Unit 1

Foundation of Algorithm Analysis

Algorithm and its properties

An algorithm is a finite set of computational instructions, each instruction can be executed in finite time, to perform computation or problem solving by giving some value, or set of values as input to produce some value, or set of values as output. Algorithms are not dependent on a particular machine, programming language or compilers i.e., algorithms run in same manner everywhere. So, the algorithm is a mathematical object where the algorithms are assumed to be run under machine with unlimited capacity.

Examples of problems

- we are given two numbers; how do we find the Greatest Common Divisor.

- Given an array of numbers, how do we sort them?

We need algorithms to understand the basic concepts of the Computer Science, programming. Where the computations are done and to understand the input output relation of the problem, we must be able to understand the steps involved in getting output(s) from the given input(s).

 we need designing concepts of the algorithms because if we only study the algorithms then we are bound to those algorithms and selection among the available algorithms. However, if you have knowledge about design then we can attempt to improve the performance using different design principles.

The analysis of the algorithms gives a good insight of the algorithms under study.

Analysis of algorithms tries to answer few questions like; is the algorithm correct? i.e., the Algorithm generates the required result or not? does the algorithm terminate for all the inputs under problem domain? The other issues of analysis

are efficiency, optimality, etc. So knowing the different aspects of different algorithms on the similar problem domain we can choose the better algorithm for our need. This can be done by knowing the resources needed for the algorithm for its execution. Two most important resources are the time and the space. Both of the resources are measures in terms of complexity for time instead of absolute time we consider growth

Algorithms Properties

Input(s)/output(s): There must be some inputs from the standard set of inputs and an algorithm's execution must produce outputs(s).

Definiteness: Each step must be clear and unambiguous.

Finiteness: Algorithms must terminate after finite time or steps.

Correctness: Correct set of output values must be produced from the each set of inputs.

Effectiveness: Each step must be carried out in finite time.

Here we deal with correctness and finiteness.

RAM model

This RAM model is the base model for our study of design and analysis of algorithms to have design and analysis in machine independent scenario. In this model each basic operations (+, -) takes 1 step, loops and subroutines are not basic operations. Each memory reference is 1 step. We measure run time of algorithm by counting the steps.

Time and Space Complexity

Best, Worst and Average case

Best case complexity gives lower bound on the running time of the algorithm for any instance of input(s). This indicates that the algorithm

can never have lower running time than best case for particular class of problems.

Worst case complexity gives upper bound on the running time of the algorithm for all the instances of the input(s). This ensures that no input can overcome the running time limit posed by worst case complexity.

Average case complexity gives average number of steps required on any instance of the input(s).

Concept of Aggregate Analysis

Asymptotic Notations:

Big-O

When we have only asymptotic upper bound then we use O notation. A function $f(x)=O(g(x))$ (read as $f(x)$ is big oh of $g(x)$) iff there exists two positive constants c and x_0 such that for all $x >= x_0$, $0 <= f(x) <= c*g(x)$
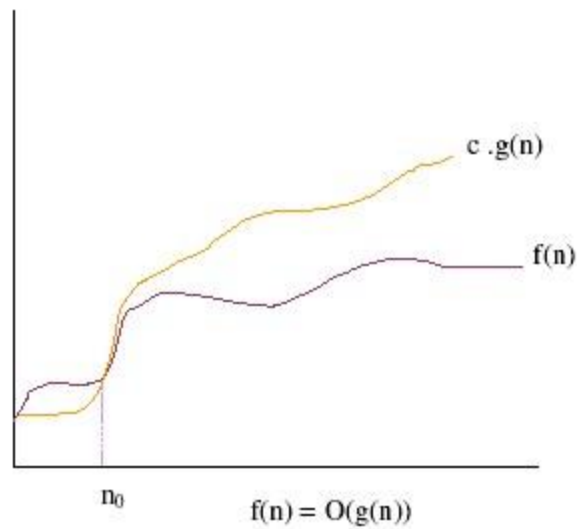
The above relation says that $g(x)$ is an upper bound of

$f(x)$ **Some properties:**

Transitivity: $f(x) = O(g(x))$ & $g(x) = O(h(x))$ _ $f(x) = O(h(x))$

Reflexivity: $f(x) = O(f(x))$

O(1) is used to denote constants.

$c.g(n)$

$f(n)$

n_0

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

For all values of n >= n_0, plot shows clearly that f(n) lies below or on the curve of c*g(n)

Examples

1. $f(n) = 3n^2 + 4n + 7$

g(n) = n^2, then prove that f(n) = O(g(n)).

Proof: let us choose c and n_0 values as 14 and 1 respectively then we can have f(n) <= c*g(n), n>=n_0 as

$3n^2 + 4n + 7 <= 14*n^2$ for all n >= 1

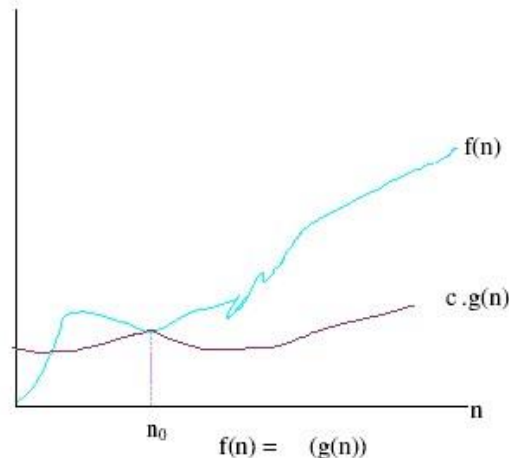The above inequality is trivially true

Hence f(n) = O(g(n))

Big-Ω

Big omega notation gives asymptotic lower bound. A function $f(x) = \Omega(g(x))$ (read as g(x) is big omega of g(x)) iff there exists two positive constants c and x_0 such that for all $x >= x_0$, $0 <= c*g(x) <= f(x)$.

The above relation says that g(x) is a lower bound of f(x).

Some properties:

Transitivity: $f(x) = \Omega(g(x))$ & $g(x) = \Omega(h(x))$ _ $f(x) = \Omega(h(x))$

Reflexivity: $f(x) = \Omega(f(x))$
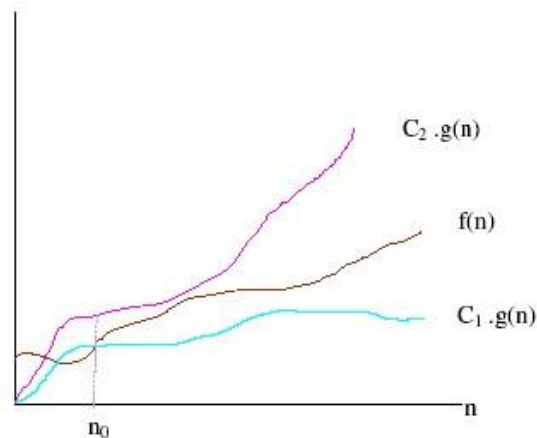
f(n) = (g(n))

For all values of n >= n_0, plot shows clearly that f(n) lies above or on the curve of c*g(n).

Examples

1. $f(n) = 3n^2 + 4n + 7$

g(n) = n^2, then prove that $f(n) = \Omega(g(n))$.

Proof: let us choose c and n_0 values as 1 and 1, respectively then we can have f(n) >= c*g(n), n>=n_0 as

$3n^2 + 4n + 7 >= 1*n^2$ for all n >= 1

The above inequality is trivially true

Hence $f(n) = \Omega(g(n))$

Big-Θ

When we need asymptotically tight bound then we use notation. A function $f(x)$ = Θ $(g(x))$ (read as $f(x)$ is big theta of $g(x)$) iff there exists three positive constants c_1, c_2 and x_0 such that for all $x >= x_0$, $c_1*g(x) <= f(x) <= c_2*g(x)$

The above relation says that $f(x)$ is order of

g(x) **Some properties:**

Transitivity: $f(x)$ = Θ $(g(x))$ & $g(x)$ = Θ $(h(x))$, $f(x)$ = Θ $(h(x))$

Reflexivity: $f(x)$ = Θ $(f(x))$

Symmetry: $f(x)$ = Θ $g(x)$ iff $g(x)$ = Θ $f(x)$

$f(n) = $ Θ $(g(n))$

For all values of $n >= n_0$, plot shows clearly that $f(n)$ lies between $c_1* g(n)$ and $c_2*g(n)$.

Examples

1. $f(n) = 3n^2 + 4n + 7$

g(n) = n^2, then prove that $f(n)$ = Θ $(g(n))$.

Proof: let us choose c_1, c_2 and n_0 values as 14, 1 and 1 respectively then we can have, $f(n) <= c_1*g(n)$, $n>=n_0$ as $3n^2 + 4n + 7 <= 14*n^2$, and $f(n) >= c_2*g(n)$, $n>=n_0$ as $3n^2 + 4n + 7 >=$ $1*n^2$ for all $n >= 1$(in both cases).

So $c_2*g(n) <= f(n) <= c_1*g(n)$ is trivial.

Hence $f(n) = \Theta (g(n))$.

Recurrences: Recursive Algorithms and Recurrence Relations

- Recursive algorithms are described by using recurrence relations.

- A recurrence is an inequality that describes a problem in terms of itself.

For Example:

Recursive algorithm for finding factorial

$T(n)=1$ when n =1

$T(n)=T(n-1) + O (1)$ when n>1

Recursive algorithm for finding Nth Fibonacci number

$T (1) =1$ when n=1

$T (2) =1$ when n=2

$T(n)=T(n-1) + T(n-2) +O (1)$ when n>2

Recursive algorithm for binary search

$T (1) =1$ when n=1

$$T(n)=T(n/2) + O(1) \qquad \text{when } n>1$$

Solving Recurrences

Recursion Tree Method

Just Simplification of Iteration method:

Consider the recurrence

$T(1)=1$ when n=1

$T(n)= T(n/2)+ 1$ when n>1

```
   1            1                    1
   |            |                    |
   v            v                    v
 T(n/2)         1                    1
                |                    |
                v                    v
              T(n/4)                 1
                                     :
                                     :
                                   T(n/2^k)
```

Cost at each level =1

For simplicity assume that $n= 2^K$

⇨   k= logn

Summing the cost at each level,

Total cost = 1 + 1 + 1 + …………………. Up to logn terms
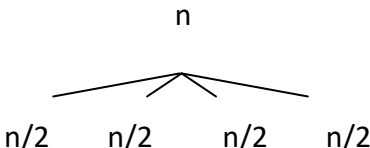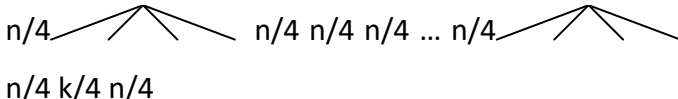
⇨   complexity = O (logn)

Second Example

$$T(n) = 1 \qquad\qquad n=1$$

$$T(n) = 4T(n/2) + n \qquad\qquad n>1$$

Third Example T(n)	Cost at this level
n	n
n/2 n/2 n/2 n/2	2n
n/4 ⟋⟍ n/4 n/4 n/4 … n/4 ⟋⟍ n/4 k/4 n/4	2^2n
… … … … … … … …	…
1 … 1	2kn

Asume: n= 2^k

$\Rightarrow$ k= logn

$T(n) = n + 2n + 4n + … + 2^{k-1}n + 2^kn$

$= n(1 + 2 + 4 + … + 2^{k-1} + 2^k$

$= n(2k+1 -1)/(2-1)$

$= n(2^{k+1} - 1)$
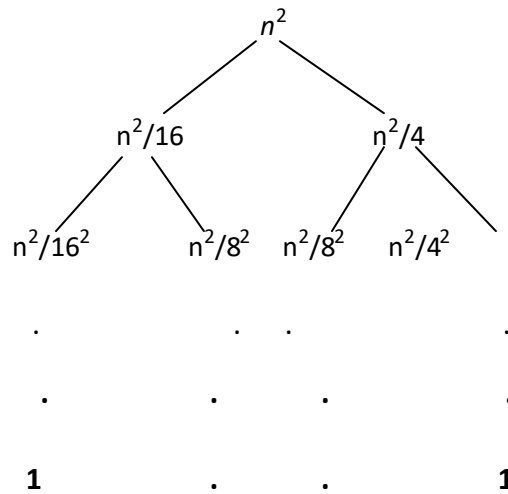
$= n(2.2^k - 1)$

$= 2n^2 -n$

$<= 2n^2$

$= O(n^2)$

Example: Solve $T(n) = T(n/4) + T(n/2) + n^2$

Total Cost $\leq n^2 + 5\ n^2/16 + 5^2 n^2/16^2 + 5^3\ n^2/16^3 + \ldots\ldots\ldots\ldots\ldots\ldots +5^k n^2/16^k$

{why $\leq$? Why not $=$?}

$\qquad = n^2\ (1 + 5/16 + 5^2/16^2 + 5^3/16^3 + \ldots\ldots\ldots\ldots\ldots + 5^k/16^k)$

$\qquad = n^2 + (1 - 5^{k+1}/16^{K+1})$

$\qquad = n^2 + \text{constant}$

$\qquad = \quad O(n^2)$

Substitution Method

Takes two steps:

1. Guess the form of the solution, using unknown constants.

2. Use induction to find the constants & verify the solution.

   Completely dependent on making reasonable guesses

   Consider the example:

   $$T(n) = 1 \qquad\qquad n=1$$

   $$T(n) = 4T(n/2) + n \qquad\qquad n>1$$

   Guess: $T(n) = O(n^3)$.

   More specifically:

   $T(n) <= cn^3$, for all large enough n.

   Prove by strong induction on n.

   Assume: $T(k) <= ck^3$ for all k<n.

                  $T(n) <= cn^3$ for all

   Show: n>n0.

   Base case,

        For n=1:

        $T(n) = 1$ Definition

        $1 <= c$ Choose large enough c for conclusion

   Inductive case, n>1:

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

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

       $= c/2\ n^3 + n$ Algebra.

