNION takes time linear in the x's length.
- Suppose n MAKE-SET(x_i) operations (O(1) each) followed by n-1 UNION
  - UNION(x_1, x_2), O(1),
  - UNION(x_2, x_3), O(2),
  - …..
  - UNION(x_{n-1}, x_n), O(n-1)
- The UNIONs cost 1+2+…+n-1=$\Theta(n^2)$
- So 2n-1 operations cost $\Theta(n^2)$, average $\Theta(n)$ each.

MAKE-SET(u) –	initialize new tree with root node u	$\Theta(1)$
FIND-SET(u) –	walk up tree from u to root	$\Theta(\text{height}) = \Theta(\lg n)$ best-case
UNION(u, v) –	change rep[S_v]'s parent to rep[S_u]	O(1)

The efficiency of the basic implementation hinges completely on the height of the tree: the shorter the tree, the more efficient the operations. As the implementation currently stands, the trees could

However the following program segment to perform the following set operations.

MAKE-SET(u):

```
void initialize( int Arr[ ], int N)
{
  for(int i = 0;i<N;i++)
  Arr[ i ] = i ;
}
```

FIND-SET(u) :

```
bool find( int Arr[ ], int A, int B)
{
if(Arr[ A ] == Arr[ B ])
return true;
else
return false;
}
```

UNION(u, v) :

```
void union(int Arr[ ], int N, int A, int B)
{
  int TEMP = Arr[ A ];
for(int i = 0; i < N;i++)
  {
  if(Arr[ i ] == TEMP)
  Arr[ i ] = Arr[ B ];
  }
}
```

Weighted-Union Heuristic

Instead appending *x* to *y*, appending the shorter list to the longer list.
Associated a length with each list, which indicates how many elements in the list.

Result: a sequence of m MAKE-SET, UNION, FIND-SET operations, n of which are MAKE-SET operations, the running time is $O(m+n\lg n)$.
Count the number of updates to back-to-representative pointer for any x in a set of n elements. Consider that each time, the UNION will at least double the length of united set, it will take at most lg n UNIONS to unite n elements. So each x's back-to-representative pointer can be updated at most lg n times.

Union by rank

Union by Rank: Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.

Path compression

Path Compression: used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.
The easiest kind of tree to walk up is a flat tree, where all non-root nodes are direct children of the root. The idea of path compression is that, every time we invoke FIND-SET and walk up the tree, we should reassign parent pointers to make each node we pass a direct child of the root .This locally flattens the tree. With path compression, the pseudocode for FIND-SET is as follows:

Path Compression example

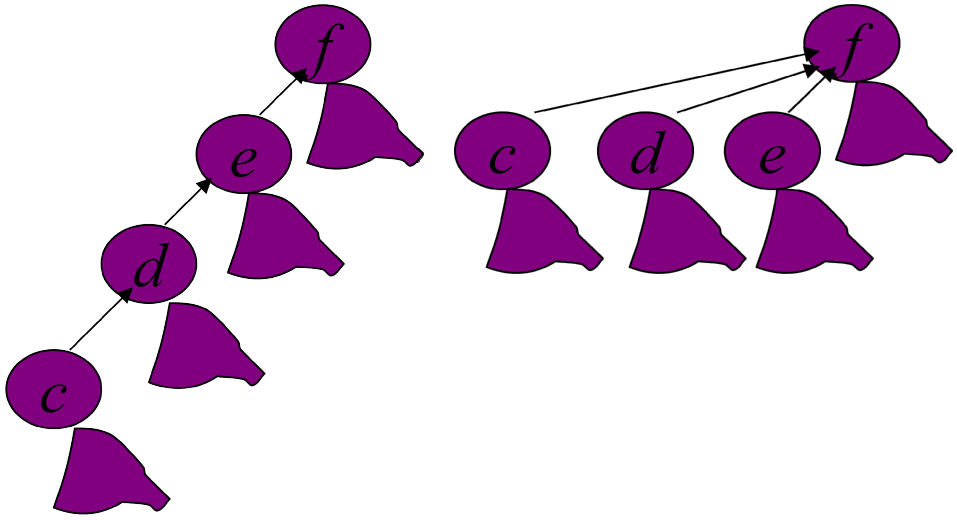


Figure.3.3 Path Compression

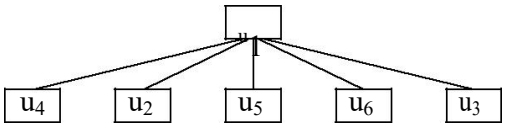


Figure 3.4 In a flat tree, each FIND-SET operation requires us to traverse only one edge.

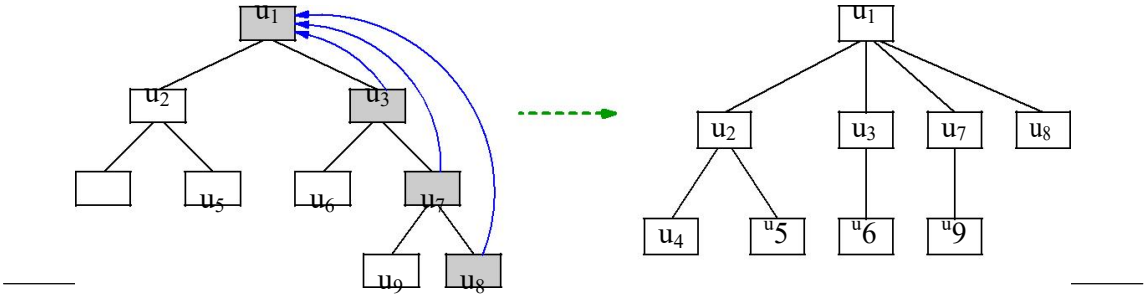


Figure 3.5. With path compression, calling FIND-SET (u_8) will have the side-effect of making u_8 and all of its ancestors direct children of the root.

MCQ:

1. Which one is complexity of find-set
   a. O(1)
   b. O(N)
   c. O(log(N))
   d. None of these
2. Which one is complexity of make-set
   e. O(1)
   f. O(N)
   g. O(log(N))
   h. None of these
3. Which one is complexity of union-set
   i. O(1)
   j. O(N)
   k. O(log(N))
   l. None of these

Question:

1. What is disjoint set ? Explain with example.
2. What do mean by union of two disjoint set?
3. Explain Algorithm MAKE-SET, FIND-SET and UNION-SET. Explain union by rank and path compression

MODULE 03: Lower Bound Theory: [1L]

In order to determine how good a given algorithm is for solving a problem, it is useful if we know lower bounds for the complexity of ANY algorithm solving the problem. Finding good lower bounds is a difficult problem, in general. However, in the important examples of sorting using comparison-based algorithms, and finding the maximum (minimum) or both the maximum and the minimum values in a list, lower bounds can be found, as well as algorithms achieving (up to positive multiplicative constants, i.e., asymptotically) these lower bounds (i.e., optimal algorithms). For many other important problems, for example, for the NP-complete problems (of which there are thousands of examples), good lower bounds have not been found. For example, for any given NP-complete problem, we only know polynomial lower bounds typically with low degree, whereas the best known algorithms for solving the problem are super-polynomial in complexity. Thus, the famous P $\neq$ NP remains an open question despite over 30 years of investigation by the best minds in theoretical computer science.

We discuss three important methods of establishing lower bounds for algorithm complexity, counting arguments, comparison trees, and adversary arguments. We begin with counting arguments, and a review of lower bounds for adjacent-key comparison-based sorting.

Adjacent-Key Comparison-Based Sorting

Recall that BubbleSort, InsertionSort, and SelectionSort are all examples of ADJACENT-KEY comparison-based sorting algorithms. Also recall that adjacent-key comparison sorts only remove one inversion per comparison. Using this, we saw that a lower bound for the worst case of adjacent-key comparison sorting algorithms is n(n-1)/2, whereas a lower bound for the average behavior of such algorithms is n(n-1)/4. Thus, we must look for sorting algorithms that sometimes compare nonadjacent list elements in order to achieve better than quadratic performance on average. Our old friends ShellSort, MergeSort and QuickSort are examples of such sorting algorithms. We also saw that a lower bound for the worst case of any comparison-based sorting algorithm is $\log_2 n! \in \Omega(n \log n)$.

MergeSort and HeapSort are examples of sorting algorithms whose worst-case performance is O(nlog n), so that they exhibit optimal worst-case behavor.

An alternate proof of the nlog n lower bound for comparison-based sorting can be given using comparison trees, which we now will introduce for the purpose of establishing lower bounds for the average behavior of comparison-based sorting algorithms.

Comparison Trees

We now show that MergeSort is also optimal on average, since nlog n is also a lower bound (again, up to a constant) for the average behavior of comparison-based sorting. This latter result will be established using a comparison tree argument. Given any comparison-based algorithm with input list $L[0:n-1] = \{x_1, x_2, ..., x_n\}$, (internal) nodes in the *comparison tree* T associated with the algorithm correspond to comparisons performed by the algorithm between list elements. For specificity, our convention will be that if the comparison is made between x_i and x_j, and $i < j$, then we will label the corresponding node $x_i:x_j$. If $x_i < x_j$, then a left child will be the node corresponding to the next comparison made next by the algorithm, or this left child will be a leaf node if the algorithm terminates. Similarly, if $x_i > x_j$ (we can assume distinct list elements for the

purpose of establishing lower bounds), then the right child will be the node corresponding to the next comparison made by the algorithm, or will be a leaf node if the algorithm terminates.

NOTE: our labeling of the nodes refers to the list elements, not to their positions at the time the comparison corresponding to a given node is made.

The following figure illustrates the comparison-tree associated with *InsertionSort* for a list L[0:2] of 3 distinct elements x_1, x_2, x_3.

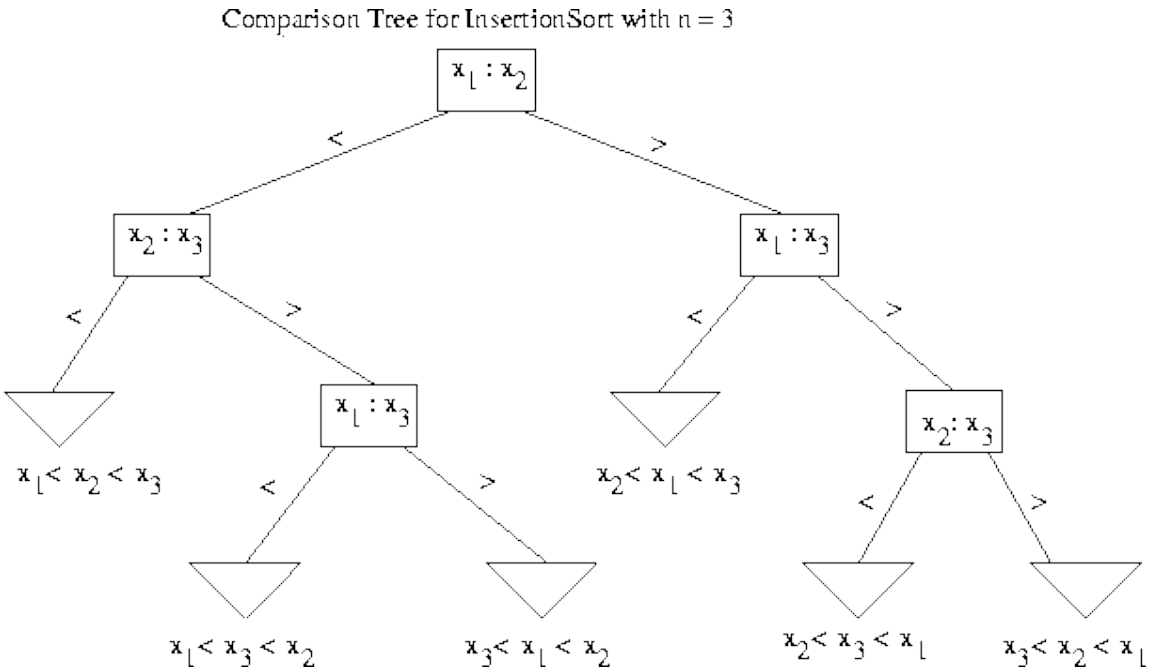


Figure 3.6. The comparson tree associated with any comparison-based sorting algorithm has n! leaf nodes.

The Key Fact follows from the fact that there are n! factorial permutations of n symbols, and different permutations must end up at different leaf nodes of the comparison tree when input to the algorithm. Since the comparison tree associated with a comparison-based sorting algorithm is a binary tree, lower bounds for both worst-case and average complexity can be obtained from lower bounds for the depth and *leaf path length* (= sum of the lengths of all paths from the root to a leaf), respectively, of a binary tree having L leaf nodes.

Proposition 1. Let T be any binary tree with L leaf nodes. Then

Depth(T)≥ ceil($\log_2$L)

Proposition 1 is clearly true for complete binary trees (verify this!), so it is intuitively evident that it holds for arbitrary binary trees since the complete binary tree has the smallest depth for a given number L of leaf nodes. The formal proof of Proposition 1 can be found on p. 120 in the text. It follows immediately from Proposition 1 that

$W(n) \geq \text{ceil}(\log_2 L)$

for any comparison-based sorting algorithm. Now, ceil($\log_2 L$) = ceil($\log_2 n!$) $\in \Omega$(nlog n), so that we have established another proof of the fact that nlog n is a lower bound for the worst-case complexity of comparison-based sorting.

The following Proposition will give us a lower bound of nlog n for the average case as well.

Proposition 2. Let T be any binary tree having L leaf nodes. Then the leaf path length LPL of T satisfies:

$LPT(T) \geq Lfloor(\log_2 L) \in \Omega(L\log L)$

Again, Proposition 2 is clearly true for complete binary trees (verify this!), so it is evidently true for arbitrary trees. A formal proof of Proposition 2 can be found on p. 124 in the text. Now if T is the comparison tree associated with any comparison-based sorting algorithm, we see that A(n) = LPT(T)/L, so that Proposition 2 shows that nlog n is a lower bound for the average behavior of any comparison-based algorithm. Again, our old friends MergeSort, QuickSort, and TreeSort are all optimal average behavior comparison-based sorting algorithms.

We now illustrate our third technique for establishing lower bounds, namely adversary arguments. This technique establishes lower bounds by creating an input instance, based on the performance of the algorithm, which guarantees that the algorithm must do a determined amount of work on this input in order to be correct for this input. This amount of work then gives a lower bound for the worst-case complexity of the algorithm. Another adversary-type technique is to construct an input to an algorithm which contradicts the correctness of the algorithm if the algorithm performs less than some given number of basic operations. We start with an example of this type of adversary argument.

Finding the Maximum in a List

The usual linear scan for finding the maximum element in a list L[0:n-1] of size n turns out to be optimal, since ANY comparison-based algorithm for solving this problem must make n - 1 comparsions between list elements. This might seem obvious, since certainly every element must participate in at least one comparison. However, only n/2 comparisons are required to ensure that each element is involved in a comparison. Just pair the elements up into disjoint pairs a make a comparison to the two elements in each pair. Of course, this doesn't yet determine the maximum, but it shows the need for further justification that n - 1 comparisons will eventually be required. Again, throughout we will assume distinct list elements.

To get the lower bound of n - 1 comparisons, we consider a comparison between list elements x and y to declare as the *loser* the smaller of the two elements. Thus, each comparison results in exactly one loser. We now note that there must be n - 1 losers if the algorithm is to determine the maximum element in a list L[0:n-1] correctly. Indeed, assume that there are two elements x and y who never lost a comparison, and, for definiteness, assume that x > y. Now we may suppose that the algorithm has declared that x is the maximum element (otherwise it is clearly incorrect). Now construct a new list L' which agrees with L except that y is replaced by an element y' > x > y. Note that the algorithm will perform exactly the same action on L' as it did with L, since y' will win every comparison that involved y (and the outcome of all the other comparisons will also be the same). Hence, the algorithm will again declare x to be the maximum element, which is a contradiction. We state this result as a proposition.

Proposition 3. Any comparison-based algorithm must make (at least) n - 1 comparisons of list elements in order to correctly determine the maximum.

Proposition 3 shows that the familiar linear scan algorithm for finding the maximum is an (exactly) optimal algorithm. It is interesting that there is another algorithm also performing n - 1 comparisons to find the maximum, but this time it is based on the familiar single elimination tournament model so familiar from the sporting world. For simplicity, we assume that $n = 2^k$. Divide up the list into disjoint pairs, and determine the n/2 pair-wise winners (1st round of the tournament). Then divide up the n/2 first-round winners into pairs and determine the n/4 second round winners. After precisely $\log_2 n$ rounds the winner (maximum) will be determined. But how many comparisons (matches) were made? Easy, we get, for $n = 2^k$:

$2^{k-1} + 2^{k-2} + ... + 1 = 2^k - 1 = n - 1.$

Finding the Maximum and the Minimum

The most naive method MaxMin1 for finding the maximum and minimum elements in a list is to make two linear scans (or run winner and loser tournaments), resulting in 2n - 2 comparisons. However, one imagines that this can be improved, since information about both elements involved in a comparison might be utilized. In fact, the following slightly less naive algorithm certainly improves MaxMin1, at least on average.

```
function MaxMin2(L[0:n-1])
Input:  L[0:n-1] (a list of size n)
Output: the maximum value in L
  Max = Min = L[0]
  for i = 1 to n-1 do
     if L[i] > Max then Max = L[i]
     else
        if L[i] < Min then Min = L[i]
        endif
     endif
  enfor
end MaxMin1
```

Note that in the best case of a strictly increasing list, MaxMin2 only makes n - 1 comparisons, whereas in the worst case W(n) of an decreasing list, MaxMin2 makes 2n - 2 comparisons, i.e., is just as bad as MaxMin1. The question is, how good is MaxMin2 on average? Well, it turns out that it is disappointing, since its average behavior, while improved slightly over W(n), is nevertheless strongly asymptotic to W(n). This is because the average number of times that Max is updated in MaxMin2 is (guess what!) logarithmic in n, so that the average complexity A(n) of MaxMin2 is of the form $A(n) = W(n) - f(n)$, where $f(n) \in O(\log n)$, i.e., $A(n) \in \Theta(W(n))$ (actually, $A(n) \sim W(n)$).

In order to determine the average behavior A(n) of MaxMin2, we assume, as usual, that the inputs are all permutations $\pi: \{1,2, ..., n\} \rightarrow \{1,2, ..., n\}$, and that each permutation is equally likely. Now if $m(\pi)$ denotes the the number of times Max is updated for input permutation π, then it is clear that

$A(n) = 2n - 2 - E[m].$

Let A*(n) = E[m]. Note that π(n) is equally likely to be 1, 2, ..., n. Hence,

A*(n) = 1/n(E[m| π(n) = 1] + E[m| π(n) = 2] + ... + E[m| π(n) = n]).

Now it is clear that E[m| π(n) = n] = A*(n-1) + 1, whereas E[m| π(n) = i ≠ n] = A*(n-1). Hence, we have the following recurrence for A*(n):

A*(n) = A*(n - 1) + 1/n
= A*(n - 2) + 1/n + 1/(n - 1)
=
...
= A*(1) + 1/n + 1/(n - 1) + ... + 1/2

= ~ ln n.

Thus, we see that A(n) ~ W(n) ~ 2n.

Lower bound for comparison based sorting algorithms

The problem of sorting can be viewed as following.

Input: A sequence of n numbers $<a_1, a_2, . . . , a_n>$.
Output: A permutation (reordering) $<a'_1, a'_2, . . . , a'_n>$ of the input sequence such that $a'_1 <= a'_2 ….. <= a'_n$.

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i <= a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i <= a_j$, and the right subtree dictates subsequent comparisons for $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering. So we can say following about the decison tree.

1) Each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

2) Let x be the maximum number of comparisons in a sorting algorithm. The maximum height of the decison tree would be x. A tree with maximum height x has at most 2^x leaves.

After combining the above two facts, we get following relation.

$n! <= 2^x$

Taking Log on both sides.
$\log_2(n!) <= x$

Since $\log_2(n!) = \Theta(n\text{Log}n)$, we can say
$x = \Omega(n\text{Log}_2 n)$

Therefore, any comparison based sorting algorithm must make at least $nLog_2n$ comparisons to sort the input array, and Heapsort and merge sort are asymptotically optimal comparison sorts.

References:

Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

MCQ:

1. If two algorithm for solving the problem where discovered and their times differed by an order of magnitude, the one with the smaller order was generally regarded as superior

   a. Comparison trees
   b. Oracle and adversary argument
   c. Lower bound through reductions
   d. All of the above

2. Any comparison-based sorting algorithm can be represented by a decision tree

a. Number of leaves (outcomes) $\geq n!$

b. Height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$

c. Minimum number of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$ for any comparison-based sorting algorithm $\lceil \log_2 n! \rceil \approx n \log_2 n$

d. *All of the above*

3. Lower bounds by problem reduction represented
   a. $\Omega(n\log n)) \geq n!$

   b. $\Omega(n\log n) \geq \lceil \log_2 n! \rceil$

   c. $\Omega(n\log n) = \lceil n\log_2 n! \rceil$

   d. *$\Omega(n\log n)$*

Question:

1. What is meant by lower bound theory?
2. What is the lower bound of a function?
3. In algorithms, what is the upper and lower bound?
4. What is the difference between lower bound and tightest upper bound?
5. What are some good materials to learn the complexity bound (upper and lower bound) for convex optimization?
6. How is it possible to prove lower bounds in complexity theory?
7. How can you solve upper and lower bounds in extended mathematics?
8. What conspiracy theories turned out to be true?

MODULE 04: String Matching Algorithms :[3L]

A string matching algorithm aims to find one or several occurrences of a string within another. The algorithm returns the position of the first character of the desired substring in the text. There are many different solutions for this problem, this article presents the four best-known string matching algorithms: Naive, Knuth-Morris-Pratt, Boyer-Moore and Rabin-Karp. The results show that Boyce-Moore is the most effective algorithm to solve the string matching problem in usual cases, and Rabin-Karp is a good alternative for some specific cases, for example when the pattern and the alphabet are very small.

There are different solutions that allow to solve the string matching problem. First, we have The naive algorithm, the simplest one, which tries to match the pattern to each string of the same length in the text. From the 1970s, several others algorithms, more sophisticated and more effective, have been invented. In 1975, Knuth, Pratt and Morris invented the first algorithm that preprocesses the pattern to obtain a better performance, it is the
Knuth-Morris-Pratt Algorithm
.

String processing problem
Input: Two strings T and P.
Problem: Find if P is a substring of T
Example (1):
Input: T = gtgatcagatcact, P = tca
Output: Yes. gtga**tca**ga**tca**ct, shift=4, 9
Example (2):
Input: T = 189342670893, P = 1673
Output: No.