   While this is $O(n^3)$, we're not done.

   Need to show $c/2\ n^3 + n <= c\ n^3$.

Fortunately, the constant factor is shrinking, not growing.

$T(n) <= c/2 \, n^3 + n$ From before.

$\quad = cn^3 - (c/2 \, n^3 - n)$ Algebra.

$\quad <= cn^3$ Since n>0, if c>2

Proved:

$T(n) <= 2n^3$ for all n>0

Thus, $T(n) = O(n^3)$.

Second Example

$\quad T(n) = 1$ n=1

$\quad T(n) = 4T(n/2) + n$ n>1

$\quad\quad$ Guess: $T(n) = O(n^2)$.

$\quad\quad$ Same recurrence, but now try tighter bound.

$\quad$ More specifically:

$\quad T(n) <= cn^2$ for all n>n0.

$\quad T(k) <= ck^2$, for all k <n.

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

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

$\quad\quad = cn^2 + n$

$\quad\quad\quad$ Not $<= cn^2$!

Problem is that the constant isn't shrinking.

Solution: Use a tighter guess & inductive hypothesis.

Subtract a lower-order term – a common technique.

Guess:

T(n) <= cn^2 - dn for all n>0

Assume T(k) <= ck^2 - dk, for all k<n. Show T(n) <= cn^2 - dn.

Base case, n=1

T(n) = 1   Definition.

1 <= c-d   Choosing c, d appropriately.

Inductive case, n>1:

T(n) = 4T(n/2) + n	Definition.
<= $4(c(n/2)^2$ - d(n/2)) + n	Induction.
= cn^2 - 2dn + n	Algebra.
= cn^2 - dn - (dn - n)	Algebra.
<= cn^2 - dn	Choosing d>1.

T(n) <= $2n^2$ – dn for all

n>0

Thus, T(n) = $O(n^2)$.

Ability to guess effectively comes with experience.

Masters Method

Cookbook solution for <u>some</u> recurrences of the form

$$T(n) = a \cdot T(n/b) + f(n)$$

where

$a \geq 1, b > 1, f(n)$ asymptotically positive

Describe its cases

Master Method Case 1

$$T(n) = a \cdot T(n/b) + f(n)$$

$f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0 \;\rightarrow\; T(n) = \Theta(n^{\log_b a})$

$$T(n) = 7T(n/2) + cn^2 \qquad\qquad a=7, b=2$$

Here $f(n) = cn^2 \quad n^{\log_b a} = n^{\log_2 7} = n^{2.8}$

$\Rightarrow\; cn^2 = O(n^{\log_b a - \varepsilon})$, for any $\varepsilon \leq 0.8$.

$$T(n) = \Theta(n^{\lg_2 7}) = \Theta(n^{2.8})$$

Master Method Case 2

$$T(n) = a \cdot T(n/b) + f(n)$$

$f(n) = \Theta(n^{\log_b a}) \;\rightarrow\; T(n) = \Theta(n^{\log_b a} \lg n)$

$$T(n) = 2T(n/2) + cn \qquad\qquad a=2, b=2$$

Here $f(n) = cn \quad n^{\log_b a} = n$

$\Rightarrow\; f(n) = \Theta(n^{\log_b a})$

$$T(n) = \Theta(n \lg n)$$

Master Method Case 3

$$T(n) = a \cdot T(n/b) + f(n)$$

$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and
$a \cdot f(n/b) \le c \cdot f(n)$ for some $c < 1$ and all large enough n
$$\rightarrow \quad T(n) = \Theta(f(n))$$

I.e., is the constant factor shrinking?

$$T(n) = 4 \cdot T(n/2) + n^3 \qquad a=4,\ b=2$$

$$n^3 \overset{?}{=} \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{\log_2 4 + \varepsilon}) = \Omega(n^{2 + \varepsilon}) \text{ for any } \varepsilon \le 1.$$

Again, $4(n/2)^3 = \tfrac{1}{2} \cdot n^3 \le cn^3$, for any $c \ge \tfrac{1}{2}$.
$$T(n) = \Theta(n^3)$$

Master Method Case 4

$$T(n) = a \cdot T(n/b) + f(n)$$
None of previous apply. Master method doesn't help.

$$T(n) = 4T(n/2) + n^2/\lg n \qquad a=4,\ b=2$$

Case 1? $n^2/\lg n = O(n^{\log_b a - \varepsilon}) = O(n^{\log_2 4 - \varepsilon}) = O(n^{2 - \varepsilon}) = O(n^2/n^\varepsilon)$

No, since $\lg n$ is asymptotically less than n^ε.
Thus, $n^2/\lg n$ is asymptotically greater than n^2/n^ε.

Case 2? $n^2/\lg n \overset{?}{=} \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

No.

Case 3? $n^2/\lg n \overset{?}{=} \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{\log_2 4 + \varepsilon}) = \Omega(n^{2 + \varepsilon})$

No, since $1/\lg n$ is asymptotically less than n^ε.

So we can not solve this recurrence relation using mater method.

Unit 2

Iterative Algorithms

Basic Algorithms:

Algorithm for GCD

Inputs: Two numbers *a* and *b*

Output: G.C.D of *a* and *b.*

Algorithm: assume (for simplicity) a>b>=0

```
gcd(a,b)
{
        While (b != 0)

        {
        d =a/b ;
         temp = b;
        b = a − b * d
        a = temp;
        }
    return a;

    }
```

Running Time: if the given numbers *a* and *b* are of n-bits then loop executes for n time and the division and multiplication can be done in O (n^2) time. So, the total running time becomes $O(n^3)$.

Another way of analyzing:

For Simplicity Let us assume

    that b=2^n

⇨ n= logb

⇨ Loop executes logb times in worst case

⇨ Time Complexity = o(logb)

Space Complexity: The only allocated spaces are for variables so space complexity is constant i.e. O(1).

Fibonacci Number

Input: *n*

Output: n^{th} Fibonacci number.

Algorithm: assume *a* as first(previous) and *b* as second(current) numbers fib(n)

```
{
 a = 0, b= 1, f=1 ;
   for(i = 2 ; i <=n ;
       i++)
       {
```

```
        f = a+b ;

        a=b     ;

        b=f ;

        }

    return f ;

    }
```

Time Complexity: The algorithm above iterates up to n-2 times, so time complexity is O(n).

Space Complexity: The space complexity is constant i.e. O(1).

Searching Algorithms

Sequential Search

Simply search for the given element left to right and return the index of the element, if found.

Otherwise return "Not Found".

Algorithm:

```
    LinearSearch (A, n, key)

    {

        for (i=0; <n; i++)

        {

                if(A[i] == key)

                {

                        return I;

                }

        }
```

```
            return -1;//-1 indicates unsuccessful search
        }
```

Analysis:

Time complexity = O(n)

Sorting Algorithms

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In-place: The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

Bubble Sort

The bubble sort algorithm Compare adjacent elements. If the first is greater than the second, swap them. This is done for every pair of adjacent elements starting from first two elements to last two elements. At the end of pass1 greatest element takes its proper place. The whole process is repeated except for the last one so that at each pass the comparisons become fewer.

Algorithm

```
        BubbleSort(A, n) {
```

```
for(i = 0; i <n-1; i++) {

    for(j = 0; j < n-i-1; j++) {

        if(A[j] > A[j+1]) {

            temp = A[j]; A[j] = A[j+1];

            A[j+1] = temp;

        } }}}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ………………………… +2 +1

$$= O(n^2)$$

There is no best-case linear time complexity for this algorithm.

Space Complexity:

Since no extra space besides 3 variables is needed for sorting

Space complexity = O(n)

Selection Sort

Idea: Find the least (or greatest) value in the array, swap it into the leftmost (or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly.

Algorithm:

```
SelectionSort(A) {

    for( i = 0;i < n ;i++){

        least=A[i];
```

```
                    p=i;

            for ( j = i + 1;j < n ;j++) {

                    if (A[j] < least) {

                        least= A[j];

                        p=j;

                    }

            }

        swap(A[i],A[p]);

    }
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ……………………… +2 +1

$$= O(n^2)$$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

Space complexity = O(n)

Insertion Sort

Idea: like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in

the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted

```
5    2    4    6    1    3

2    5    4    6    1    3

2    4    5    6    1    3

2    4    5    6    1    3

1    2    4    5    6    3

1    2    3    4    5    6
```
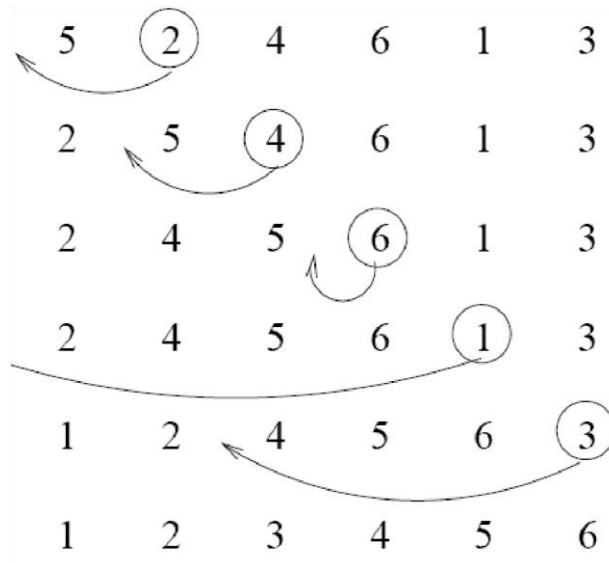
Algorithm:

InsertionSort(A) {

    for (i=1;i<n;i++) {

        key = A[i]

         for(j=i; j>0 && A[j] >key; j--) {

             A[j + 1] = A[j]

             }

        A[j + 1] = key

    }}
```

**Time Complexity:**

    **Worst Case Analysis:**

    Array elements are in reverse sorted order

Inner loop executes for 1 time when i=1, 2 times when i=2... and n-1 times when i=n-1:

Time complexity = 1 + 2 + 3 + …………………………. +(n-2) +(n-1)

$$= O(n^2)$$

**Best case Analysis:**

Array elements are already sorted

Inner loop executes for 1 time when i=1, 1 time when i=2... and 1 time when i=n-1:

Time complexity = 1 + 2 + 3 + …………………………. +1 +1

$$= O(n)$$

**Space Complexity:**

Since no extra space besides 5 variables is needed for sorting Space

complexity = $O(n)$

# Unit 3

**Divide and Conquer Algorithms**

**Searching Algorithms: Binary Search**

To find a key in a large _le containing keys z[0; 1; : : : ; n-1] in sorted order, we first compare key with z[n/2], and depending on the result we recurse either on the first half of the file, z[0; : : : ; n/2 - 1], or on the second half, z[n/2; : : : ; n-1].

**Algorithm**

```
BinarySearch(A,l,r, key)

{
if(l= = r)
        {
         if(key = = A[l])
         return l+1; //index starts from 0
        else
         return 0;
        }


        else

        {
         m = (l + r) /2; //integer division
         if(key = = A[m]
```

```
            return m+1;

            else if (key < A[m])

            return BinarySearch(l, m-1, key) ;

            else

            return BinarySearch(m+1, r, key) ; }

        }
```

Analysis:

From the above algorithm we can say that the running time of the algorithm

$$\text{is: } T(n) = T(n/2) + O(1)$$

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

In the best-case output is obtained at one run i.e., O (1) time if the key is at middle. In the worst case the output is at the end of the array so running time is O(logn) time. In the average case also running time is O(logn). For unsuccessful search best, worst and average time complexity is O(logn).

Min-Max Finding

Main idea behind the algorithm is: if the number of elements is 1 or 2 then max and min are obtained trivially. Otherwise, split problem into approximately equal part and solved recursively.

MinMax(l,r)

```
{
if(l = = r)
max = min = A[l];
else if(l = r-1)
{
if(A[l] < A[r])
{
```

```
 max = A[r];

min = A[l];

}

else

{

 max = A[l];

 min = A[r];

}}


else {        //Divide the problems mid = (l + r)/2; //integer division //solve the subproblems

Mid = (l+r)/2;

{ min,max}=MinMax(l,mid);

{ min1,max1}= MinMax(mid +1,r);

//Combine the solutions
if(max1 > max)
 max = max1;
if(min1 < min)
 min = min1;
}

}
```

Analysis:

We can give recurrence relation as below for MinMax algorithm in terms of number of comparisons.

$T(n) = 2T(n / 2) + 1$, if n>2

$T(n) = 1$, if n ≤2

Solving the recurrence by using master method complexity is (case 1) O(n).

Sorting Algorithms

Merge Sort

To sort an array A[l . . r]:

- Divide
    - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each
- Conquer
    - Sort the subsequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do
- Combine
    - Merge the two sorted subsequences

Divide

Merging:

	1	2	3	4	5	6	7	8	9	10	11
	1	2	2	3	4	4	5	6	6	7	7

Left branch: positions 1-6 → 1 2 4 4 6 7; positions 7-11 → 2 3 5 6 7

Next level: positions 1-3 → 2 4 7; positions 4-6 → 1 4 6; positions 7-9 → 3 5 7; positions 10-11 → 2 6

Next level: positions 1-2 → 4 7; position 3 → 2; positions 4-5 → 1 6; position 6 → 4; positions 7-8 → 3 7; position 9 → 5; position 10 → 2; position 11 → 6

Bottom: position 1 → 4; position 2 → 7; position 4 → 6; position 5 → 1; position 7 → 7; position 8 → 3

MergeSort(A, l, r)

{

    If (l < r)

    {   //Check for base case

      m = ⌊(l + r)/2⌋   //Divide

         MergeSort(A, l, m)   //Conquer