Naïve Algorithm (T, P)

```
suppose n = length(T), m = length(P);
for shift s=0 through n-m do
        if  (P[1..m] = = T[s+1 .. s+m])  then  // actually a for-loop runs here
                print shift s;
End algorithm.
```

Complexity: O((n-m+1)m)
A special note: we allow O(k+1) type notation in order to avoid O(0) term, rather, we want to have O(1) (constant time) in such a boundary situation.

Note: Too many repetition of matching of characters.

Rabin-Karp scheme

Consider a character as a number in a radix system, e.g., English alphabet as in radix-26.
Pick up each m-length "number" starting from shift=0 through (n-m).
So, T = gtgatcagatcact, in radix-4 (a/0, t/1, g/2, c/3) becomes
gtg = '212' in base-4 = 32+4+2 in decimal,
tga = '120' in base-4 = 16+8+0 in decimal,
….
Then do the comparison with P - number-wise.

Advantage: Calculating strings can reuse old results.
Consider decimals: 4359 and 3592
3592 = (4359 - 4*1000)*10 + 2
General formula: $t_{s+1} = d\ (t_s - d^{m-1}\ T[s+1])\ +\ T[s+m+1]$, in radix-d, where t_s is the corresponding number for the substring T[s..(s+m)]. Note, m is the size of P.

The first-pass scheme: (1) preprocess for (n-m) numbers on T and 1 for P, (2) compare the number for P with those computed on T.

Problem: in case each number is too large for comparison
Solution: *Hash*, use modular arithmetic, with respect to a prime q.

New recurrence formula:
$t_{s+1} = (d\ (t_s - h\ T[s+1])\ +\ T[s+m+1])\ \text{mod}\ q$,
where $h = d^{m-1}\ \text{mod}\ q$.
q is a prime number so that we do not get a 0 in the mod operation.

Now, the comparison is not perfect, may have spurious hit (see example below).
So, we need a naïve string matching when the comparison succeeds in modulo math.

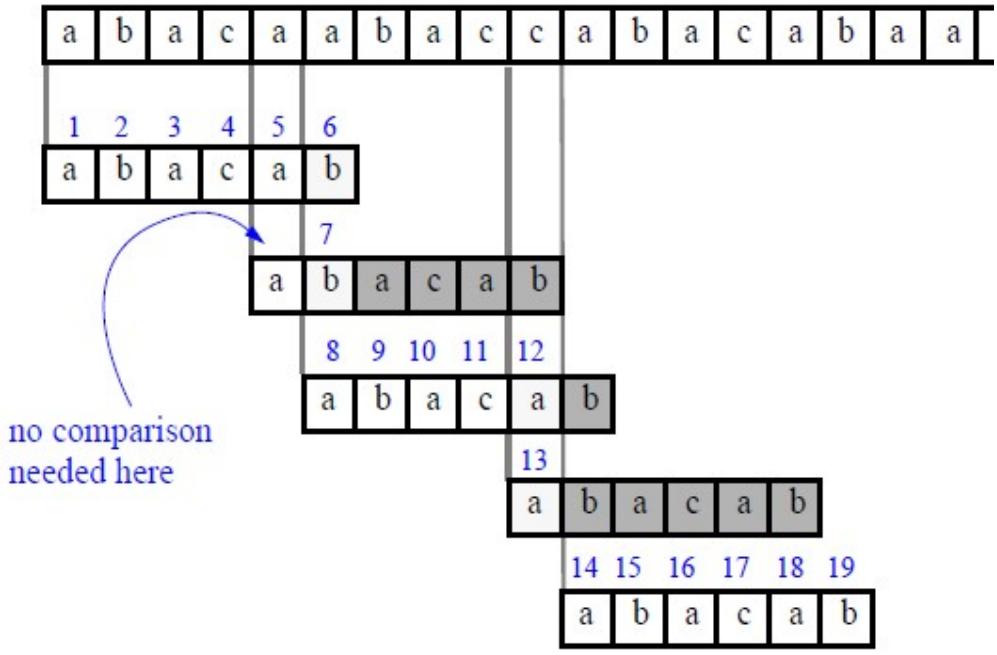


Figure 4.1. A graphical representation of the KMP string searching algorithm

Rabin-Karp Algorithm:
Input: Text string T, Pattern string to search for P, radix to be used d (= |∑|, for alphabet ∑), a prime q
Output: Each index over T where P is found

Rabin-Karp-Matcher (T, P, d, q)
n = length(T); m = length(P);
h = d^{m-1} mod q;
p = 0; t_0 = 0;
for i = 1 through m do // Preprocessing
    p = (d*p + P[i]) mod q;
    t_0 = (d* t_0 + T[i]) mod q;
end for;
for s = 0 through (n-m) do // Matching
    if (p = = t_s) then
        if (P[1..m] = = T[s+1 .. s+m]) then
            print the shift value as s;
    if (s < n-m) then
        t_{s+1} = (d (t_s - h*T[s+1]) + T[s+m+1]) mod q;
end for;
End algorithm.

Complexity:

Preprocessing: O(m)

Matching:
O(n-m+1)+ O(m) = O(n), considering each number matching is constant time.

However, if the translated numbers are large (i.e., m is large), then even the number matching could be O(m). In that case, the complexity for the worst case scenario is when every shift is successful ("valid shift"), e.g., T=a^n and P=a^m. For that case, the complexity is O(nm) as before.
But actually, for c hits, O((n-m+1) + cm) = O(n+m), for a small c, as is expected in the real life.

THIRD ALGORITHM USING AUTOMATON
(Efficient with less alphabet |∑|)
Finite Automaton: (Q, q_0, A, ∑, d), where
Q is a finite set of states, q_0 is one of them - the start state, some states in Q are 'accept' states (A) for accepting the input, input is formed out of the alphabet ∑, and d is a binary function mapping a state and a character to a state (same or different).
Matcher scheme: (1) Pre-processing: Build an automaton for the pattern P, (2) Matching: run the text on the automaton for finding any match (transition to accept state).
Example automaton for 'ababaca' :

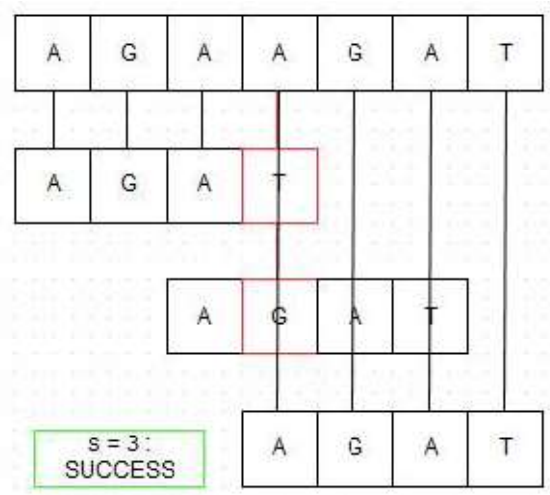


Figure 4.2. Example of operation of the KMP string matcher in a DNA string

Algorithm FA-Matcher (T, d, m)
n = length(T); q = 0; // '0' is the start state here
// m is the length(P), and
// also the 'accept' state's number
for i = 1 through n do
q = d (q, T[i]);
if (q = = m) then
print (i-m) as the shift;
end for;
End algorithm.

Complexity: O(n)
However, we need to build the finite-state automaton for P first:
Input: $\sum$, and P
Output: The transition table for the automaton
Algorithm Compute-Transition-Function(P, $\sum$)
m = length(P);
for q = 0 through m do
for each character x in $\sum$
k = min(m+1, q+2); // *+1 for x, +2 for subsequent repeat loop to decrement*
repeat k = k-1 // *work backwards from q+1*
until P_k 'is-suffix-of' $P_q x$;
d(q, x) = k; // assign transition table
end for; end for;

return d;
End algorithm.

Examples (from the above figure P = 'ababaca'):
Suppose, q=5, x=c
P_q = ababa, $P_q x$ = ababac,
P_k (that is suffix of $P_q x$) = ababac, for k=6 (note transition in the above figure)

Say, q=5, x=b

P_q = ababa, P_qx = abababb,
P_6 = ababac ≠ suffix of P_qx
P_5 = ababa ≠ suffix of P_qx , but
P_k (that is suffix of P_qx) = abab, for k=4

Say, q=5, x=a
P_q = ababa, P_qx = ababaa,
P_k (that is suffix of P_qx) = a, for k=1

Complexity of the above automaton-building (preprocessing):
Outer loops: m|∑|
Repeat loop (worst case): m
Suffix checking (worst case): m
Total: $O(m^3|\sum|)$

Good, when you build automaton once, search many times.
Bad, when you have build automata for different P many times, or #searches/#keys ratio is low.

Knuth-Morris-Pratt Algorith

We do not need the whole *transition table* as in an automaton.
An array can keep track of: for each *prefix-sub-string* S of P, what is its largest *prefix-sub-string* K of S (or of P), such that K is also a *suffix* of S (kind of a symmetry within P).

Symmetry: prefix = suffix

Thus, P=abababababca, when S=P6=ababab, largest K is abab, or Pi(6)=4.

An array Pi[1..m] is first developed for the whole set for S, Pi[1] through Pi[10] above.

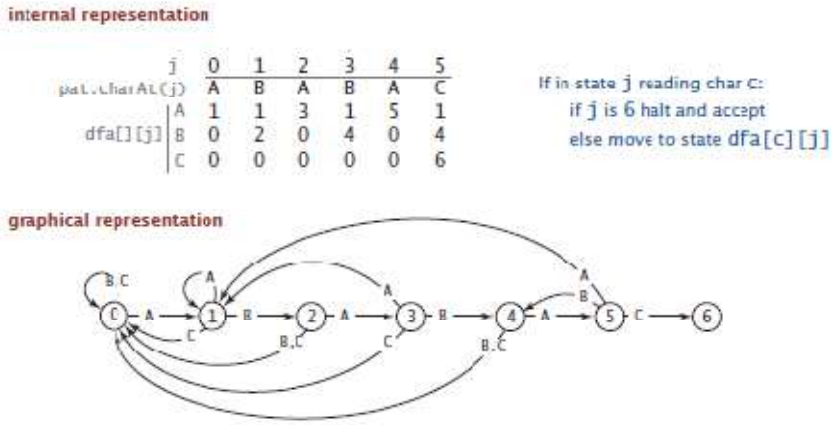


Figure 4.3. Example of operation of the KMP string matcher

The array Pi actually holds a chain for transitions, e.g., Pi[8] = 6, Pi[6]=4, …,
always ending with 0.

Algorithm KMP-Matcher(T, P)
n = length[T]; m = length[P];
Pi = Compute-Prefix-Function(P);

```
q = 0;              // how much of P has matched so far, or could match possibly

for  i=1 through n do
        while  (q>0  &&  P[q+1] ≠ T[i])  do
                q = Pi[q];        // follow the Pi-chain, to find next smaller available symmetry, until 0
        if  (P[q+1] = = T[i])  then
                q = q+1;
        if  (q = = m)  then
                print valid shift as (i-m);
                q = Pi[q];  // old matched part is preserved, & reused in the next iteration
        end if;
end for;
End algorithm.


Algorithm Compute-Prefix-Function (P)
m = length[P];
Pi[1] = 0;
k = 0;

for  q=2 through m do
        while  (k>0  &&  P[k+1] =/= P[q])  do  // loop breaks with k=0 or next if succeeding
                k = Pi[k];
        if   (P[k+1] = = P[q])   then // check if the next pointed character extends previously identified symmetry
                k = k+1;
        Pi[q] = k;  // k=0 or the next character matched

return Pi;
End algorithm.
```

Complexity of second algorithm Compute-Prefix-Function: O(m), by amortized analysis (on an average).
Complexity of the first, KMP-Matcher: O(n), by amortized analysis.