```
                MergeSort(A, m + 1, r)        //Conquer
                Merge(A, l, m+1, r)           //Combine
        }
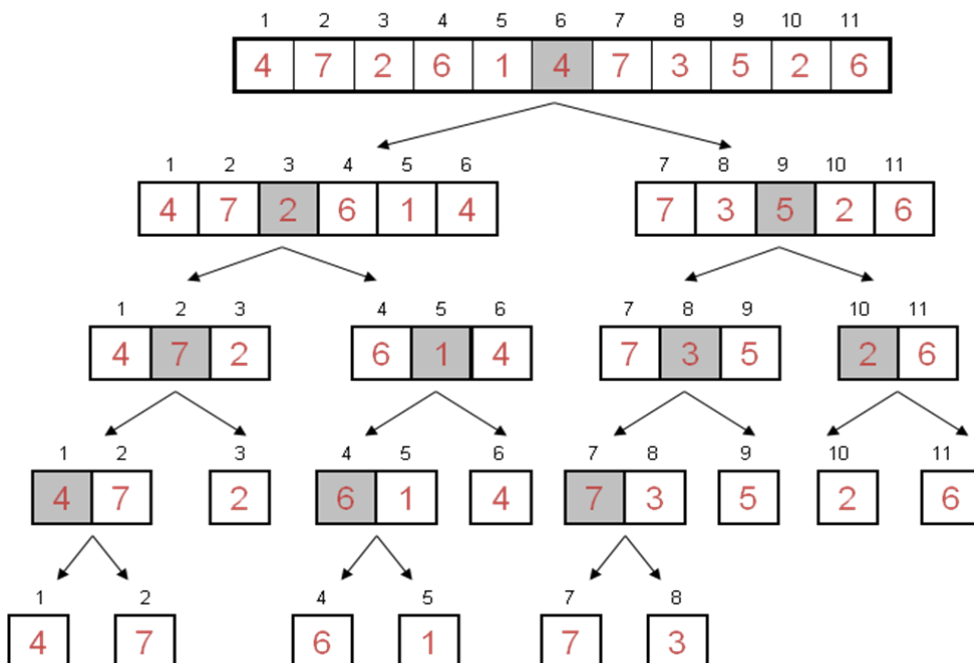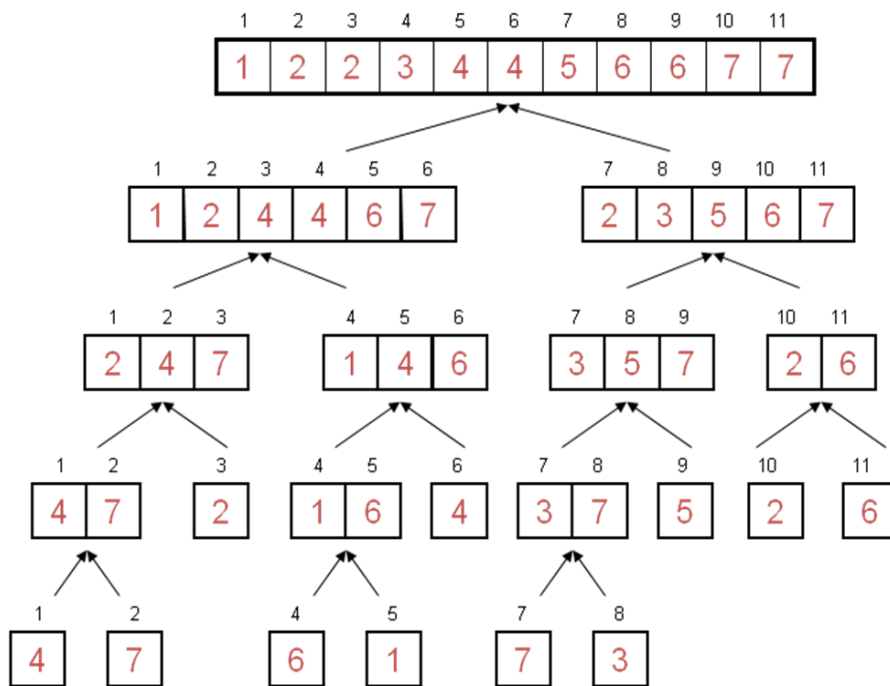
}



Merge(A,B,l,m,r)
{
        x=l,       y=m;
        k=l;
         while(x<m && y<r)

        {
                if(A[x] < A[y])
                {
                        B[k]= A[x];
                        k++; x++;
                } else
                {
                        B[k] = A[y];
                        k++; y++;
                }
        }
        while(x<m)
        {
                A[k] = A[x];
                k++; x++;
        } while(y<r)
        {
```

```
                        A[k] = A[y];

                        k++; y++;

                }

                for(i=l;i<= r; i++)

                {

                        A[i] = B[i]

                }

        }
```

Time Complexity:

Recurrence Relation for Merge sort:

T(n) = 1   if n=1

T(n) = 2 T(n/2) + O(n)   if n>1

Solving this recurrence we get

Time Complexity = O(nlogn)

Space Complexity:

It uses one extra array and some extra variables during sorting, therefore

Space Complexity= 2n + c = O(n)

Quick Sort

Partition the array A[l...r] into 2 subarrays A[l..m] and A[m+1..r], such that each element of A[l..m] is smaller than or equal to each element in A[m+1..r]. Need to find index p to partition the array.

- **Conquer**

Recursively sort A[p..q] and A[q+1..r] using Quicksort

- **Combine**

Trivial: the arrays are sorted in place. No additional work is required to combine them.

5	3	2	6	4	1	3	7	
x						y		

5	3	2	6	4	1	3	7	
			x			y		{swap x & y}

5	3	2	3	4	1	6	7	
					y	x		{swap y and pivot}

1	3	2	3	4	5	6	7
				p			

Algorithm:

QuickSort(A,l,r)

```
{ if(l<r)
        {   p   =   Partition(A,l,r);
                QuickSort(A,l,p-1);
                QuickSort(A,p+1,r);

        }

}

Partition(A,l,r)

{ x =l; y =r ; p = A[l];
while(x<y)
{ do { x++; }while(A[x] <=
        p); do {

                y--;

        } while(A[y] >=p);

        if(x<y)

                swap(A[x],A[y]);

}

A[l] = A[y]; A[y] = p; return y;   //return position of pivot }
```

Time Complexity:

We can notice that complexity of partitioning is $O(n)$ because outer while loop executes cn times.

Thus recurrence relation for quick sort is:

$T(n) = T(k) + T(n-k-1) + O(n)$ **Best**

Case:

Divides the array into two partitions of equal size, therefore $T(n)$

$= T(n/2) + O(n)$, Solving this recurrence we get,

⇨ Time Complexity = O(nlogn)

$\Theta(n \lg n)$

When array is already sorted or sorted in reverse order, one partition contains n-1 items and another contains zero items, therefore

T(n) = T(n-1) + O(1), Solving this recurrence we get

⇨ Time Complexity = O(n²)

$\Theta(n^2)$

Case between worst and best:

9-to-1 partitions split

T(n) = T(n=9n/10) + T(n/10) + O(n), Solving this recurrence we get

Time Complexity = O(nologn)

$$
n \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\!\blacktriangleright \quad n
$$

$$
\frac{1}{10}n \qquad\qquad \frac{9}{10}n \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\!\blacktriangleright \quad n
$$

$$
\frac{1}{100}n \quad \frac{9}{100}n \qquad \frac{9}{100}n \quad \frac{81}{100}n \quad \cdots\cdots\cdots\!\blacktriangleright \quad n
$$

$$
\frac{81}{1000}n \quad \frac{729}{1000}n \quad \cdots\cdots\!\blacktriangleright \quad n
$$

$$
\cdots\cdots\!\blacktriangleright \ \leq n
$$

$$
1 \quad \cdots\cdots\!\blacktriangleright \ \leq n
$$

$\log_{10} n$

$\log_{10/9} n$

1

$\Theta(n \lg n)$

Heap Sort

A **heap** is a nearly complete binary tree with the following two properties:

— **Structural property:** all levels are full, except possibly the last one, which is filled from left to right

— **Order (heap) property:** for any node x, Parent(x) ≥ x

Array Representation of Heaps

A heap can be stored as an array *A*. −

Root of tree is A[1]

— Left child of A[i] = A[2i]

— Right child of A[i] = A[2i + 1]

— Parent of A[i] = A[⌊i/2⌋]

— Heapsize[A] ≤ length[A]

The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves

Max-heaps (largest element at root), have the max-heap property:

— for all nodes i, excluding the root:

A[PARENT(i)] ≥ A[i]

Min-heaps (smallest element at root), have the min-heap property:

— for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

Adding/Deleting Nodes

New nodes are always inserted at the bottom level (left to right) and nodes are removed from the bottom level (right to left).

Operations on Heaps

➢ Maintain/Restore the max-heap property
   − MAX-HEAPIFY
➢ Create a max-heap from an unordered array
   − BUILD-MAX-HEAP
➢ Sort an array in place
   − HEAPSORT
➢ Priority queues

Maintaining the Heap Property

Suppose a node is smaller than a child and Left and Right subtrees of i are max-heaps. To eliminate the violation:

   —   Exchange with larger child

   —   Move down the tree

   —   Continue until node is not smaller than children

Algorithm:

Max-Heapify(A, i, n)

{ l = Left(i) r = Right(i) largest=i; **if** l ≤

    n   and   A[l]   >   A[largest]

    largest = l

    if r ≤ n and A[r] > A[largest] largest

      = r

if largest ≠ i   exchange (A[i] ,

A[largest])

Max-Heapify(A, largest, n)

}

Analysis:

In the worst case Max-Heapify is called recursively h times, where h is height of the heap and since each call to the heapify takes constant time

Time complexity = O(h) = O(logn)

Building a Heap

Convert an array A[1 … n] into a max-heap (n = length[A]). The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves. Apply MAX-HEAPIFY on elements between 1 and ⌊n/2⌋.

Algorithm:

Build-Max-Heap(A) n = length[A]

    for i ← n/2 **downto** 1

      do MAX-HEAPIFY(A, i, n)

Time Complexity:

Running time: Loop executes $O(n)$ times and complexity of Heapify is $O(\lg n)$, therefore complexity of Build-Max-Heap is $O(n\log n)$.

This is not an asymptotically tight upper bound

   Heapify takes $O(h)$

   $\Rightarrow$ The cost of Heapify on a node i is proportional to the height of the node i in the tree

   $\Rightarrow T(n) = \sum_{i=0}^{h} n_i\, h_i$

   $h_i = h - i$ height of the heap rooted at level i

   $n_i = 2^i$ number of nodes at level i

   $\Rightarrow T(n) = \sum_{i=0}^{h} 2^i (h-i)$

   $\Rightarrow T(n) = \sum_{i=0}^{h} 2^h (h-i) / 2^{h-i}$

   Let $k = h-i$

   $\Rightarrow T(n) = 2^h \sum_{i=0}^{h} k / 2^k$

   $\Rightarrow T(n) \leq n \sum_{i=0}^{\infty} k / 2^k$

   We know that, $\sum_{i=0}^{\infty} x^k = 1/(1-x)$ for $x<1$

   Differentiating both sides we get,

   $\sum_{i=0}^{\infty} k\, x^{k-1} = 1/(1-x)^2$

$$\sum_{i=0}^{\infty} k\, x^k = x/(1-x)^2$$

Put x=1/2

$$\sum_{i=0}^{\infty} k\, /2^k = 1/(1-x)^2 = 2$$

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

- Build a max-heap from the array

- Swap the root (the maximum element) with the last element in the array

- "Discard" this last node by decreasing the heap size

- Call Max-Heapify on the new root

- Repeat this process until only one node remains

Heapify(A,1)

Heapify(A,1)

Heapify(A,1)

Algorithm:

```
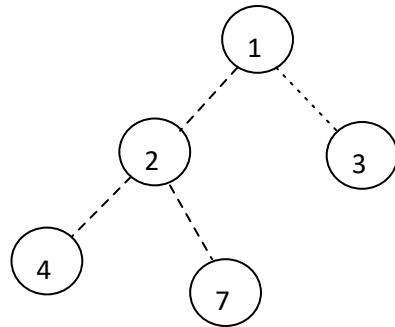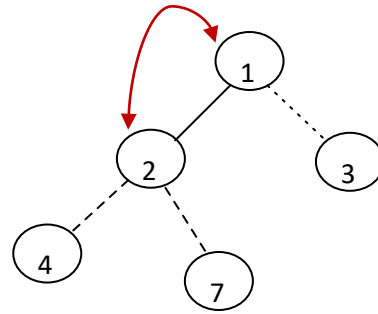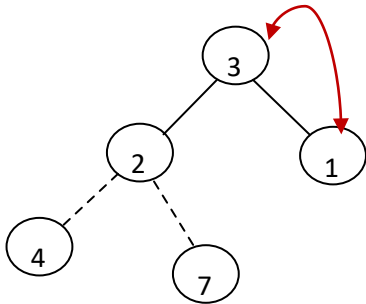HeapSort(A)
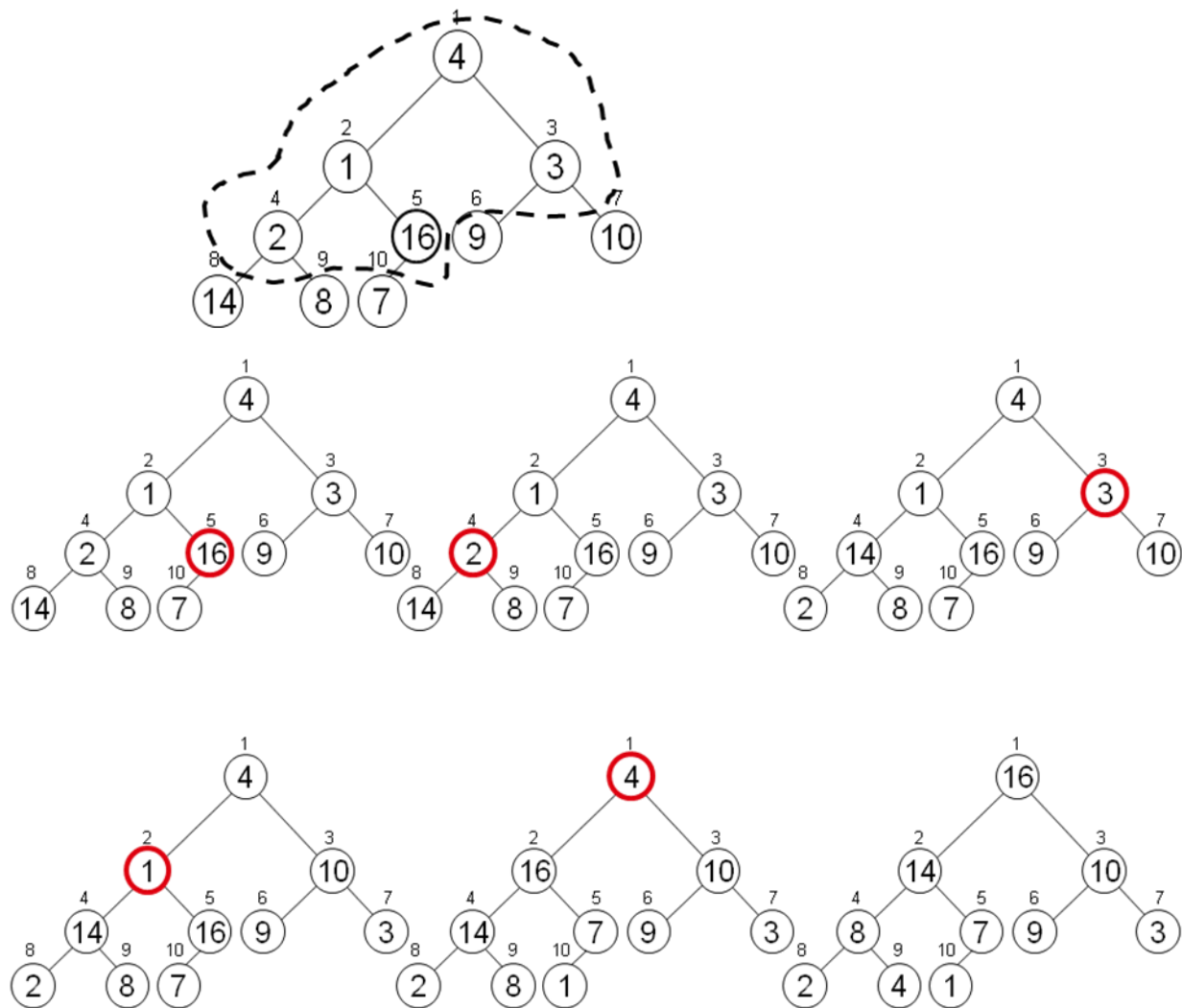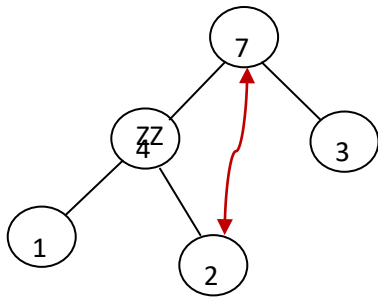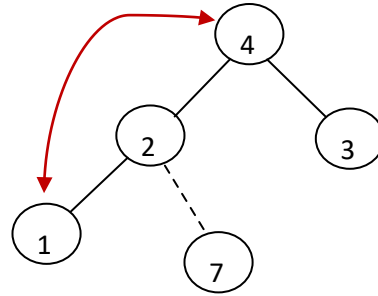
{

        BuildHeap(A); //into max heap

        n = length[A];

        for(i = n ; i >= 2; i--)

        {

        swap(A[1],A[n]);

        n = n-1;

        Heapify(A,1);

        }

}
```

Randomized Quick sort

The algorithm is called randomized if its behavior depends on input as well as random value generated by random number generator. The beauty of the randomized algorithm is that no particular input can produce worst-case behavior of an algorithm. IDEA: Partition around a random element. Running time is independent of the input order. No assumptions need to be made about the input distribution. No specific input elicits the worst-case behavior. The worst case is determined only by the output of a random-number generator. Randomization cannot eliminate the worst-case but it can make it less likely!

Algorithm:

```
RandQuickSort(A,l,r)

{

    if(l<r)

    {
```

```
        m = RandPartition(A,l,r);
    RandQuickSort(A,l,m-1);
        RandQuickSort(A,m+1,r);

    }

}
```

RandPartition(A,l,r)

{   k = random(l,r); //generates random number between i and j including both.

   swap(A[l],A[k]);

   return Partition(A,l,r);

}

Time Complexity:

   Worst Case:

   $T(n)$ = worst-case running time

   $T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + O(n)$

         Use substitution method to show that the running time of Quicksort is $O(n^2)$

   Guess $T(n) = O(n^2)$

      $-$ Induction goal: $T(n) \le cn^2$

      $-$ Induction hypothesis: $T(k) \le ck^2$ for any $k < n$ Proof of induction

   goal:

      $T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + O(n)$

$$= c \ \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + O(n)$$

The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints

$$\max_{1 \le q \le n-1} (q^2 + (n - q)^2) = 1^2 + (n - 1)^2 = n^2 - 2(n - 1)$$

$$T(n) \le cn^2 - 2c(n - 1) + O(n)$$

$$\le cn^2$$

Order Statistics:

i^{th} order statistic of a set of elements gives i^{th} largest(smallest) element. In general lets think of ith order statistic gives i^{th} smallest. Then minimum is first order statistic and the maximum is last order statistic. Similarly a median is given by i^{th} order statistic where $i = (n+1)/2$ for odd n and $i = n/2$ and $n/2 + 1$ for even n. This kind of problem commonly called selection problem.

This problem can be solved in $\Theta(n\log n)$ in a very straightforward way. First sort the elements in $\Theta(n\log n)$ time and then pick up the ith item from the array in constant time. What about the linear time algorithm for this problem? The next is answer to this.

Selection in Expected Linear Time,

This problem is solved by using the "divide and conquer" method. The main idea for this problem solving is to partition the element set as in Quick Sort where partition is randomized one.

Algorithm:

RandSelect(A,l,r,i)

{

```
if(l = =r )

return A[p];

p = RandPartition(A,l,r);

 k = p – l + 1;

 if(i <= k)

return RandSelect(A,l,p-1,i);

else

return RandSelect(A,p+1,r,i - k);

}
```

Time Complexity:

Worst Case:

T(n) = worst-case running time

$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \Theta(n)$

         Use substitution method to show that the running time of Quicksort is $O(n^2)$

Guess $T(n) = O(n^2)$

- Induction goal: $T(n) \leq cn^2$
- Induction hypothesis: $T(k) \leq ck^2$ for any $k < n$

Proof of induction goal:

     $T(n) \leq \max_{1 \leq q \leq n-1} (cq^2 + c(n\text{-}q)^2) + \Theta(n)$

         $= c \cdot \max_{1 \leq q \leq n-1} (q^2 + (n\text{-}q)^2) + \Theta(n)$

The expression $q^2 + (n\text{-}q)^2$ achieves a maximum over the range $1 \leq q \leq n\text{-}1$ at one of the endpoints

     $\max_{1 \leq q \leq n-1} (q^2 + (n - q)^2) = 1^2 + (n - 1)^2 = n^2 - 2(n - 1)$

       $T(n) \leq cn^2 - 2c(n - 1) + \Theta(n)$

         $\leq cn^2$

Average Case:

To analyze average case, assume that all the input elements are distinct for simplicity. If we are to take care of duplicate elements also the complexity bound is same but it needs more intricate analysis. Consider the probability of choosing pivot from n elements is equally likely i.e. 1/n.

Now we give recurrence relation for the algorithm as

$$T(n) = 1/n \sum_{k=1}^{n-1} (T(k) + T(n-k)) + O(n)$$

For some k = 1,2, ..., n-1,   T(k) and T(n-k) is repeated two times

$$T(n) = 2/n \sum_{k=1}^{n-1} T(k) + O(n)$$

$$nT(n) = 2 \sum_{k=1}^{n-1} T(k) + O(n^2)$$

Similarly

$$(n-1)T(n-1) = 2 \sum_{k=1}^{n-2} T(k) + O(n-1)^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$

$$nT(n) - (n+1) T(n-1) = 2n-1$$

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

Let $A_n = T(n)/(n+1)$

$\Rightarrow$ $A_n = A_{n-1} + (2n+1)/n(n-1)$

$\Rightarrow$ $An = \sum_{i=1}^{n} 2i - 1/i(i+1)$

$\Rightarrow$ $An \approx \sum_{i=1}^{n} 2i/i(i+1)$

$\Rightarrow$ $An \approx 2\sum_{i=1}^{n} 1/(i+1)$

$\Rightarrow$ $An \approx 2\log n$

Since $An = T(n)/(n+1)$

$T(n) = n\log n$

Selection in Worst Case Linear Time

Divide the n elements into groups of 5. Find the median of each 5-element group. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

Algorithm

   Divide *n* elements into groups of 5

   Find median of each group

   Use Select() recursively to find median *x* of the ⌊n/5⌋ medians

   Partition the *n* elements around *x*. Let *k* = rank(*x*) //index of x **if** (i == k) **then** return x **if** (i
      < k) **then** use Select() recursively to find *i*th smallest element in first partition **else**
      (i > k) use Select() recursively to find (*i-k*)th smallest element in last partition

At least half the group medians are ≤x, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

Therefore, at least $3\lfloor n/10 \rfloor$ elements are ≤x.

Similarly, at least $3\lfloor n/10 \rfloor$ elements are ≥x

For n≥50, we have $3\lfloor n/10 \rfloor \geq n/4$.

Therefore, for n≥50the recursive call to SELECT in Step 4 is executed recursively on ≤3n/4elements in worst case.

Thus, the recurrence for running time can assume that Step 4 takes time T(3n/4)in the worst case.

Now, We can write recurrence relation for above algorithm as"

$$T(n) = T(n/5) + T(3n/4) + \Theta(n)$$

Guess T(n) = O(n)

To Show T(n) ≤ cn

Assume that our guess is true for all k<n

Now,

$$T(n) \le cn/5 + 3cn/4 + O(n)$$

$$= 19cn/20 + O(n)$$

$$= cn - cn/20 + O(n)$$

$$\le cn \quad \{ \text{Choose value of c such that } cn/20 - O(n) \le 0 \}$$

$$T(n) = O(n)$$

Unit 4
Greedy Algorithms

Optimization Problems and Optimal Solution

Introduction of Greedy Algorithms

In many optimization algorithms a series of selections need to be made. In dynamic programming we saw one way to make these selections. Namely, the optimal solution is described in a recursive manner, and then is computed "bottom-up". Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times. Greedy method typically leads to simpler and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming. Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (non-optimal solution strategies), are often used in finding good approximations.

Elements of Greedy Strategy.

To prove that a greedy algorithm is optimal we must show the following two characteristics are exhibited.

➡ Greedy Choice Property

➡ Optimal Substructure Property

Greedy Algorithms:

Fractional Knapsack

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of ith item is w_i and it worth v_i. Any amount of item can be put into the bag i.e. x_i fraction of item can be collected, where $0 <= x_i <= 1$. Here the objective is to collect the items that maximize the total profit earned.

Algorithm

Take as much of the item with the highest value per weight (v_i/w_i) as you can. If the item is finished then move on to next item that has highest (v_i/w_i), continue this until the knapsack is full. $v[1 … n]$ and $w[1 … n]$ contain the values and weights respectively of the n objects sorted in non increasing ordered of $v[i]/w[i]$. W is the capacity of the knapsack, $x[1 … n]$ is the solution vector that includes fractional amount of items and n is the number of items.

```
GreedyFracKnapsack(W,n)

{ for(i=1; i<=n; i++)

            x[i] = 0.0; tw

            = W;
        for(i=1; i<=n; i++)

        { if(w[i] > tw)  break;

            else   x[i]   =
            1.0;


            tempw -= w[i];

        } if(i<=n)  x[i] =
        tw/w[i];

    }
```

Analysis:

We can see that the above algorithm just contain a single loop i.e. no nested loops the running time for above algorithm is O(n). However our requirement is that $v[1 … n]$ and $w[1 … n]$ are sorted, so we can use sorting method to sort it in O(nlogn) time such that the complexity of the algorithm above including sorting becomes O(nlogn).

Job sequencing with Deadlines

We are given a set of n jobs. Associated with each job I, $d_i \geq 0$ is an integer deadline and $p_i \geq 0$ is profit. For any job i profit is earned iff job is completed by deadline. To complete a job one has to process a job for one unit of time. Our aim is to find feasible subset of jobs such that profit is maximum.

Example

n=4, (p_1,p_2,p_3,p_4)=(100,10,15,27), (d_1,d_2,d_3,d_4)=(2,1,2,1) n=4,

(p1,p4,p3,p2)=(100,27,15,10),   (d1,d4,d3,d2)=(2,1,2,1)

Feasible Solution	processing sequence	value
1. (1, 2)	2, 1	110
2. (1, 3)	1, 3 or 3, 1	115
3. (1, 4)	4, 1	127
4. (2, 3)	2, 3	25
5. (3, 4)	4, 3	42
6. (1)	1	100
7. (2)	2	10
8. (3)	3	15
9. (4)	4	27

We have to try all the possibilities, complexity is O(n!).

Greedy strategy using *total profit* as optimization function to above example. Begin with J=∅

- — Job 1 considered, and added to J ∅ J={1}

- — Job 4 considered, and added to J ∅ J={1,4}

- — Job 3 considered, but discarded because not feasible ∅ J={1,4}

- Job 2 considered, but discarded because not feasible ▢ J={1,4} Final solution is J={1,4} with total profit 127 and it is optimal

Algorithm

Assume the jobs are ordered such that p[1]▢p[2]▢...▢p[n] d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs n are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i] is the ith job in the optimal solution, 1<=i<=k.

```
JobSequencing(int d[], int j[], int n)

{

        for(i=1;i<=n;i++)

        {//initially no jobs are selected

                J[i]=0;

        }
        for (int i=1; i<=n;  i++)
        {

                d=d[i];

                for(k=d;k>=0;k--)

                {

                        if(j[k]==0)

                        {

                                J[k]=i;

                        }

                }

        }
```

```
                  }
```

Analysis

First for loop executes for O(n) times .

In case of second loop outer for loop executes O(n) times and inner for loop executes for at most O(n) times in the worst case. All other statements takes O(1) time. Hence total time for each iteration of outer for loop is O(n) in worst case.

Thus time complexity= $O(n) + O(n^2) = O(n^2)$.

Kruskal's Algorithm

Kruskal's Algorithm: The problem of finding MST can be solved by using Kruskal's algorithm. The idea behind this algorithm is that you put the set of edges form the given graph G = (V,E) in nondecreasing order of their weights. The selection of each edge in sequence then guarantees that the total cost that would from will be the minimum. Note that we have G as a graph, V as a set of n vertices and E as set of edges of graph G.

Example:

Find the MST and its weight of the graph.

Solution:

Algorithm:

KruskalMST(G)

{

   T = {V} // forest of n nodes
   S = set of edges sorted in nondecreasing order of weight
   while(|T| < n-1 and E !=Ø)
   {

      Select (u,v) from S in order
      Remove (u,v) from E
      if((u,v) doesnot create a cycle in T))

$$T = T \cup \{(u,v)\}$$
$$\}$$
$$\}$$

Analysis:
In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes O(ElogE) time and while loop execute O(n) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(ElogE) or asymptotically equivalently O(ElogV)!.

Prims Algorithm

This is another algorithm for finding MST. The idea behind this algorithm is just take any arbitrary vertex and choose the edge with minimum weight incident on the chosen vertex. Add the vertex and continue the above process taking all the vertices added. Remember the cycle must be avoided.

Example:

Find the minimum spanning tree of the following graph.

Solution: note: dotted edge is chosen.

The total weight of MST is 64.

Algorithm:

    PrimMST(G)

    {

        T = Ø; // T is a set of edges of MST
        S = {s} ; //s is randomly chosen vertex and S is set of vertices
        while(S != V)
        {
        e = (u,v) an edge of minimum weight incident to vertices in T and not forming a
        simple circuit in T if added to T i.e. u ∈ S and v∈ V-S
        T = T ∪ {(u,v)};
        S = S ∪ {v};
        }

    }

Analysis:

In the above algorithm while loop execute $O(V)$. The edge of minimum weight incident on a vertex can be found in $O(E)$, so the total time is $O(EV)$. We can improve the performance of the above algorithm by choosing better data structures as priority queue and normally it will be seen that the running time of prim's algorithm is $O(ElogV)$!.

Dijkstra's Algorithm

This is another approach of getting single source shortest paths. In this algorithm it is assumed that there is no negative weight edge. Dijkstra's algorithm works using greedy approach, as we will see later.

Example:

Find the shortest paths from the source g to all other vertices using Dijkstra's algorithm.

There will be no change for vertices b and d. continue above steps for b and d to complete. The tree is shown as dark connection.

Algorithm:

Dijkstra(G,w,s)
{
    for each vertex v∈ V
        do d[v] = ∞
        d[s] = 0
        S = Ø
        Q = V
        While(Q!= Ø)
        {
        u = Take minimum from Q and delete.
        S = S ∪ {u}
        for each vertex v adjacent to u

            do if d[v] > d[u] + w(u,v)
                then d[v] = d[u] + w(u,v)

        }

}

Analysis:

In the above algorithm, the first for loop block takes O(V) time. Initialization of priority queue Q takes O(V) time. The while loop executes for O(V), where for each execution the block inside the loop takes O(V) times. Hence the total running time is $O(V^2)$.

Huffman Coding

Huffman coding is an algorithm for the lossless compression of files based on the frequency of occurrence of a symbol in the file that is being compressed. In any file, certain characters are used more than others. Using binary representation, the number of bits required to represent each character depends upon the number of characters that have to be represented. Using one bit we can represent two characters, i.e., 0 represents the first character and 1 represents the

second character. Using two bits we can represent four characters, and so on. Unlike ASCII code, which is a fixed-length code using seven bits per character, Huffman compression is a variable-length coding system that assigns smaller codes for more frequently used characters and larger codes for less frequently used characters in order to reduce the size of files being compressed and transferred.

For example, in a file with the following data: 'XXXXXXYYYYZZ'. The frequency of "X" is 6, the frequency of "Y" is 4, and the frequency of "Z" is 2. If each character is represented using a fixed-length code of two bits, then the number of bits required to store this file would be 24, i.e., (2 x 6) + (2x 4) + (2x 2) = 24. If the above data were compressed using Huffman compression, the more frequently occurring numbers would be represented by smaller bits, such as: X by the code 0 (1 bit), Y by the code 10 (2 bits) and Z by the code 11 (2 bits), the size of the file becomes 18, i.e., (1x 6) + (2 x 4) + (2 x 2) = 18. In this example, more frequently occurring characters are assigned smaller codes, resulting in a smaller number of bits in the final compressed file.
Huffman compression was named after its discoverer, David Huffman.

To generate Huffman codes we should create a binary tree of nodes. Initially, all nodes are leaf nodes, which contain the **symbol** itself, the **weight** (frequency of appearance) of the symbol. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n - 1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths. The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree. The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

Example

The following example bases on a data source using a set of five different symbols. The symbol's frequencies are:

Symbol	Frequency
A	24
B	12
C	10
D	8
E	8

----> total 186 bit (with 3 bit per code word)

Step1:

| E | 8 | | D | 8 | | C | 10 | | B | 12 | | A | 24 |

Step2:

| C | 10 | | B | 12 | (16) | A | 24 |

E 8   D 8

Step3:

(16) (22) | A | 24 |

E 8   D 8   C 10   B 12

Step4:

Step5:

Symbol	Frequency	Code	Code length	total Length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

Total length of message: 138 bit

Algorithm

A greedy algorithm can construct Huffman code that is optimal prefix codes. A tree corresponding to optimal codes is constructed in a bottom up manner starting from the |C| leaves and |C|-1 merging operations. Use priority queue Q to keep nodes ordered by frequency. Here the priority queue we considered is binary heap.

HuffmanAlgo(C)

```
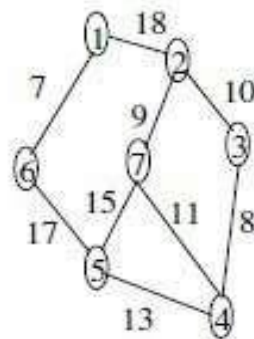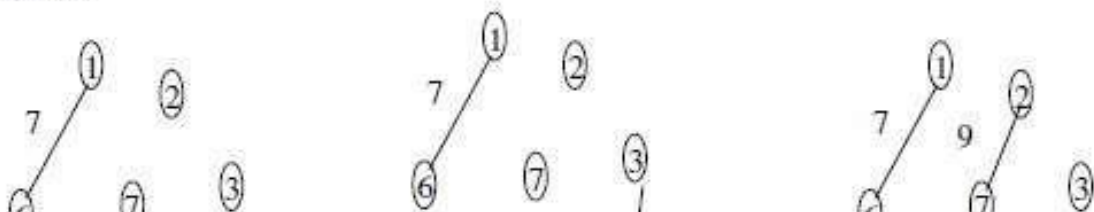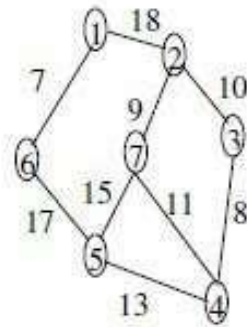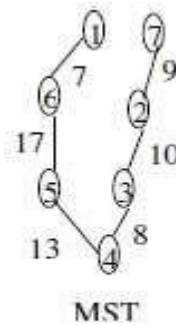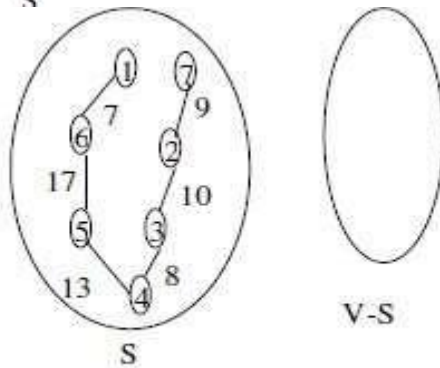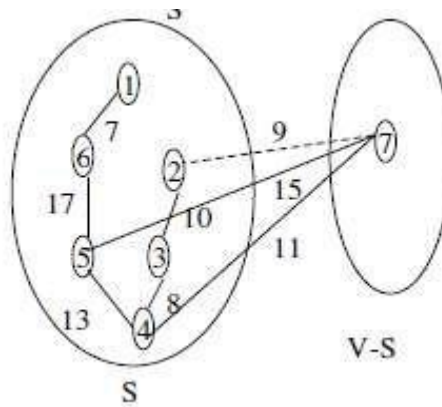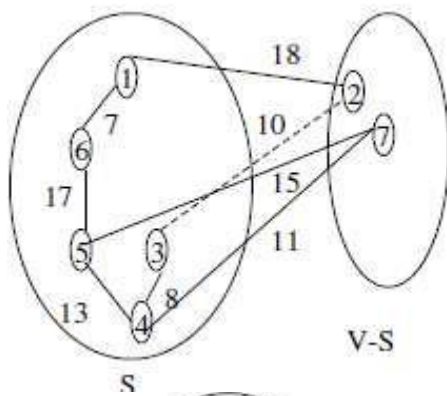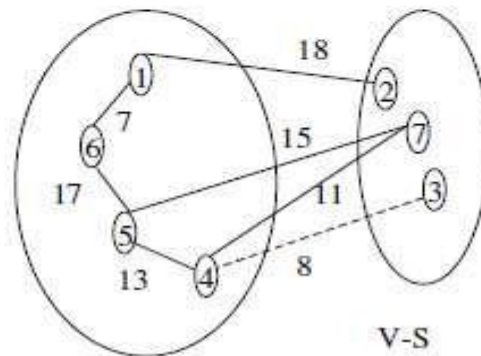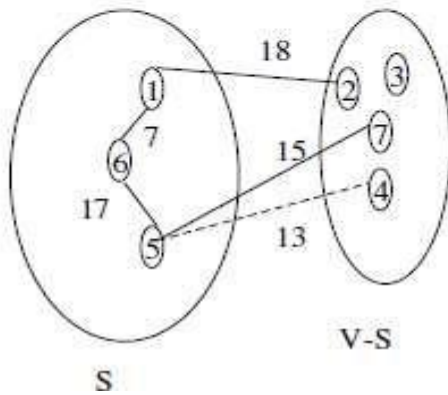{
        n = |C|;   Q = C;          n le chai complete set of characters haru ko size leko
                                    Q le chai set of characters liraxa sabai
        For(i=1; i<=n-1; i++)      yeha sayed integer haru xa k value haru hae
        {
                z = Allocate-Node();
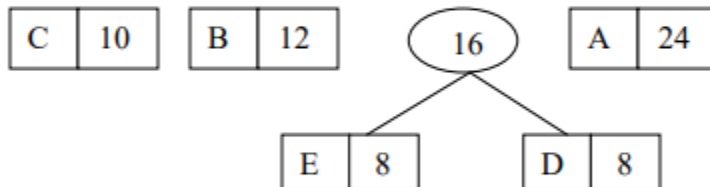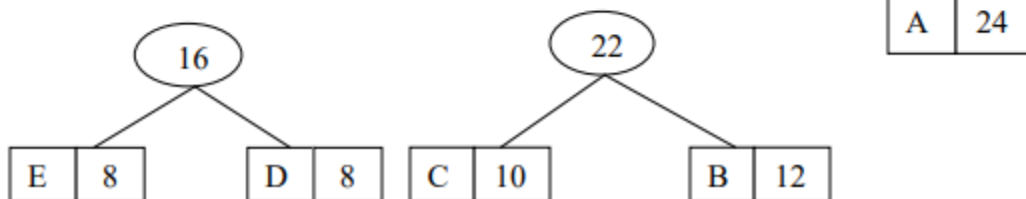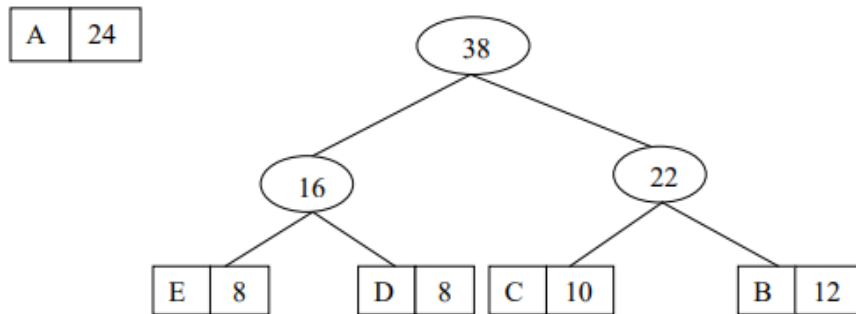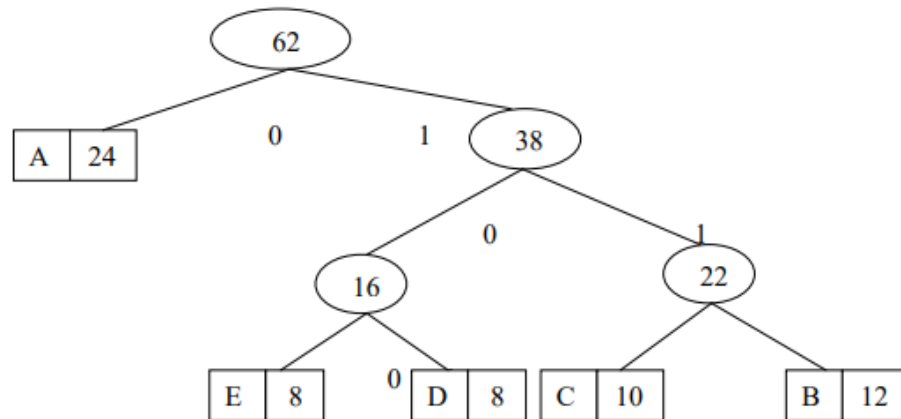
                x = Extract-Min(Q);

                y = Extract-Min(Q);

                left(z) = x;   right(z) = y;

                f(z) = f(x) + f(y);