In reality the inner while loop runs only a few times as the symmetry may not be so prevalent. Without any symmetry the transition quickly jumps to q=0, e.g., P=acgt, every Pi value is 0!

Exercise:

For P= abababababca, run the Compute-prefix-function to develop the Pi array.
For T= cabacababababcabababababcac, run the KMP algorithm for searching P within T.
Show the traces of your work, not just the final results.

MCQ:

1. Brute Force-Complexity of sring/pattern of M characters in length, and a text N characters in length matching is
   a. O(N)

b. O(M)
c. O(MN)
d. O(M/N)

2. The Rabin-Karp string searching algorithm calculates a……. for the pattern, and for each M-character subsequence of text to be compared

a. Index value
b. Pattern value
c. Augmenting value
d. Hash value

3. Which one string matching algorithm use auxiliary function.

a. Naïve algorithm
b. Brute force
c. KMP algorithm
d. Robin-Karp

Question:

2. What is String matching problem ? What do mean by pattern? Explain the Brute-Force Algorithm. Define auxiliary function in KMP algorithm.
3. Explain The Knuth-Morris-Pratt (KMP) Algorithm. Derive the complexity.

References:

Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

MODULE 04: Amortized Analysis [3L]

In amortized analysis, the time required to perform a sequence of operations is averaged over all the operations performed

Data structures typically support several different types of operations, each with its own cost (e.g., time cost or space cost). The idea behind amortized analysis is that, even when expensive operations must be performed, it is often possible to get away with performing them rarely, so that the aver-age cost per operation is not so high. It is important to realize that these "average costs" are not expected values—there needn't be any random events.

Aggregate Analysis
Aggregate Method: we determine an upper bound *T(n)* on the total sequence of *n* operations. The cost of each will then be *T(n)/n*

(possibly with some condition on how many times each type of operation may occur). "Worst-case" means that no adversary could choose a sequence of n operations that gives a worse running time.

In this lecture we discuss three methods of amortized analysis: aggregate analysis, the account-ing method, and the potential method.

In **aggregate analysis**, one assumes that there is no need to distinguish between the different operations on the data structure. One simply asks, what is the cost of performing a sequence of n operations, of any (possibly mixed) types?

Example. Imagine a stack S with three operations:

PUSH(S, x) – Θ(1) – pushes object x onto the stack

POP(S) – Θ(1) – pops and returns the top object of S

- **Accounting Method:** we overcharge some operations early and use them to as prepaid charge later. Credit can be used later to pay the cost of operations whose actual cost is greater than its amortized cost. The total credit stored must always be non-negative at all times.

- **Potential Method:** we maintain credit as potential energy associated with the structure as a whole.

If we analyze a sequence of *n* Push, Pop, and Multipop operations, then the stack size is at most *n* and Multipop takes *O(n)*. Therefore, the sequence takes $O(n^2)$. However, since an object can be only popped once (whether by pop or multipop) for every time it is pushed, the total number of Push operations is at most the total number of Push operations (which is at most *n*).

Accounting Method: Stack

- Thing about this: when we push a plate onto a stack, we use $1 to pay actual cost of the push and we leave $1 on the plate.

- At any point, every plate on the stack has a dollar on top of it.
- When we execute a pop operation, we charge it nothing and pay its cost with the dollar that is on top of it.

Potential Method

- Instead of representing prepaid work as credit stored with specific objects, the potential method represents the prepaid work as potential energy than can be released to pay for future operations.
- Potential is associated with the data structure as a whole rather than with specific objects.
- We start with an initial data structure D_0 on which n operations are performed.
- Let c_i be the cost the ith operations and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}
- A potential function Φ maps D_i to a real number $\Phi(D_i)$ which is the potential associated with D_i

Dynamic Tables

- In some applications, we don't know in advance how many objects will be stored in a table.
- We must re-allocate with a larger or smaller size as data is added or removed.
- Define **load factor**
- When the table becomes full, i.e. load factor = 1, then the next operation triggers an expansion.
- We allocate a new area twice the size of the old and copy all items from the old table to the new table.
- If an operation does not trigger an expansion, its cost is 1. If it does trigger, its cost is *num [T] + 1.*

Example for amortized analysis

- Stack operations:
  - PUSH(S,x), $O(1)$
  - POP(S), $O(1)$
  - MULTIPOP(S,k), min(s,k)
    - **while** not STACK-EMPTY(S) and k>0
    - **do** POP(S)

- $k=k-1$

- Let us consider a sequence of n PUSH, POP, MULTIPOP.
  - The worst case cost for MULTIPOP in the sequence is $O(n)$, since the stack size is at most n.
  - thus the cost of the sequence is $O(n^2)$. Correct, but not tight.

Another example: increasing a binary counter

- Binary counter of length k, A[0..k-1] of bit array.
- INCREMENT(A)

1. iß0
2. **while** $i<k$ and A[i]=1
3. **do** A[i]ß0 (flip, reset)
4. ißi+1
5. **if** $i<k$
6. **then** A[i]ß1 (flip, set)

Accounting analysis

- Charge \$3 per insertion of x.
  - \$1 pays for x's insertion.
  - \$1 pays for x to be moved in the future.
  - \$1 pays for some other item to be moved.
- Suppose we've just expanded, $size = m$ before next expansion, $size = 2m$ after next expansion.
- Assume that the expansion used up all the credit, so that there's no credit stored after the expansion.
- Will expand again after another m insertions.
- Each insertion will put \$1 on one of the m items that were in the table just after expansion and will put \$1 on the item inserted.

Have \$2m of credit by next expansion, when there are 2m items to move. Just enough to pay for the expansion, with no credit.

Potential method

- $\forall$ $\Phi(T) = 2 \cdot num[T] - size[T]$

- Initially, $num = size = 0 \rightarrow \Phi = 0$.
- • Just after expansion, $size = 2*num \rightarrow \Phi = 0$.
- Just before expansion, $size = num \rightarrow = num \rightarrow \Phi = 0$ have enough potential to pay for moving all items.
- Need $\Phi \geq 0$, always.
- Always have
  - $size \geq num \geq \frac{1}{2}\, size \rightarrow 2 \cdot num \geq size\ \Phi\ F \geq 0$.

Amortized cost of ith operation:

- $num_i = num$ after ith operation ,
- $size_i = size$ after ith operation ,
- $\Phi_i = \Phi$ after ith operation .

- If no expansion:
  - $size_i = size_{i-1}$,
  - $num_i = num_{i-1} + 1$,
  - $ci = 1$.
- Then we have
  - $C_i' = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 3.$
- If expansion:
  - $size_i = 2size_{i-1}$,
  - $size_{i-1} = num_{i-1} = num_i - 1$,
  - $c_i = num_{i-1} + 1 = num_i.$
- $C_i' = c_i + \Phi_i - \Phi_{i-1} = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3$

MCQ:

1. Which data structures whose operations are analyzed using Amortized Analysis
   a. Hash Tables
   b. Disjoint Sets
   c. Splay Trees
   d. All of the above
2. What is the time complexity of n insertions using the dynamic table scheme?
   a. $O(n^2)$

b. $O(n)$
c. $\Theta(n)$
d. $\Theta(1)$

Questions:

1. What is amortize analysis? Explain with proper example
2. Define potential function.
3. Define accounting function.
4. Explain aggregate method.
5. What is stack implementation of array of problem?
6. How to compare an experimental study with amortized times
7. Define disjoint set and analyse the union-find problem.
8. Define splay tree. Why is maximum size of root is $2n + 1$ in Splay trees?
9. Define live and dead node

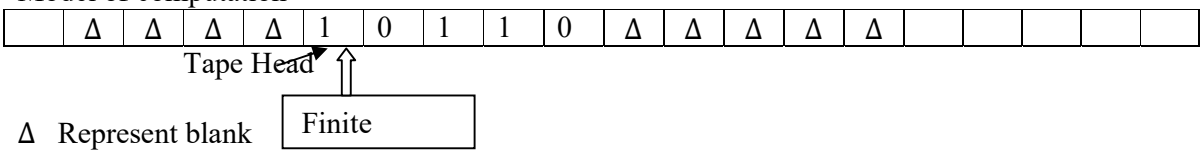
References:

Introduction to Algorithms, (Cormen, Leiserson, Riveset, and Stein, 2001, ISBN: 0-07-013151-1 (McGraw Hill), Chapter 32, p906

MODULE 05: Notion of NP-completeness: [3L]

Turing Machine(TM)

Model of computation



Δ Represent blank

The TM performs the following in one move.

1) Changes state(may remain same state)
2) Replaces the symbol on tape cell by a new symbol(may be by same symbol)
3) Move tape head to left or to right by one cell or remain stationary (denoted b L,R,S respectively).
4) Generally the initial state of the machine is denoted by q_0.

Example:Construct a TM that creates a copy of the input string (∑={a,b}) , to the right of the input but with a blank (Δ) separating the original from the copy. Initially tape head is assumed to be placed at the leftmost end of the string. [∑= set of input symbols]

Initially ➔ Δabba Δ

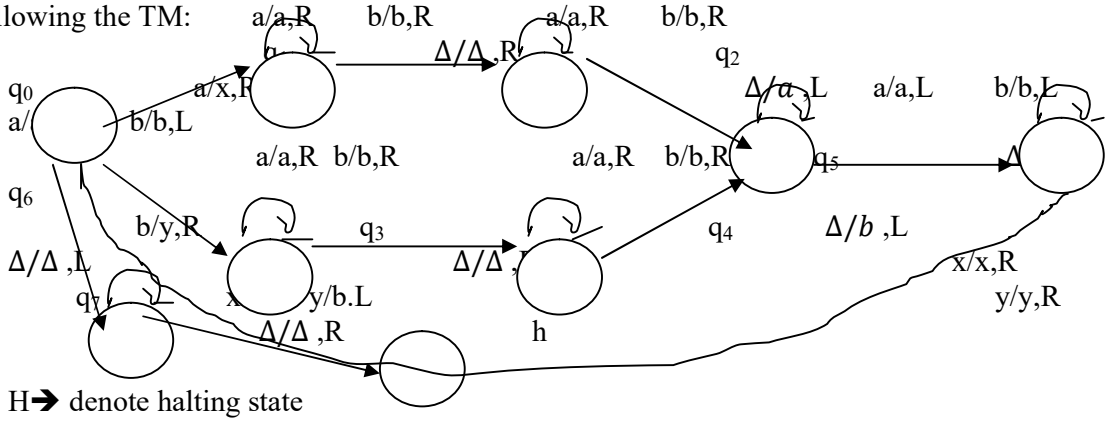
Δ xbbaΔaΔ ↓ sequence of action by TM

Δxyba$\Delta ab\Delta$

Δxyya$\Delta abb\Delta$

Δxyyx$\Delta abba\Delta$

Following the TM:



H➔ denote halting state

Meaning of transition ➔ X/Y,D

X Denotes i/p symbol

Y Denotes o/p symbol

D direction of head movement<u>Formed definition of TM:</u>

A TM is a 5 tuple(Q,∑,Γ,∂,q_0)

Where,

Q= A finite set of states not including the halting state h_a , h_r

∑= A finite set of input symbols.