                Insert(Q,z);

        }

}
```

Analysis

We can use BuildHeap(C) to create a priority queue that takes $O(n)$ time. Inside the for loop the expensive operations can be done in $O(\log n)$ time. Since operations inside for loop executes for n-1 time total running time of Huffman algorithm is $O(n \log n)$.

Unit 5

Dynamic Programming

Greedy Algorithms vs Dynamic Programming

Feature	Greedy method	Dynamic programming
Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution.
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
Recursion	A greedy method follows the problem-solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.

Feature	Greedy method	Dynamic programming
Memoization	It is more efficient in terms of memory as it never looks back or revise previous choices	It requires dp table for memorization and it increases it's memory complexity.
Time complexity	Greedy methods are generally faster. For example, Dijkstra's shortest path algorithm takes O(ELogV + VLogV) time.	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.
Example	Fractional knapsack.	0/1 knapsack problem

Recursion vs Dynamic Programming

Recursion is a way of finding the solution by expressing the value of a function in terms of other values of that function directly or indirectly and such function is called a recursive function. It follows a top-down approach.

Dynamic programming is nothing but recursion with memoization i.e. calculating and storing values that can be later accessed to solve subproblems that occur again, hence making your code faster and reducing the time complexity (computing CPU cycles are reduced).

Here, the basic idea is to save time by efficient use of space. Recursion takes time but no space while dynamic programming uses space to store solutions to subproblems for future reference thus saving time.

Understanding Dynamic Programming with Examples

Fibonacci series is a sequence of numbers in such a way that each number is the sum of the two preceding ones, starting from 0 and 1.

$F(n) = F(n-1) + F(n-2)$

Recursive method:

```
def r_fibo(n):
  if n <= 1:
    return n
  else:
    return(r_fibo(n-1) + r_fibo(n-2))
```

Here, the program will call itself, again and again, to calculate further values. The calculation of the time complexity of the recursion based approach is around $O(2^N)$. The space complexity of this approach is O(N) as recursion can go max to N.

For example-

$F(4) = F(3) + F(2) = ((F(2) + F(1)) + F(2) = ((F(1) + F(0)) + F(1)) + (F(1) + F(0))$

In this method values like F(2) are computed twice and calls for F(1) and F(0) are made multiple times. Imagine the number of repetitions if you have to calculate it F(100). This method is **ineffective for large values.**

Top-Down Method

```
def fibo(n, memo):
 if memo[n] != null:
   return memo[n]
 if n <= 1:
   return n
 else:
   res = fibo(n-1) + fibo(n+1)
   memo[n] = res
   return res
```

Here, the computation time is reduced significantly as the outputs produced after each recursion are stored in a list which can be reused later. This method is much more efficient than the previous one.

Bottom down

```
def fib(n):
 if n<=1:
   return n
 list_ = [0]*(n+1)
 list_[0] = 0
 list_[1] = 1
 for i in range(2, n+1):
   list_[i] = list_[i-1] + list[i-2]
 return list_[n]
```

This code doesn't use recursion at all. Here, we create an empty list of length (n+1) and set the base case of F(0) and F(1) at index positions 0 and 1. This list is created to store the corresponding calculated values using a for loop for index values 2 up to n.

Unlike in the recursive method, the time complexity of this code is linear and takes much less time to compute the solution, as the loop runs from 2 to n, i.e., it runs in $O(n)$. This approach is the **most efficient way** to write a program.

Time complexity: $O(n) <<< O(2^n)$

Elements of DP Strategy

technique is among the most powerful for designing algorithms for optimization problems. Dynamic programming problems are typically optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose your problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.

Table-structure: Store the answers to the sub-problems in a table. This is done because subproblem solutions are reused many times.

Bottom-up computation: Combine solutions on smaller subproblems to solve larger subproblems. (We also discusses a top-down alternative, called memoization)

The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the formulation of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to be applicable.

Optimal substructure: (Sometimes called the principle of optimality.) It states that for the global problem to be solved optimally, each subproblem should be solved

optimally. (Not all optimization problems satisfy this. Sometimes it is better to lose a

little on one

subproblem in order to make a big gain on another.)

Polynomially many subproblems: An important aspect to the efficiency of DP is that

the total number of subproblems to be solved should be at most a polynomial number.

DP Algorithms: Matrix Chain Multiplication

Chain Matrix Multiplication Problem: Given a sequence of matrices $A_1; A_2; : : : ; A_n$ and dimensions $p_0; p_1; : : : ; p_n$, where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the number of operations.
Important Note: This algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

Although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A1 be 5 x 4, A2 be 4 x 6 and A3 be 6 x 2.

$$multCost[((A1A2)A3)] = (5 . 4 . 6) + (5 . 6 . 2) = 180$$
$$multCost[(A1(A2A3))] = (4 . 6 . 2) + (5 . 4 . 2) = 88$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.
Let $A_{i...j}$ denote the result of multiplying matrices i through j. It is easy to see that $A_{i...j}$ is a $p_{i-1} \times p_j$ matrix. So for some k total cost is sum of cost of computing $A_{i...k}$, cost of computing $A_{k+1...j}$, and cost of multiplying $A_{i...k}$ and $A_{k+1...j}$.

Recursive definition of optimal solution: let m[j,j] denotes minimum number of scalar multiplications needed to compute $A_{i...j}$.

$$C[i,w] = \begin{cases} 0 & \text{if } i{=}j \\ \min_{i \le k < j}\{m[I,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i{<}j \end{cases}$$

Algorithm:
Matrix-Chain-Multiplication(p)
```
{
        n =length[p]
        for( i= 1 i<=n i++)
        {



            m[i, i]= 0
    }
for(l=2; l<= n; l++)
{
    for( = 1; i<=n-l+1; i++)
    {
            j = i + l − 1
        m[i, j] = ∞
        for(k= i; k<= j-1; k++)
        {
                        c= m[i, k] + m[k + 1, j] + p[i−1] * p[k] * p[j]
                        if c < m[i, j]
                        {
                            m[i, j] = c
                            s[i, j] = k
                        }
        }
    }
}
return m and s
}
```

Analysis
The above algorithm can be easily analyzed for running time as $O(n^3)$, due to three nested loops.
The space complexity is $O(n^2)$.

Example:

Consider matrices A1, A2, A3 And A4 of order 3x4, 4x5, 5x2 and 2x3.

M Table(Cost of multiplication)

j \\ i	1	2	3	4
1	0	60	64	82
2		0	40	64
3			0	30
4				0

S Table (points of parenthesis)

j \\ i	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Constructing optimal solution

(A1A2A3A4) => ((A1A2A3)(A4)) => (((A1)(A2A3))(A4))

String Editing

Given two strings, s1 and s2 and edit operations (given below). Write an algorithm to find minimum number operations required to convert string s1 into s2.

Allowed Operations:

Insertion – Insert a new character.

Deletion – Delete a character.

Replace – Replace one character by another.

Example:

String s1 = "horizon"

String s2 = "horzon"

Output: 1 {remove 'i' from string s1}

String s1 = "horizon"

String s2 = "horizontal"

Output: 3 {insert 't', 'a', 'l' characters in string s1}

Approach:

Start comparing one character at a time in both strings. Here we are comparing string from right to left (backwards).

Now for every string we there are two options:

If last characters in both the strings are same then just ignore the character and solve the rest of the string recursively.

Else if last characters in both the strings are not same then we will try all the possible operations (insert, replace, delete) and get the solution for rest of the string recursively for each possibility and pick the minimum out of them.

Example:

X = EXECUTION

Y = INTENTION

-	0	E	X	E	C	U	T	I	O	N
0	0	1	2	3	4	5	6	7	8	9
I	1	1	2	3	4	5	6	6	7	8
N	2	2	2	3	4	5	6	7	7	7
T	3	3	3	3	4	5	5	6	7	8
E	4	3	4	3	4	5	6	6	7	8
N	5	4	4	4	4	5	6	7	7	7
T	6	5	5	5	5	5	5	6	7	8
I	7	6	6	6	6	6	6	5	6	7
O	8	7	7	7	7	7	7	6	5	6
N	9	8	8	8	8	8	8	7	6	5

Minimum Distance = 5

Algorithm

```
editDistDP(str1, str2, m, n)
{

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0)
                dp[i][j] = j;
```

```
        else if (j == 0)

            dp[i][j] = i;

        else if (str1[i - 1] == str2[j - 1])

            dp[i][j] = dp[i - 1][j - 1];

        else

            dp[i][j] = 1+ min(dp[i][j - 1], // Insert

                    dp[i - 1][j],    // Remove

                    dp[i - 1][j - 1]); // Replace

    }

  }


    return dp[m][n];

 }
```

Time Complexity: O(m x n)

Zero-One Knapsack Problem

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of ith item is w_i and it worth v_i. An amount of item can be put into the bag is 0 or 1 i.e. x_i is 0 or 1. Here the objective is to collect the items that maximize the total profit earned.

We can formally state this problem as, maximize $\sum_{i=1}^{n} x_i v_i$ Using the constraints $\sum_{i=1}^{n} x_i w_i \leq W$

The algorithm takes as input maximum weight W, the number of items n, two arrays v[] for values of items and w[] for weight of items. Let us assume that the table c[i,w] is the value of solution for items 1 to i and maximum weight w. Then we can define recurrence relation for 0/1 knapsack problem as

$$C[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ C[i-1,w] & \text{if } w_i > w \\ Max\{v_i + C[i-1,w-w_i], c[i-1,w]\} & \text{if } i>0 \text{ and } w>w_i \end{cases}$$

Algorithm:

DynaKnapsack(W,n,v,w)

{

    for(w=0; w<=W; w++)

        C[0,w]   =   0;

    for(i=1; i<=n; i++)

        C[i,0]   =   0;

    for(i=1; i<=n; i++)

    {

        for(w=1; w<=W;w++)

        { if(w[i]<w)

            {

            If(v[i] +C[i-1,w-w[i]] > C[i-1,w])

                {

                      C[i,w] = v[i] +C[i-1,w-w[i]];

                }

            else {

                    C[i,w] = C[i-1,w];

            }

            }

           els

           e

           {

C[i,w] = C[i-1,w];

          }

      }

   }}

Analysis

For run time analysis examining the above algorithm the overall run time of the algorithm is $O(nW)$.

Example

Let the problem instance be with 7 items where v[] = {2,3,3,4,4,5,7}and w[] = {3,5,7,4,3,9,2}and W = 9.

w i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2	2
2	0	0	0	2	2	3	3	3	5	5
3	0	0	0	2	2	3	3	3	5	5
4	0	0	0	2	4	4	4	6	6	7
5	0	0	0	4	4	4	6	8	8	8
6	0	0	0	4	4	4	6	8	8	8
7	0	0	7	7	7	11	11	11	13	15

Profit= C[7][9]=15

Floyd Warshwall Algorithm

The algorithm being discussed uses dynamic programming approach. The algorithm being presented here works even if some of the edges have negative weights. Consider a weighted

graph G = (V,E) and denote the weight of edge connecting vertices i and j by w_{ij}. Let W be the adjacency matrix for the given graph G. Let D^k denote an n×n matrix such that $D^k(i,j)$ is defined as the weight of the shortest path from the vertex i to vertex j using only vertices

using only vertices from 1,2,...,k as intermediate vertices in the path. If we consider shortest path with intermediate vertices as above then computing the path contains two cases. $D^k(i,j)$ does not contain k as intermediate vertex and $.D^k(i,j)$ contains k as intermediate vertex. Then we have the following relations

$D^k(i,j) = D^{k-1}(i,j)$, when k is not an intermediate vertex, and

$$D^{k-1}(i,k) \qquad D^{k-1}(k,j)$$

$D^k(i,j) = D^{k-1}(i,k) + D^{k-1}(k,j)$, when k is an intermediate vertex.

So from the above relations we obtain:

$D^k(i,j) = \min\{D^{k-1}(i,j), \; D^{k-1}(i,k) + D^{k-1}(k,j)\}$.

The above relation is used by flyod's algorithm to compute all pairs shortest path in bottom up manner for finding $D^1, D^2, ..., D^n$.

Algorithm:

FloydWarshalAPSP(W,D,n)   // W is adjacency matrix of graph G.

{

for(i=1;i<=n;i++)

   for(j=1;j<=1;j++)

        D[i][j] = W[i][j];   // initially D[][] is D^0.

For(k=1;k<=n;k++)

   for(i=1;i<=n;i++)

     for(j=1;j<=1;j++)

        D[i][j] = min{D[i][j], D[i][k]+ D[k][j]};   //D[][]'s are D^k's.

}

Example:

Solution:

Adjacency Matrix

W	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

D^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

Remember we are not showing $D^k(i,i)$, since there will be no change i.e. shortest path is zero.

$D^1(1,2) =$ min$\{D^0(1,2), D^0(1,1)+ D^0(1,2)\}$
= min$\{4, 0+4\} = 4$

$D^1(1,3) =$ min$\{D^0(1,3), D^0(1,1)+ D^0(1,3)\}$
= min$\{11, 0+11\} = 11$

$D^1(2,1) =$ min$\{D^0(2,1), D^0(2,1)+ D^0(1,1)\}$
= min$\{6, 6+0\} = 6$

$D^1(2,3) =$ min$\{D^0(2,3), D^0(2,1)+ D^0(1,3)\}$
= min$\{2, 6+11\} = 2$

$D^1(3,1) =$ min$\{D^0(3,1), D^0(3,1)+ D^0(1,1)\}$
= min$\{3, 3+0\} = 3$

$D^1(3,2) =$ min$\{D^0(3,2), D^0(3,1)+ D^0(1,2)\}$
= min$\{∞, 3+4\} = 7$

$D^2(1,2) =$ min$\{D^1(1,2), D^1(1,2)+ D^1(2,2)\}$
= min$\{4, 4+0\} = 4$

$D^2(1,3) =$ **min$\{D^1(1,3), D^1(1,2)+ D^1(2,3)\}$**

D^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

$$
\begin{aligned}
&= \quad \textbf{min\{11, 4+ 2\} = 6} \\
D^2(2,1) = &\quad \min\{D^1(2,1),\ D^1(2,2)+ D^1(2,1)\} \\
&= \quad \min\{6,\ 0+ 6\} = 6 \\
D^2(2,3) = &\quad \min\{D^1(2,3),\ D^1(2,2)+ D^1(2,3)\} \\
&= \quad \min\{2,\ 0+ 2\} = 2 \\
D^2(3,1) = &\quad \textbf{min\{D}^1\textbf{(3,1), D}^1\textbf{(3,2)+ D}^1\textbf{(2,1)\}} \\
&= \quad \textbf{min\{3, 7+ 6\} = 3} \\
D^2(3,2) = &\quad \min\{D^1(3,2),\ D^1(3,2)+ D^1(2,2)\} \\
&= \quad \min\{7,\ 7+ 0\} = 7
\end{aligned}
$$

D^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

$$
\begin{aligned}
D^3(1,2) = &\quad \textbf{min\{D}^2\textbf{(1,2), D}^2\textbf{(1,3)+ D}^2\textbf{(3,2)\}} \\
&= \quad \textbf{min\{4, 6 + 7\} = 4} \\
D^3(1,3) = &\quad \min\{D^2(1,3),\ D^2(1,3)+ D^2(3,3)\} \\
&= \quad \min\{6,\ 6+ 0\} = 6 \\
D^3(2,1) = &\quad \textbf{min\{D}^2\textbf{(2,1), D}^2\textbf{(2,3)+ D}^2\textbf{(3,1)\}} \\
&= \quad \textbf{min\{6, 2+ 3\} = 5} \\
D^3(2,3) = &\quad \min\{D^2(2,3),\ D^2(2,3)+ D^2(3,3)\} \\
&= \quad \min\{2,\ 2+ 0\} = 2 \\
D^3(3,1) = &\quad \min\{D^2(3,1),\ D^2(3,3)+ D^2(3,1)\} \\
&= \quad \min\{3,\ 0+ 3\} = 3 \\
D^3(3,2) = &\quad \min\{D^2(3,2),\ D^2(3,3)+ D^2(3,2)\} \\
&= \quad \min\{7,\ 0+ 7\} = 7
\end{aligned}
$$

Analysis:

Clearly the above algorithm's running time is $O(n^3)$, where n is cardinality of set V of vertices.

Travelling Salesman Problem

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)? To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term C(S, i) be the cost of the minimum cost path visiting

each vertex in set S exactly once, starting at 1 and ending at i. We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},

 C(S, i) = dist(1, i)

Else if size of S is greater than 2.

 C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

Memorization ensures that a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs (usually in a hash map).

Unit 6
Backtracking

Concept of Backtracking

Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.

- Optimization problem used to find the best solution that can be applied.

- Enumeration problem used to find the set of all feasible solutions of the problem.

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Recursion vs Backtracking

In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Backtracking Algorithms:

Subset-sum Problem,

Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number. The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive approach towards subset sum problem.

Example: n =4, w [] = {3,4,5,6}, W = 13

$$w_1 = 3 \qquad w_2 = 4 \qquad w_3 = 5 \qquad w_4 = 6.$$

void subset_sum(int list[], int sum, int starting_index, int target_sum)

{