Γ= A finite set of tape symbols not including the special tape symbol Δ(blank) ∑ <= Γ

∂=Q x Γ ➔(Q U {h}) X(ΓU {Δ}) X({L ,R ,S})

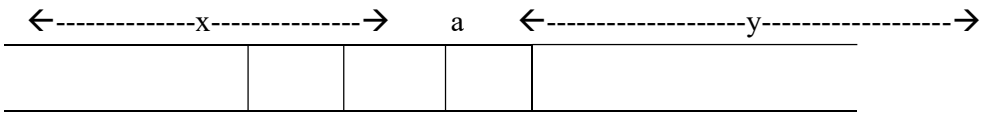
q_0=Starting state

Note : ∂ can be undefined for some parameters.

Instantaneous Description (ID)

ID must satisfy

i) Current state of the machine
ii) Content of the head.
iii) Current head position.



xq<u>a</u>y

This is to describe the status of a TM at some point, we must specify the current state, the complete contents of the tape(through the rightmost nonblank symbol), and the current position of the tape head.

Therefore the above instance of the TM may be represented by x q <u>a</u> y or by the pair(q, x <u>a</u> y)

Where q ε Q, x and y are strings over ΓU{ Δ} , a is a symbol in ΓU{ Δ} and the underlined symbol represents the tape head position.

- A move in TM may be represented as bellow
  X q <u>a</u> y I------- z r <u>b</u> w or (q ,x <u>a</u> y) I---------- (r, z <u>b</u> w)
  T T
- A sequence of moves may be represented as bellow

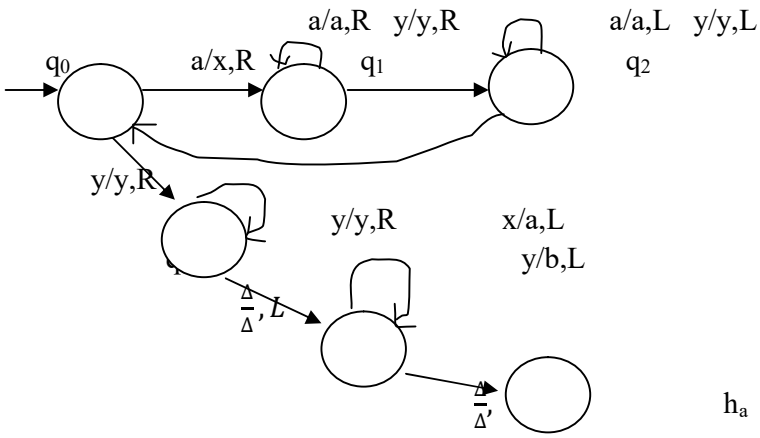
  X q <u>a</u> y I---*---- z r <u>b</u> w or (q, x <u>a</u> y) I-----*-------- (r, z <u>b</u> w)
  T T

For example, if T is currently in the configuration (q ,a a b a Δ a)
and $\delta(q,a) = \frac{r}{\Delta}, L)$ *we would write*
(q ,a a b a Δ a) I---*---- (r,a a <u>b</u> ΔΔa)

TM as an acceptor:

If T= (Q, ∑,Γ,q_0, δ) is a Turing machine , and x ∈∑ * , x is accepted by T, if starting in the initial configuration corresponding to input x ,T eventually reaches an accepting configuration. In other words, x is accepted if there exist y,Z ∈ (∑,U{ Δ }) * and a ∈ (∑,U{ Δ }) so that (q_o, Δ x) I---*----(h_a,y,<u>a</u>,z) when h_a is a halting state.

The language accepted by T is the set L(T) of input strings accepted by T.
Example: Let L = {$a^i b^i$ 1 i>=0} Construct a TM that accepts L.
Sol:



Assume L=aabb
Therefore ID at the beginning is Δ q_0 a a b b Δ
Following is the sequence of move

Δ q_0 a a b b Δ I---------- Δ x q_1 a b b Δ I---------- Δ x a q_1 b b Δ
T T
I---------- Δ x q_2 a y b Δ I---------- Δ q_2 x a y b Δ I---------- Δ x q_0 a y b Δ
T T T
I---------- Δ x x q_1 y b Δ I---------- Δ x x y q_1 b Δ I---------- Δ x x q_2 y y Δ
T T T
I---------- Δ x q_2 x y y Δ I---------- Δ x x q_0 y y Δ I---------- Δ x x y q_3 y Δ
T T T
I---------- Δ x x y y q_3 Δ I---------- Δ x x y q_4 y Δ I---------- Δ x x q_4 y b Δ
T T T
I---------- Δ x q_4 x b b Δ I---------- Δ q_4 x a b b Δ I---------- Δ q_4 Δ a a b b Δ
T T T
I---------- Δ h_a a a b b Δ
T

Encoding of Tuning machine, Universal Tuning machine:

- Tuning machine is created to execute a specific algorithm/ function. If we have a Tuning machine (TM) for computing one function, then computing a different function or doing some other calculation requires a different machine.
- A universal Tuning machine can simulate the operation of any Tuning machine. It works follows.

It is a TM Tu whose input consists essentially of a program and a data set for the program to process.

The process takes the form of a string specifying some other (special purpose) TM T1 , and the data set is a second string z interpreted as input to T1.

Tu then simulate the processing of Z by T1.

- The first step in doing that is to formulate a notational system in which we can encode both an arbitrary TM T1 and an input string z over an arbitrary alphabet as strings e(T1) and e(Z) over some fixed alphabet. The crucial aspect of the encoding is that it must not destroy any information, given the string e(T1) and e(z), we must be able to reconstruct the Tuning machine T1 and the string z. We will use the alphabet {0,1}, although we must remember that the TM we are encoding may have a much larger alphabet. We start by assigning positive integers to each state, each tape symbol, and each of the three " direction" S, L and R in TM T1, we want to encode.
- We use the following convention in order to make the encoding function one to one ie to guarantee that two different TMs will be encoded with different encoding.

  Convention: We assume from this point on that there are two fixed infinite sets Q={q1,q2,….}

  And S={a1,a2,…} so that for any TM T=(Q, ∑,Γ,q_0, δ) ,we have Q<=2 and Γ <=S.
- **The encoding function e:**

  First we associated to each tape symbol (including Δ), to each state (including ha and hr) and to each of the three " direction" , a string of O's let

  S(Δ) = O

  S(ai)= O^{i+1} (for each ai εQ)

  S(ha)=O

  S(hr)=OO

  S(qi)=O^{i+2} (for each qi εQ)

  S(S) =O

  S(L)=OO

  S(R)=OOO

  Each move m of a TM, described by the formula

  δ(p, a)= (q ,b,,D) is encoded by the string

  $$e(m) = S(P)1S(a)!s(q)1S(b)1S(D)1$$

  And for any TM T, with initial state q, T is encoded by the string

  e(T) = S(q)1 e(m_1)1 e(m_2) 1----1e(m_k)1

  Where ,m_1, m_2……, m_k are the distinct moves of T, arranged in some arbitrary order. Finally, any string z=z_1, z_2….,z_k , where each z_i ε S is encoded by

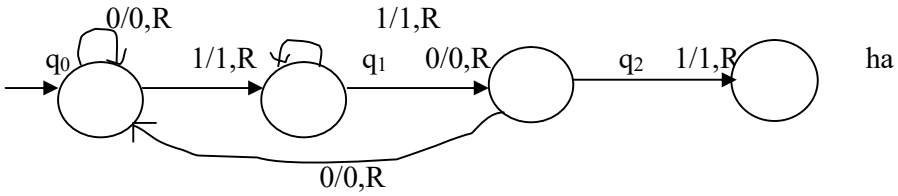
  e(z)=1S(z_1)1S(z_2)1….,1S(z_k)1

  Decision Problem: Answer are either yes(True) or no(False)

  Example:

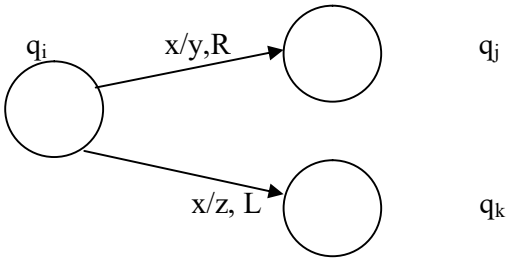
  Given x ε{ 0,Δ}* does x contain 101?

Corresponding TM:



An algorithm as well as a TM can be completely described by q0 m_1 m_2 m_3….m_k where q0= initial state of TM
m_1 m_2 m_3….m_k are distinct move of T arranged in some arbitrary order.

Non Deterministic Turing Machine (NDTM):



Definition:
A NDTM is a 5 tuple(Q, ∑,Γ , δ, q_0)
Where Q= A finite set of states not including the halting state ha, hr
∑= A finite set of input symbols.
Γ= A finite set of tape symbols not including the special tape symbol Δ (blank). ∑<=Γ
q_0= Starting state.

δ, = Q, * Γ<=(Q U {h0 , hr}) *(,Γ U { Δ}) * { R ;L ,S}

Acceptance by NDTM :
An input string x is deemed accepted by an NDTM , if at least one sequence of move with x as input leads to an accepting state.
i.e,
A string x ∈ ∑ * is accepted by T if for some a ∈ Γ ,U{Δ } and some y,z x ∈ (Γ , U{Δ })* ,

It is clear from above the Instantaneous Description (ID) of NDTM is same as TM.

Note: 1. Any language can be accepted by a NDTM can be accepted by an ordinary TM also

2. Also any language that can be accepted by an ordinary TM can be accepted by an NDTM

Problem classification:

Problem can be classified into two categories:

1) **Easy Problem:** Problem that can be solved in polynomial time.
   1, log n, n, n log n, n^2, $n^k(\log n)^j$, n^{k+1},….
   If the running time of the algorithm remains in this series then it is called polynomial time algorithm.

2) **Hard Problem:** Problems that needs exponential time algorithm to solve is called Hard Problem.
   That the running time is of the order of C^n or Ln 2^n,3^n,……k^n
   If the running time of the algorithm remains in this series then it is called exponential time algorithm.

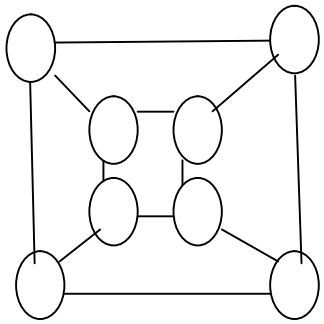
   [if $t(n) = O(2^n)$
   IE $t(n) = C.2^n$ and C=1 second and n=30 t(n)=34 years]

Example of some hard Problem:

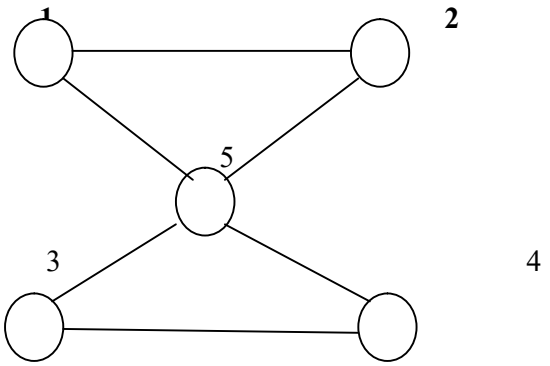
Hamiltonian Circuit Problem[HAM]:

Determine whether a given connected graph, G=(V,E) has a Hamiltonian Circuit. A Hamiltonian Circuit is a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

Example: Following is one of the Hamiltonian Circuit 1, 5, 8, 4, 3, 7, 6 ,2, 1



There is no Hamiltonian Circuit of the graph.