```
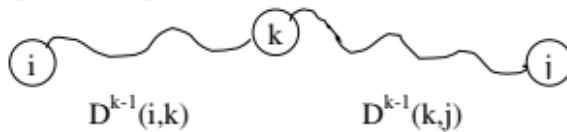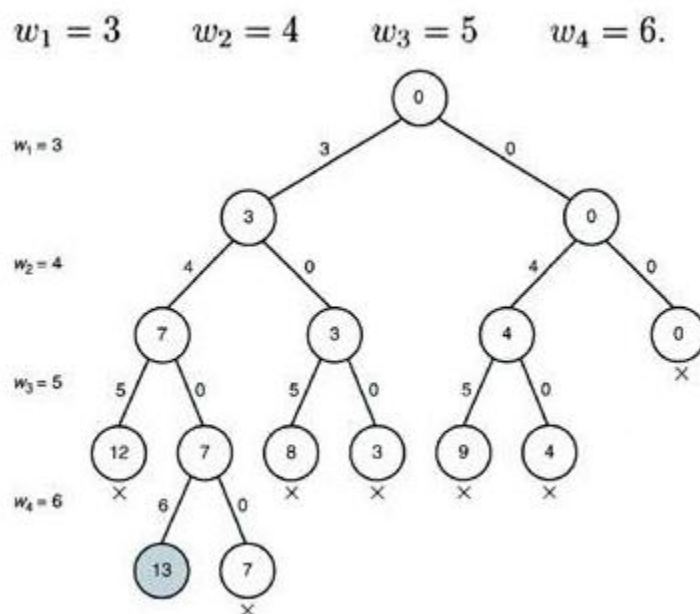    if( target_sum == sum )

    {

      subset_count++;

      if(starting_index < list.length)

        subset_sum(list, sum - list[starting_index-1], starting_index, target_sum);

    }

    else

    {

      for( int i = starting_index; i < list.length; i++ )

      {

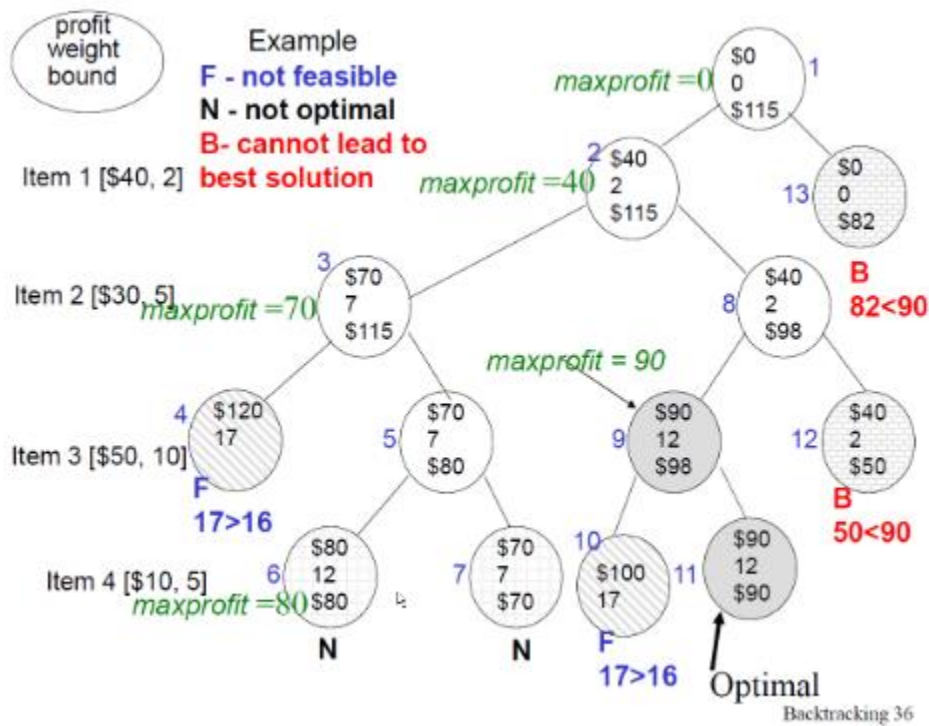        subset_sum(list, sum + list[i], i + 1, target_sum);

      }

    }

  }
```

Time complexity:

backtracking approach which will take O(2^N) time complexity but is significantly faster than the recursive approach which take exponential time as well.

Zero-one Knapsack Problem,

Backtracking is a search algorithm that is both systematic and jumpy. It searches the solution space tree from the root node according to the depth-first strategy in the solution space tree containing all the solutions of the problem. When the algorithm searches for any node of the solution space tree, it first judges whether the node contains the solution of the problem. If it is definitely not included, skip the search of the subtree rooted at the node, and backtrack to its ancestor node layer by layer; otherwise, enter the subtree and continue searching according to the depth-first strategy.

Example:

w[] = {2,5,10,5}

W= 16

P[] = {40 ,30,50,10}

Time complexity analysis:

The upper bound function bound () needs O(n) time. In the worst case, there are $O(2^n)$ right sub-nodes that need to calculate the upper bound. The backtrack algorithm backtrack requires $O(n2^n)$.

N-queen Problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

The expected output is a binary matrix which has 1s for the blocks where queens are placed.
For example, following is the output matrix for above 4 queen solution.

```
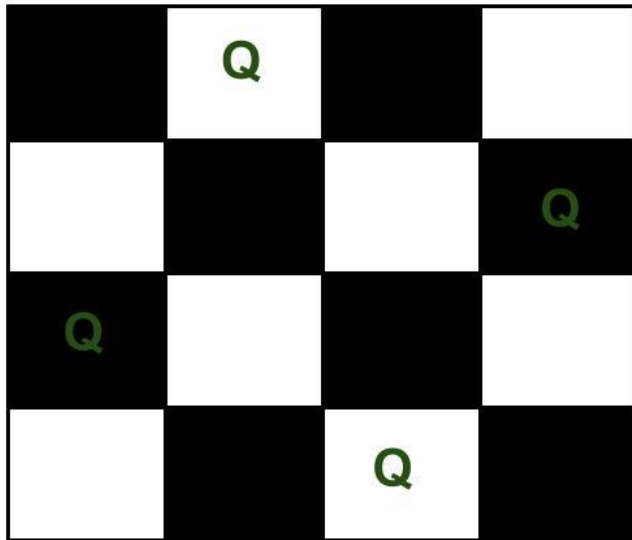{ 0,      1,      0,      0}

{ 0,      0,      0,      1}

{ 1,      0,      0,      0}

{ 0,      0,      1,      0}
```

The idea is to place queens one by one in different columns, starting from the leftmost column.
When we place a queen in a column, we check for clashes with already placed queens. In the
current column, if we find a row for which there is no clash, we mark this row and column as

part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

Algorithm:

is_attacked(x, y, board[][], N) //checking for row and column

   if any cell in xth row is 1

     return true

   if any cell in yth column is 1

     return true

   //checking for diagonals

   if any cell (p, q) having p+q = x+y is 1

     return true

   if any cell (p, q) having p-q = x-y is 1

     return true

   return false

N-Queens(board[][], N)

  if N is 0   //All queens have been placed

    return true

  for i = 1 to N {

    for j = 1 to N {

      if is_attacked(i, j, board, N) is true

        skip it and move to next cell

      board[i][j] = 1   //Place current queen at cell (i,j)

      if N-Queens(board, N-1) is true   // Solve subproblem

        return true   // if solution is found return true

      board[i][j] = 0 /* if solution is not found undo whatever changes