For completely connected graph the number of Hamilton Circuit is ½(n-1)!
Question: What is the necessary and sufficient condition for G to have Hamilton Circuit (1859)?
Ans: Unsolved till date.
To find the Hamiltonian Circuit of a connected graph G=(V,E) . We assumed that the vertices are numbered 1,2,3,….n
So a Hamiltonian Circuit may be represented by n,P1,P2,……Pn-1,n
Where n is the starting vertex and P1 is the permutation of the remaining (n-1) vertices.
Now there are (n-1)! possible permutation and n number of possible starting vertex. So total time should be of the order of O(n* (n-1)!) which is O(n!).

1) Algorithm HAM

```
Void HAM (n,E) //  n is the number of vertices numbered 1 to n.
O((n-1)!) → for (i=1, i<=n-1)! ; i++)
                {
                  Generate (P1, P2, P3 ,…… Pn-1) a permutation of (1, 2, 3 ,….. (n-1)
              P0=Pn=n;
  circuit _check=1;
  For (j=0 ;j<n; j++)
  If((Pi,Pj) ∄E)
O(n)→  { circuit _check=0;
  Break;
  }If(circuit_check==1)
  Print(Po,P!,P2,….Pn);}
  }
```

Complexity of the algorithm is O(n!)

2. Travelling salesman Problem [TSP]:

Find the shortest tour through n cities with known positive integer distances between them(i.e. find the shortest Hamiltonian circuit in a complete graph with positive integer weight).

Example :

	1	2	3	4	5
1	0	1	2	7	5
2	1	0	4	4	3
3	2	4	0	1	2
4	7	4	1	0	3
5	5	3	2	3	0

Tour 1: 5 – 1 – 2 – 3 – 4 – 5 distance travelled= 14
Tour 2: 5 – 1 – 2 – 4– 3 – 5 distance travelled= 13

There are (n-1)! Different tours for a particular starting city. There are n possible starting city. So total time should be of the order of O(n!).

Algorithm: Void TSP(W[][], n)
{ tour =Ø, min=α;
O((n-1)!) → for(i=1; i<=(n-1)!; i++)
{
Generate (P1, P2, P3 ,…… Pn-1) a permutation of (1, 2, 3 ,….. (n-1);
Construct tour t=(n,P1,P2,…Pn-1,n);
O(n) → calculate distance $\sum_{k=0}^{n} W[Pi][Pi+1]$ where P0=Pn=n;
If (distance<min)
{ Tour=t;
Min=distance; }
Print(min,tour);}

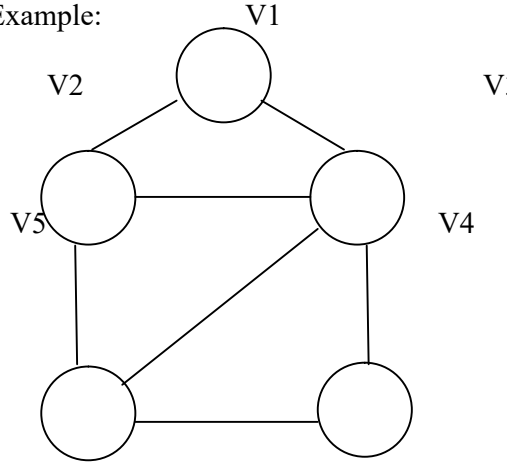
Complexity= O(n!)

3. The Clique Problem [CLIQUE]:

A Clique in an undirected graph G= (V,E) is a subset V'<=V of vertices, each pair of which is connected by an edge in E. In other words, a Clique is a complete sub graph of G. The size of a Clique is the number of vertices it contains.

Problem: Find a Clique of size K, if any.

Example:



Maximum size Clique is 3
(V1,V2,V3),(V2,V3,V5),(V3,V4,V5)

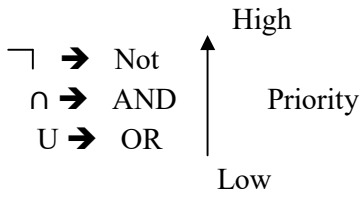
Algorithm:
Int Clique(G,K)
{
flag=0;
O(^nC_K)→for (all subset of V' of V having K elements)
{
O($^nC_2$)→ if (vertices in V' from a Clique)
{ print(V');
Flag=1;
Break;
}
} return(flag);
}

[Let n= I V I
Number of subset with K nodes is (^n_K) is ^nC_K]

Complexity : $\Omega (K^V .({}^nC_K))$
If K is constant → Polynomial time algo.
If K is Proportional with n → exponential time algo.

3) The SAT_CNF_Problem[SAT_CNF]

- A Boolean variable is a variable that can take a value from the set { TRUE, FALSE } or {1,0}
- A Boolean expression / formula is a Boolean variable or Boolean variable combined by the followings:



- **Literal:** A literal is a Boolean variable or negation of Boolean variable.
  Example:
  i) X
  ii) ¬ X (or often written as $\overline{X}$)

- **Clause:** A clause is a literal or disjunction of literals
  Example:
  L1 V L2 V …….V Lk is a clause where Li , 1<=i<=k is a literal.

- Boolean expression in CNF:
  A Boolean expression is said to be in CNF if it is a clause or a conjunction of clauses.
  Example: C1 ∩ C2 ∩ $C3$ ….∩ Cm where Ci, 1 <= I <= m is a clame.

  More Example:
  i) Xi ∩ (X2 U X3) U X4 → not in CNF
  ii) Xi ∩ (X2 U X3) ∩ X4 → is in CNF

The Problem: Find the assignment of the variables for which a given Boolean expression in CNF is satisfiable (i.e. the expression is time).

Example:
Consider the Boolean expression is X1 ∩ (X2 U X3) ∩ X4
Clearly this expression is in CNF. Some assignment for which the above expression is true

i) X1=1, X2=1, X3=0, X4=1
ii) X1=1, X2=0, X3=1 ,X4=1

If the expression has n number of variables, then the total number of different assignment to be checked is 2^n

So the solution of the problem has a complexity of the order of $O(2^n)$.

Recasting of a problem to Decision Problem:

1. [HAM]: does the graph G=(V,E) contains a Hamiltonian Circuit?
2. [CLIQUE]: Is there a clique of size k in G=(V,E) ?
3. [SAT _ CNF]: Are there assignments that make a Boolean expression in CNF satisfiable?
4. [TSP]: Is there a tour for travelling salesman with length smaller than or equal to k?

Question : Does such recasting reduces the impact of NP completeness?
Ans: No (WHY?)

Let P= Optimization version of a problem (say minimization)
PD= Decision version of the same problem with imposed bound.

Now, Given some way to solve P, we can solve PD as follows.

1. Solve P to find optimum value.
2. If (Optimum value<= bound)
   Return True
   Else
   Return False

Therefore
(P is easy) ➔ (PD is easy)
Or (PD is easy) ➔ (P is easy) [by taking negation]
Or PD is Hard ➔ P is hard.
[Note: Given some way to solve PD ,there is no general way to solve P although some possible care may solve P]

Encoding of problem Instances:
Objective: To write instances of a problem as strings over an arbitrary but fixed finite alphabet. [alphabet= {0,1}].
Example: Let P be a problem whose sole input is an integer K.
Complexity ➔ O(k)

Case 1: When encoded in binary
IKI=[$\log_2 K$]
Complexity ➔ O(n) where n=[$\log_2 K$]
Case 2: When encoded in unary
iKi=K=n^2
Complexity ➔ $O(n^2)$ where n=[$\log_2 K$]

Formally, we shall use the encoding e, such that e: I→ {0,1}*

Concrete Problem:

A concrete decision problem is a language L, the set of all "Yes" instances.

L= {xi X←{0,1}*, P(x)=1}

Definition 1: Complexity class P:

The complexity class P is the set of all languages each of which can be recognized by a corresponding TM/ Algorithm in Polynomial time.[TM is deterministic].

Definitions 2: Complexity class NP:

The complexity class NP is the set of all languages each of which can be recognized by a corresponding Non deterministic TM/ Algorithm in Polynomial time.[N stand for Non deterministic].

A non deterministic TM has the power to "guess" the right choice when faced with several options.Exam: X=a or b→ guess the next vertex.

Example:

```
TSP (Travelling salesman Problem):
Int  tsp bound)
{ tour= Ø ;  cost=0;
V=1;
Mark the vertex 1 as "visited" and all other as "Unvisited"
For (K=1;K<n;K++)
{
guess the edge (v,w) from v to unlimited
vertex w;
e=(u,w);
tour= tour U {e}
cost= cost + weight[e];
mark w as "visited"
v=w;
}
E=(v,1);
Tour=tour U{e}
cost = cost + weight[e];
if( cost<= bound)
return(1);
else return(0);
}
```

Complexity=O(n)

Considering guessing takes no time, so that it is in NP.

How to show that a problem is in NP?

Power of Non-determinism is the ability to guess the right option. We think of a TM/ algorithm that guesses a " solution" and then "verifies" that solution is correct.

[The solution in this case does not mean "Yes" or "No"- Formally a "solution" is called a certificate that can be verified]

For any X ε L ,there is a certificate y that can be used to proved that really X ε L. This proof is called verification. For any string X ε L, there should not be any certificate.

A verification algorithm A is a two argument algorithm, whose one argument is the input string X and other is the certificate Y such that A(X, Y) = 1.

Alternative definition of NP class:
The problem L ε NP if there exists a verification algorithm A and a constant K such that L ={ X I X ε { 0,1}* and there exists a certificate Y ε {0,1}* with I Y I ε O(I X I^K) such that A(X,Y)=1 }
[Y is the size of the algorithm]
Example: Show that TSP ε NP
Proof:
Let y be a certificate consisting of an ordered list of vertices defining a tour with a cost at must equal to the bound.
To verify the certificate, we must check
$O(n^2)$ → i) The given order of the vertices is a permutation of vertices V.
O(n * e) → ii)Each of the edges in the tour (including the one from the last to the first) actually exists in E.
O(n*e)→ iii) Sum of the costs of the edges is less than or equal to the bound.
Overall complexity of the algorithm is $O(\max(n^2, n*e))$
So TSP ε NP
Theorem: P<=NP
Some unsolved questions:
Q1: Is it possible that P = NP?
Q2: Is it possible that P ≠ NP?

Theorem: If X ε P then $\overline{X}$ ε P
Since X ε P
There must exist a polynomials time algorithm A(x) such that A(x) =1 for x ε X and A(x)=0 for x ε X.
Construct B(x) as
int B(x)
{ return ¬ A(x) ;
}
B(x)=1 for all x $\overline{\varepsilon}$ X.
B(x)=0 for all x ε X.
}
$\underline{X}$ ε P
Q3: Is the class NP closed under complement?
In other words
If X ε NP can we say $\overline{X}$ ε NP ?
Observation: The definition of NP is asymmetric.
Example of some NP problem:
1. HAM ε NP
   Suppose we have a certificate listing vertices in the order of a Hamiltonian Circuit.

   $O(n^V)$ → i) The given ordered list of vertices is a permutation of vertices in V.
   n=IVI
   O(n*e) → ii)Each edge defined by the consecutive vertices in the ordered list and the
   e=IEI edge from last to first, exists in E.

   Overall complexity = O(max(n2, n*e)) which is polynomial.

Hence HAM ε NP

2. SAT_CNF ε NP

   Proof:

Let Ø be a Boolean expression in CNF and
let $v_1, v_2, \ldots, v_r$ be the distinct variables in Ø.
Let $f((v_1, a_1), (v_2,a_2), \ldots,(v_r,a_r))$ be a certificate for Ø where a_i ε {0,1} denotes assignment for v_i;
1<=i<=r
Given Ø and f , the verification can be done in O(n) time where n=length of Ø.
Hence SAT_CNF ε NP.

3. CLIQUE ε NP

   Proof:

   Let v ' <=v of size k is the certificate.

   To verify that v ' is a k-clique check for every (u,v) ε v ' the edge (u,v) exists in E .

   Complexity of doing that

   $({}^{k}c_2) = k*(k-1)/2$ which is $O(k^2)$.

   Hence CLIQUE ε NP.

Example of a problem not in NP:

The Non Hamilton graph problem is not in NP.

Proof:

Decision Problem:

Does a graph G has no Hemiltonian Circuit?

No certificate is possible.

Hence not in NP.

Polynomial Reduction:

A language L1 is polynomial time reducible to a language L2 written as $L1 <=_P L2$ [is L1 no harder than L2]

If there exists a polynomial time computable function

F: {0,1}* → {0,1} * such that for all x ε {0,1}*, x ε L1 if f f(x) ε L2

[If a language L1 is polynomial time reducible to a language L2 also written as $L1 <=_T^P$ L2 is polynomial time reduction in the sense of turning machine].

Theorem:

1) The relation $<=_P$ is transitive i.e. if $L1 <=_P L2$ and $L2 <=_P L3$ then $L1<=_P L3$.

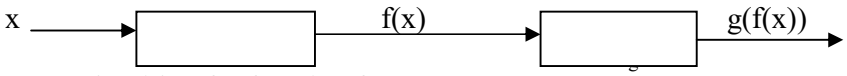
   Proof:

   Since $L1<=_P L2$ there is a polynomial time function f such that x ε L1 if f f(x) ε L2.

   Since $L2<=_P L3$ there is a polynomial time function f such that x ε L1 if f g(f(x)) ε L3.

   Hence, X ε L1 if f G(f(x)) ε L3.

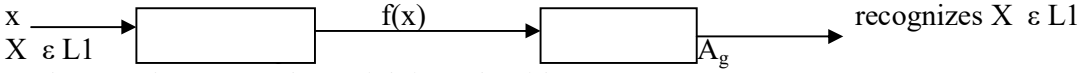
The algorithm that computes g(f(x)) is



$A_{f:}$ Algorithm for function f
A_g : Algorithm for function g
Hence g(f(x)) is polynomial time , $L1 <=_P L3$

Theorem 2:

If $L2 \varepsilon P$ and $L1 <=_P L2$ then $L1 \varepsilon P$
Proof:
Since $L2 \varepsilon P$, there must be a polynomial time algorithm A2 that recognizes $X \varepsilon L2$.
Let A_f be the polynomial time algorithm that computes the reduction function.
For any $X \varepsilon \{0,1\}^*$
$X \varepsilon L1$ if $f f(x) \varepsilon L2$ (as $L1 <=_P L2$)
The algorithm A1,that recognize $X \varepsilon L1$ is given below.



Both A_f and A_2 are Polynomial time algorithm
Therefore A_1 is also polynomial time algorithm.
So $L1 \varepsilon P$

Theorem 3:

If $L1 \varepsilon NP$ and $L1 <=_P L2$ then $L1 \varepsilon NP$.
Proof is similar to proof of Theorem 2. It is left as an exercise.
NP Completeness:
Definition:
A problem / language L is NP- complete if

a) $L \varepsilon NP$
b) $L' <=_P L$ for every $L' \varepsilon NP$

[Note : NPC= Set of all NP- complete problem. If only the condition (b) holds, L is called NP-hard.]

Theorem: Let X be an NP Complete problem and Y be another problem in NP such that $X <=_P Y$, then Y is also NP Complete.
Proof: $Y \varepsilon NP$ according to statement.
Now we have to show that Y is NP hard.
Since X is NP complete, for any $Z \varepsilon NP$ we have $Z <=_P X$
Since $X <=_P Y$ and the relation $<=_P$ is transitive we have $Z <=_P Y$
So, Y is NP hard.
General Approch for proving NP- completeness:

a) Prove that the problem $Y \varepsilon NP$
b) Prove that Y is NP Hard.

Proving of (a) is straightforward.
To prove (b) we must choose an appropriate problem X already known to be NP- complete.
Then in some tricky way we must show that $X <=_P Y$.
Assuming that HAM is in NPC, show that TSP is also in NPC.

Proof: TSP ε NP
Now to prove that TSP is NP – hard is HAM <=$_P$ TSP.
Let,
G=(V,E) be an instance of HAM.
We define f(G) as an instance of TSP as follows:

i) A Complete graph H=(V,V x V).
ii) The distance matrix ($C_{u,v}$)=1 if the edge (u,v) ε E and ($C_{u,v}$)=2 otherwise
iii) Bound=n where n is the number of vertices in G

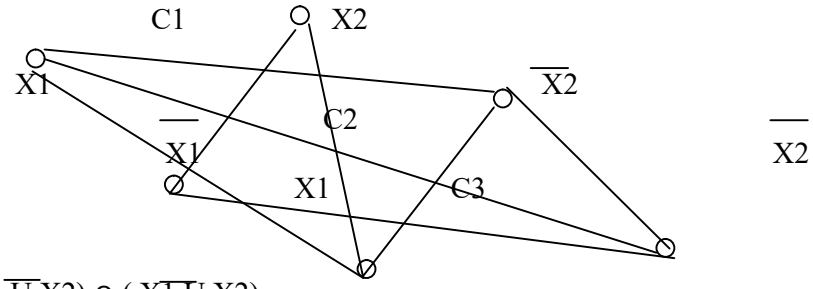
If G is an "Yes" instance of HAM; the Hamiltonian circuit in G translates into a tour in TSP with distance travelled=n.
If G is a "No" instance of HAM; the tour in TSP must contain at least one edge with distance =Z leading to overall distance travelled>n.
If G is an "Yes" instance of HAM if f f(a) is a "yes" instance of TSP
So, HAM <=$_P$ TSP
Cook's Theorem (1971):
If L ε NP then L <=$_P$ SAT_ CNF

Show that CLIQUE ε NPC
To show that CLIQUE is NP hard, we propose to show that
SAT_ CNF <=$_P$ CLIQUE
Let B be a a boolean expression in CNF, from this we construct a graph G and an integer K as follows-

i) K= no. of clauses in B
ii) Vertices of G are all the occurrences of literals in B .
iii) There is an edge in G between two such occurrences

- If they are in different clauses and
- If the two literals are consistent ie one is not a negation of the other.

Example:



$(X1 \cup X2) \cap (\overline{X1} \cup X2) \cap (\overline{X1} \cup \overline{X2})$
C1 C2 C3

We shall now show that G has a K-Clique if f B is satisfiable.
Let t^ : { X1,X2…..Xn} → {0,1} be a truth assignment satisfying B , where Xi , 1<=i<=n are the literal.
There is K number of clauses.
At least one literal in each clauses must be 1 .

Choose one such literals from each clause the vertices of G corresponding to these literals must be connected with each other.

The vertex cover problem (VCOVER):

A vertex cover of an undirected graph G=(V,E)is a subset V'<=V such that for any edge (u,v) ε E either u,v or both are in V'.

Decision Problem:

Given G and K , is there a vertex cover of size K?

Show that VCOVER ε NPC.

Proof:

i) VCOVER ε NP

Given a certificate V' linear time is required to show whether all vertices in V' ε V

ii) VCOVER is NP Hard

Now, We propose to show the CLIQUE $<=_P$ VCOVER

Let G=(V, E) be the given graph with n vertices.

$\overline{G} = (\overline{V} , \overline{E})$ the complement of G.

We claim that a set S<= V is a clique in G If f V-S is a vertex cover of G .

If S is a clique in G , no edge in G connects two vertices in S. So, the remaining vertices in V-S cover all the edges in G.

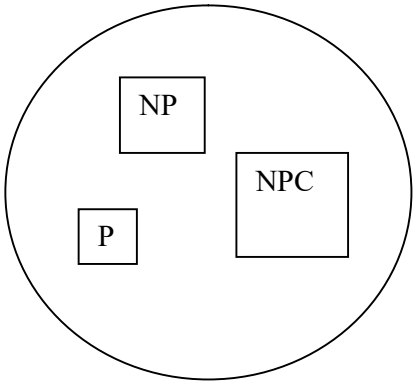
Alternatively, If G-S is a $\overline{\text{vertex}}$ cover of $\overline{\overline{G}}$.o edge in G connects two vertices in S. Therefore, every pair of vertices in S is connected.

S is clique.

CLIQUE $<=_P$ VCOVER

Hence proved.

Relationships among P, $\overline{NP}$ and NPC.



The above ven diagram shows the relationships among P , NP and NPC that the most theoretical computer scientists belive.

MODULE 05: Approximation Algorithm: [2L]

Why Approximation Algorithm?

There are some combinatorial optimization problems such as Travelling salesman problem (TSP) , knapsack Problem, the decision version of which are NP- complete but the Optimization version are in the class of NP- hard problems(Problems that are at least as hard as NP- complete problem).Hence there are no known polynomial time algorithms for these problems and there are serious theoretical reasons to believe that such algorithms do not exist. What then are our options for handling such problems, many of which are of significant practical importance?

There is a radically different way of dealing with different optimization problems. Solve them approximately by a fast algorithm. This approach is particularly appealing for applications where a good but not necessarily optimal solution will suffice.

Approximation Algorithm:

An algorithm that produces that produces near–optimal solution to an optimization solution to an optimization algorithm.

In the case of approximation algorithm there must be provable solution quality and provable run time bound.

Performance bounds for Approximation Algorithm:

1. Ratio Bound:

   C= cost of the solution produced by the Approximation Algorithm

   C*= cost of optimum solution.

   We say that an approximation algorithm has a ratio bound of ρ(n)., if for any input of size n

   Max(C/C^* , C^*/C)<= ρ(n)

   When ρ(n) is independent of n, we say, the solution of the approximation algorithm is within a factor of ρ.

2. Relative Error Bound:

   Relative Error Bound ε(n) is defined as I C- C^*I/C^*<= ε(n)

   For any input of size n.

   Example:

   Show that ε(n)<= ρ(n)-1

   For minimization problem 0<=C^*<=C

   Therefore ,

   C/C^*= ρ(n)

   ε(n)= I C- C^*I/C^*= C/C^* -1 = ρ(n)-1

   For maximization problem 0<C<=C^*

   Therefore C/C^*= ρ(n)

   ε(n)= I C- C^*I/C^* = I C^*-CI/C^* = 1-1/ ρ(n) = ρ(n)-1/ ρ(n)

   since

   ρ(n)>=1, ε(n)<= ρ(n)-1

3. Approximation Scheme

An approximation scheme is an Approximation Algorithm that takes as input not only an instance of the problem but also a value $\varepsilon>0$ such that for only fixed ε , the scheme has an approximation algorithm with relative error bound ε.

3 a) **Polynomial Time Approximation Scheme (PTAS):**

An approximation scheme for fixed $\varepsilon>0$ that runs in polynomial time in the input size n.

3 b) **Fully PTAS:**

An approximation scheme such that its running time is polynomial both in $1/\varepsilon$ and n where ε is relative error bound and n is the input size,$[0((1/\varepsilon)^2 n^3)]$.

Example:

Minimum Vertex Cover:

```
Set approx vector (V,E)
{
C=Ø;
While(E!= Ø)
{
E={u,v} where (u,v) is an arbitrary edge in E;
C=C U e;
Remove from E any edge incident on either u or v;
}
Return( C);
}
```

Runtime complexity 0(e) where e= no of edges.

The approximation algorithm has a ratio bound of 2 .

Proof:

Let A denote the set of edges picked up by the algorithm to include its end points in C

Therefore,

ICI =2 IAI

No two edges in A share the same end points

Let

C*= Optimal vertex cover

C* must contain at least one end point each edge in A, to cover the edges in A.

Since no two edges in A share the same end points. No vertex cover is incident on more than one edge in A.

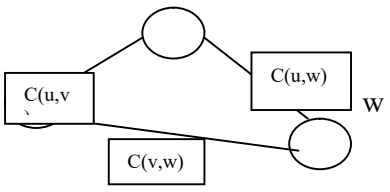
Therefore IAI<=I C*I

ICI = 2IAI <= 2 I C*I

So, the ratio bound is 2.

TSP with Triangle Inequality

Triangle Inequality:



v

C(u,w)<=C(u,v)+C(v,w) for all u and v and w.
U is the distance between nodes u and w cannot exceed the distance of two leg path from u to some intermediate node v to w.

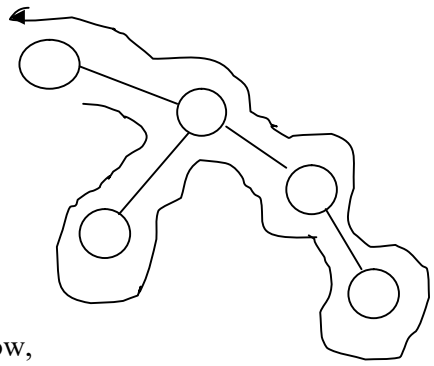
Hamiltonian Circuit approx –TSP (V,E,C)

Select a vertex r ε v to be the root vertex;
Grow a minimum spanning tree T for the graph from r using Prim's algorithm;
Let L=list of vertices visited in a preorder tree walk of T;
Return Hamiltonian circuit H that visits the vertices in the order of L;

The Approx-TSP algorithm has a ratio bound of 2.
Proof:
Let, H* = Optimal Tour
H=Tour return by Approx TSP.

To show that C(H)<=2C(H*)
We delete an edge from H*
Let T* be the resulting spanning tree.
If T* is a minimum spanning tree

C(T)<=C(T*) <=C(H*)
Or C(T)<= C(H*) ……..(1)
A full walk of T lists the vertices when they are first visited and whenever they are returned to after a visit to a sub tree.
Let this full walk be denoted by W.



Now,
Every edge in T is traversed twice. Therefore
C(W)<=2.C(T) ……(2)

From (1) and (2)
C(W)<=2.C(H*) ……(3)

By triangle inequality, we can delete a visit from W and the cost does not increase. By repeatedly applying the delete operation, we remove from W all but the first visit to each vertex. This ordering is same as the preorder walk.
Now,
C(H)<=C(W)…….(4)
From (3) and (4)
C(H) <=2C(H*).

0/1 Knapsack Problem:

Consider the Knapsack instance n=3, m=100, {P_1,P_2,P_3}={20,10,19} and {W_1, W_2,W_3}={65,20,35} X_1,X_2,X_3)=(1,1,1) is not a feasible solution as $\sum W_iX_i$>M. the solution (X_1,X_2,X_3)=(1,0,1) is an optimal solution. It value $\sum P_iX_i$ is 39, hence F*(I)=39 for this instance. The solution (X_1,X_2,X_3)=(1,1,0) is suboptimal. Its value is $\sum P_iX_i$=30.
This is a candidate for a possible output from an approximation algorithm.In fact every feasible solution is a candidate for output by an approximation algorithm .If the solution(1,1,0) is generated by an approximation algorithm on this instance then F^(I)=30

I F*(I) - F^(I) I=9 and I F*(I) - F^(I) I/F*(I)=0.3
Now consider the following approximation algorithm for the 0/1 knapsack problem. Assure that the objects are in non increasing order of Pi/Xi .If object i fits then set Xi=1,
Otherwise set Xi=0. When this algorithm is used on the instance of the above problem, the objects are considered in the order 1,3,2. The result is (X1,X2,X3) =(1,0,1).The optimal solution is obtained. Now, consider the following instance n=2, (P1,P2)=(2,r),
(W1, W2)=(1,r) and M=r. When r>1 the optimal solution is (X1,X2)=(0,1). Its value F*(I) is r. The solution generated by the approximation algorithm is (X1,X2)=(1,0).
Its value F^(I) is 2.
Hence , I F*(I) - F^(I) I=r-2. This approximation algorithm is not an absolute approximation algorithm as there exists no constant k such that I F*(I) - F^(I) I<=k for all instances I. Furthermore, note that I F*(I) - F^(I) I/F*(I)=1- 2/r .This approaches 1 as r becomes large . I F*(I) - F^(I) I/F*(I)<=1 for every feasible solution to every knapsack instance. Since the above algorithm always generates a feasible solution, it is a 1- approximation algorithm. It is, however not an ε approximation algorithm for any approximation algorithm ε, ε<1.
Following is the ε approximation algorithm for 0/1 knapsack problem for any ε, 0<ε<1.

Procedure ε-Approx (P,W,M,N,K)

// The size of a combination is the number of objects in it.

// The weight of a combination is the sum of weights of the objects in that combination

// k is a non-negative integer which defines the order of the algorithm.

1. PMAX <- 0;
2. For all combination I of size <=k and weight <=M do

3. P_I<-$\sum_{i \in I}$ Pi
4. PMAX<-Max(PMAX, P_I+L(I,P,W,M,N)
5. Repeat
6. End ε-Approx

Procedure L(I,P,W,M,N)

S<-0
I<-1
T<-M- $\sum_{i \in I}$ Wi
For I <- 1 to n do
If i ∌ I and W_i <=T
S <-S<-S+ $P_{i;}$
T <- T- W_i *;*
End if
Repeat
Return(S) ;
End L;
n_i

In this procedure P and W are sets of profits and weights respectively .It is assumed that Pi/Wi>= Pi+1/Wi+1, 1<=i<N.
M is the Knapsack capacity and K a non-negative integer. In the loop of lines 2-5, all $\sum_{i=0}^{k}$ Ni , different subsets, I consisting of at most K of the n objects and generated. If the currently generated subset I is such that $\sum_{i \in I}$ Wi>M. It is discarded (as it is infeasible).Otherwise, the space remaining in the knapsack (i<M-$\sum_{i \in I}$ Wi is filled using the procedure L.
Consider the Knapsack Problem instance with n=8 objects.
Size of Knapsack = m=110, P={11,21,31,33,43,53,55,65} and W={1,11,21,23,33,43,45,55}.

The optimal solution is obtained by putting objects 1,2,3,5 and 6 into the knapsack. This results in an optimal profit P*, of 159 and weight of 109

We obtain the following approximations for different K:
K=0, PMAX is just the lower bound solution L(∅,P,W,M,N);
PMAX=139; X=(1,1,1,1,1,0,0,0); W=$\sum_i$ Xi Wi=89;
(P*-PMAX)/P*=20/159=126

K=1, PMAX =151; X=(1,1,1,1,,0,0,1,0); W=101;
(P*-PMAX)/P*=8/159=.05

K=2, PMAX =P*=159; X=(1,1,1,0,1,1,0,);w=109;

The following table gives the details for K=1. It is interesting to note that the combinations I= {1},{2},{3},{4}.{5} need not be tried since for 1={Ø}, X6 is the first Xi which is 0 and so there combinations will yield the same .

PMAX as I ={Ø} .This will be true for all combinations I that include only objects for which Xi was 1 in the solution I={Ø}.

PMAX $X_{optional}$	P_I	R_I	L	PMAX=MAX{PMAX, P_I+L}
Ø 0 (1,1,1,1,1,0,0,0)	11	1	128	139
6 139 (1,1,1,1,0,1,0,0)	53	43	96	149
7 149 (1,1,1,1,0,0,1,0)	55	45	9	151
8 151 (1,1,1,1,0,0,1,0)	65	55	63	151

QUESTIONS:

1. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (Or) or several *terms*, each of which is the conjunction (And) of one or more literals. For example, the formula

$$(x \wedge y \wedge \bar{z}) \vee (y \wedge \bar{z}) \vee (x \wedge y \wedge z)$$

is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

(a) Describe a polynomial-time algorithm to solve DNF-SAT.

(b) What is the error in the following argument that P=NP?

> Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,
>
> $$(x \vee y \vee z) \wedge (\bar{x} \vee \bar{y}) \Leftrightarrow (x \wedge \bar{y}) \vee (y \wedge \bar{x}) \vee (z \wedge \bar{x}) \vee (\bar{z} \wedge y)$$

Now we can use the algorithm from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time. Since 3SAT is NP-hard, we must conclude that P=NP!

2. (a) Describe a polynomial-time reduction from Partition to Sub set Sum.
   (b) Describe a polynomial-time reduction from Sub set Sum to Partition.

3. (a) Describe a polynomial-time reduction from Undirected Hamiltonian Cycle to Directed Hamiltonian Cycle.
   (b) Describe a polynomial-time reduction from Directed Hamiltonian Cycle to Undirected- Hamiltonian Cycle.

4. (a) Describe a polynomial-time reduction from Hamiltonian Path to Hamiltonian Cycle.
   (b) Describe a polynomial-time reduction from Hamiltonian Cycle to Hamiltonian Path.
   [Hint: A polynomial-time reduction may call the black-box subroutine more than once.]

5. (a) Prove that Planar Circuit Sat is NP-hard. *[Hint: Construct a gadget for crossing wires.]*
   (b) Prove that Not All Equal 3SAT is NP-hard.
   (c) Prove that the following variant of 3SAT is NP-hard: Given a boolean formula Φ in conjunctive normal form where each clause contains at most 3 literals and each variable appears in at most 3 clauses, does Φ have a satisfying assignment?

6. (a) Using the gadget on the right below, prove that deciding whether a given *planar* graph is 3-colorable is NP-hard. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*
   (b) Using part (a) and the middle gadget below, prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-hard. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*

Reference Book(s):

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, “Introduction to Algorithms”
2. A. Aho, J.Hopcroft and J.Ullman “The Design and Analysis of Algorithms” D.E.Knuth “The Art of Computer Programming”, Vol. 3 Jon Kleiberg and Eva Tardos, "Algorithm Design"
3. E.Horowitz and Shani “Fundamentals of Computer Algorithms”
4. K.Mehlhorn, “Data Structures and Algorithms” - Vol. I & Vol. 2.
5. S.Baase “Computer Algorithms”
6. E.M.Reingold, J.Nievergelt and N.Deo- “Combinational Algorithms- Theory and Practice”, Prentice Hall, 1997.