```
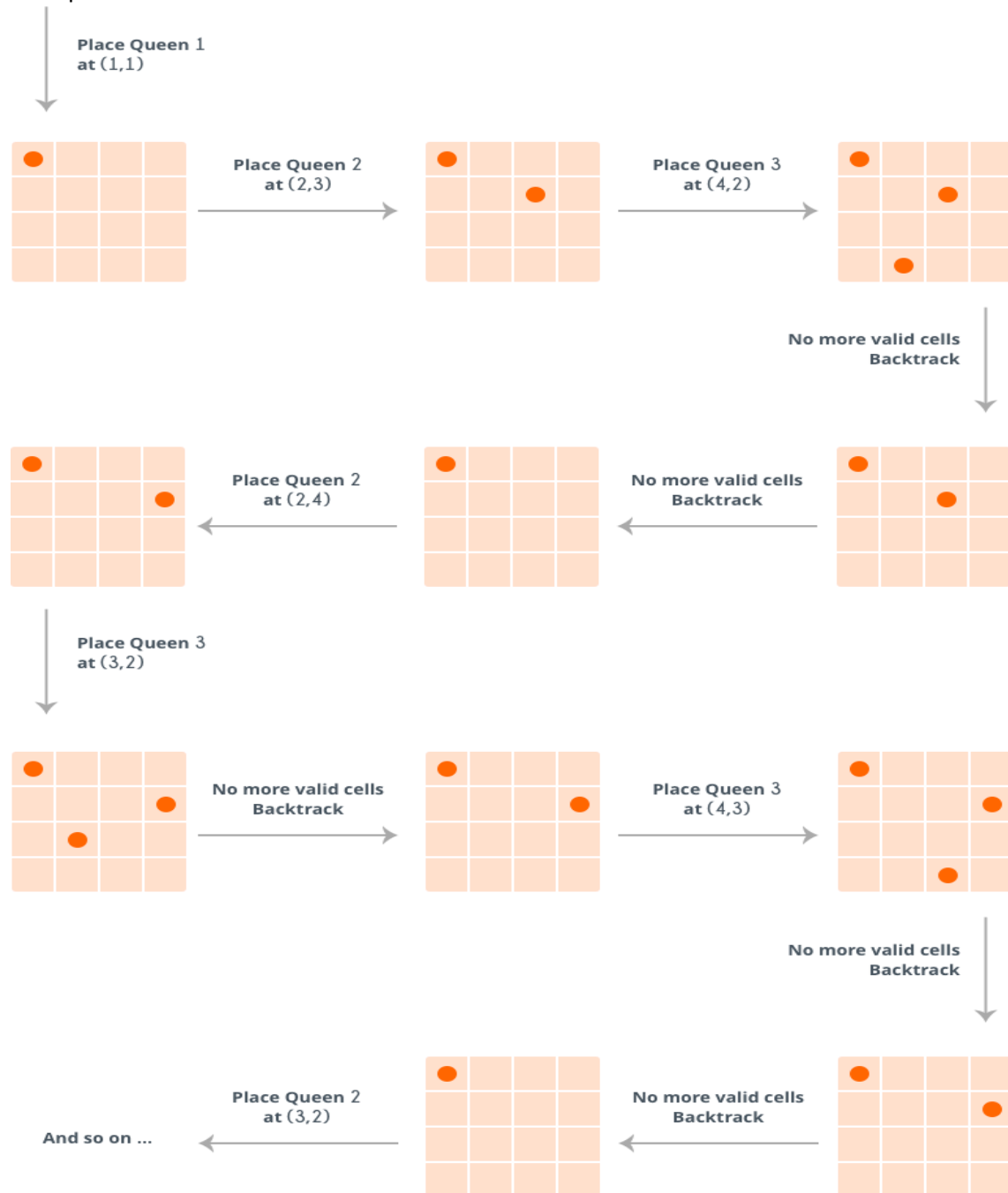                were made i.e., remove  current queen from (i,j)*/
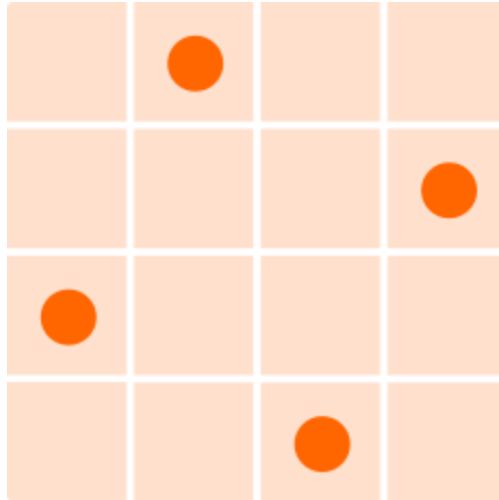    }
  }
  return false
```

Time complexity:

O(n^n) is definitely an upper bound on solving n-queens using backtracking. However, consider this - when we assign a location of the queen in the first column, we have n options, after that, we only have n-1 options as we can't place the queen in the same row as the first queen, then n-2 and so on. Thus, the worst-case complexity is still upper bounded by O(n!).

Example: N =4

Place Queen 1
at (1,1)

Place Queen 2
at (2,3)

Place Queen 3
at (4,2)

No more valid cells
Backtrack

No more valid cells
Backtrack

Place Queen 2
at (2,4)

Place Queen 3
at (3,2)

No more valid cells
Backtrack

Place Queen 3
at (4,3)

No more valid cells
Backtrack

Place Queen 2
at (3,2)

No more valid cells
Backtrack

And so on ...

So, at the end it reaches the following solution:

So, clearly, the above algorithm, tries solving a subproblem, if that does not result in the solution, it undo whatever changes were made and solve the next subproblem. If the solution does not exist (N=2), then it returns false.

Unit 7
Number Theoretic Algorithms

Number Theoretic Notations

Euclidean Algorithm

Euclidean algorithm was formulated as follows: subtract the smaller number from the larger one until one of the numbers is zero. Indeed, if g divides a and b, it also divides a−b. On the other hand, if g divides a−b and b, then it also divides a=b+(a−b), which means that the sets of the common divisors of {a,b} and {b,a−b} coincide.

Note that a remains the larger number until b is subtracted from it at least ⌊a/b⌋ times. Therefore, to speed things up, a−b is substituted with a−⌊a/b⌋ b=a mod b. Then the algorithm is formulated in an extremely simple way:

$$\gcd(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Algorithm:

```
gcd (int a, int b) {
   while (b) {
      a %= b;
      swap (a, b);
   }
   return a;
}
```

Time complexity:

Given that Fibonacci numbers grow exponentially, we get that the Euclidean algorithm works in **O (log min (a, b)).**

Another way to estimate the complexity is to notice that $a \bmod b$ for the case $a \geq b$ is at least 2 times smaller than a, so the larger number is reduced at least in half on each iteration of the algorithm.

Extended Euclid's Algorithms.

While the Euclidean algorithm calculates only the greatest common divisor (GCD) of two integers a and b, the extended version also finds a way to represent GCD in terms of a and b, i.e. coefficients x and y for which:

$a \cdot x + b \cdot y = \gcd(a,b)$

It's important to note, that we can always find such a representation, for instance $\gcd(55,80)=5$ therefore we can represent 5 as a linear combination with the terms 55 and 80:

XGCD (Extended Euclidean Algorithm)

Input: Integers **a** and **b**

Output: Integers **g**, **s**, and **t** such that **g = GCD(a,b)** and $as+bt=gas+bt=g$.

```
def xgcd(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        q, r = divmod(a, b)
        (g, s0, t0) = xgcd(b, r)
        return (g, t0, s0 - t0*q)
```

Chinese Remainder Theorem

If m_1, m_2, ..., m_k are pairwise relatively prime positive integers, and if a_1, a_2, .., ak are any integers, then the simultaneous congruences

$x \equiv a_1 \pmod{m_1}$,

$x \equiv a_2 \pmod{m_2}$,

...,

$x \equiv a_k \pmod{m_k}$ have a solution, and the solution is unique modulo m, where $m = m_1 m_2 \cdots m_k$.

Example: Solve the simultaneous congruences

$x \equiv 6 \pmod{11}$,

$x \equiv 13 \pmod{16}$,

$x \equiv 9 \pmod{21}$,

$x \equiv 19 \pmod{25}$.

Solution: Since 11, 16, 21, and 25 are pairwise relatively prime, the Chinese Remainder Theorem tells us that there is a unique solution modulo m, where $m = 11 \cdot 16 \cdot 21 \cdot 25 = 92400$.

We apply the technique of the Chinese Remainder Theorem with k = 4,

$m_1 = 11$, $m_2 = 16$, $m_3 = 21$, $m_4 = 25$,

$a_1 = 6$, $a_2 = 13$, $a_3 = 9$, $a_4 = 19$,

to obtain the solution

We compute

$$z_1 = m/m_1 = m_2 m_3 m_4 = 16 \cdot 21 \cdot 25 = 8400$$
$$z_2 = m/m_2 = m_1 m_3 m_4 = 11 \cdot 21 \cdot 25 = 5775$$
$$z_3 = m/m_3 = m_1 m_2 m_4 = 11 \cdot 16 \cdot 25 = 4400$$
$$z_4 = m/m_4 = m_1 m_3 m_3 = 11 \cdot 16 \cdot 21 = 3696$$

$$y_1 \equiv z_1^{-1} \pmod{m_1} \equiv 8400^{-1} \pmod{11} \equiv 7^{-1} \pmod{11} \equiv 8 \pmod{11}$$
$$y_2 \equiv z_2^{-1} \pmod{m_2} \equiv 5775^{-1} \pmod{16} \equiv 15^{-1} \pmod{16} \equiv 15 \pmod{16}$$
$$y_3 \equiv z_3^{-1} \pmod{m_3} \equiv 4400^{-1} \pmod{21} \equiv 11^{-1} \pmod{21} \equiv 2 \pmod{21}$$
$$y_4 \equiv z_4^{-1} \pmod{m_4} \equiv 3696^{-1} \pmod{25} \equiv 21^{-1} \pmod{25} \equiv 6 \pmod{25}$$

$$w_1 \equiv y_1 z_1 \pmod{m} \equiv 8 \cdot 8400 \pmod{92400} \equiv 67200 \pmod{92400}$$
$$w_2 \equiv y_2 z_2 \pmod{m} \equiv 15 \cdot 5775 \pmod{92400} \equiv 86625 \pmod{92400}$$
$$w_3 \equiv y_3 z_3 \pmod{m} \equiv 2 \cdot 4400 \pmod{92400} \equiv 8800 \pmod{92400}$$
$$w_4 \equiv y_4 z_4 \pmod{m} \equiv 6 \cdot 3696 \pmod{92400} \equiv 22176 \pmod{92400}$$

The solution, which is unique modulo 92400, is

$$
\begin{aligned}
x &\equiv a_1 w_1 + a_2 w_2 + a_3 w_3 + a_4 w_4 \pmod{92400} \\
&\equiv 6 \cdot 67200 + 13 \cdot 86625 + 9 \cdot 8800 + 19 \cdot 22176 \pmod{92400} \\
&\equiv 2029869 \pmod{92400} \\
&\equiv \mathbf{51669} \pmod{92400}
\end{aligned}
$$

Primality Testing: Miller-Rabin Randomized Primality Test

TEST (n) is:

1. Find integers k, q, $k > 0$, q odd, so that $(n-1)=2^k q$

2. Select a random integer a, $1 < a < n-1$

3. **if** a^q mod $n = 1$ **then** return ("maybe prime");

4. **for** $j = 0$ **to** $k - 1$ **do**

5. **if** ($a^{2^j q}$ mod $n = n\text{-}1$)

                then return(" maybe prime ")

6. return ("composite")

Probabilistic Considerations

- if Miller-Rabin returns "composite" the number is definitely not prime

- otherwise is a prime or a pseudo-prime

- chance it detects a pseudo-prime is $< {}^1/_4$

- hence if repeat test with different random a then chance n is prime after t tests is:

  - Pr(n prime after t tests) = $1-4^{-t}$

  - eg. for t=10 this probability is > 0.99999

Unit 8

NP-Completeness

Most of the problems considered up to now can be solved by algorithms in worst-case polynomial time. There are many problems and it is not necessary that all the problems have the apparent solution. This concept, somehow, can be applied in solving the problem using the computers. The computer can solve: some problems in limited time e.g. sorting, some problems requires unmanageable amount of time e.g. Hamiltonian cycles, and some problems cannot be solved e.g. Halting Problem. In this section we concentrate on the specific class of problems called NP complete problems (will be defined later).

Tractable and Intractable Problems

We call problems as tractable or easy, if the problem can be solved using polynomial time algorithms. The problems that cannot be solved in polynomial time but requires super polynomial time algorithm are called intractable or hard problems. There are many problems for which no algorithm with running time better than exponential time is known some of them are, traveling salesman problem, Hamiltonian cycles, and circuit satisfiability, etc.

Polynomial time reduction

Given two problems A and B, a polynomial time reduction from A to B is a polynomial time function f that transforms the instances of A into instances of B such that the output of algorithm for the problem A on input instance x must be same as the output of the algorithm for the problem B on input instance f(x) as shown in the figure below. If there is polynomial time computable function f such that it is possible to reduce A to B, then it is denoted as A $\leq_p$ B. The function f described above is called reduction function and the algorithm for computing f is called reduction algorithm.

P and NP classes and NP completeness

The set of problems that can be solved using polynomial time algorithm is regarded as class P. The problems that are verifiable in polynomial time constitute the class NP. The class of NP complete problems consists of those problems that are NP as well as they are as hard as any problem in NP (more on this later). The main concern of studying NP completeness is to understand how hard the problem is. So if we can find some problem as NP complete then we try to solve the problem using methods like approximation, rather than searching for the faster algorithm for solving the problem exactly.

Complexity Class P

P is the class of problems that can be solved in polynomial time on a deterministic effective computing system (ECS). Loosely speaking, all computing machines that exist in the real world are deterministic ECSs. So P is the class of things that can be computed in polynomial time on real computers.

Complexity Class NP

NP is the class of problems that can be solved in polynomial time on a non-deterministic effective computing system (ECS) or we can say that "NP is the class of problems that can be solved in super polynomial time on a deterministic effective computing system (ECS)". Loosely speaking, all computing machines that exist in the real world are deterministic ECSs. So NP is the class of problem that can be computed in super polynomial time on real computers. But problem of class NP are verifiable in polynomial time. Using the above idea we say the problem is in class NP (nondeterministic polynomial time) if there is an algorithm for the problem that verifies the problem in polynomial time. For e.g. Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable? i.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time.

So we can say that circuit satisfiability problem is NP.

NP-Completeness:

NP complete problems are those problems that are hardest problems in class NP. We define some problem say A, is NP-complete if
1. A ∈ NP, and
2. B ≤$_p$ A, for every B ∈ NP.
We call the problem (or language) A satisfying property 2 is called NP-hard.

The problem satisfying property b is called NP-hard.

NP-Complete problems arise in many domains like: Boolean logic; graphs, sets and partitions; sequencing, scheduling, allocation; automata and language theory; network design; compilers, program optimization; hardware design/optimization; number theory, algebra etc.

Cook's Theorem

SAT is NP-complete

Proof

To prove that SAT is NP-complete, we have to show that

- SATεNP

- SAT is NP-Hard

SAT ε NP

Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable? i.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

SAT is NP-hard

Take a problem V ε NP, let A be the algorithm that verifies V in polynomial time (this must be true since V ε NP). We can program A on a computer and therefore there exists a (huge) logical circuit whose input wires correspond to bits of the inputs x and y of A and which outputs 1 precisely when A(x,y) returns yes. For any instance x of V let Ax be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of Ax from x is our reduction function.

Approximation Algorithms

An approximate algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time. If we are dealing with optimization problem (maximization or minimization) with feasible solution having positive cost then it is worthy to look at approximate algorithm for near optimal solution.

Vertex Cover Problem

Vertex Cover Problem:

A vertex cover of an undirected graph G =(V,E) is a subset V' ⊆ V such that for all edges (u,v) ∈ E either u∈ V' or v∈ V' or u and v ∈ V'. The problem here is to find the vertex cover of minimum size in a given graph G. Optimal vertex-cover is the optimization version of an NP-complete problem but it is not too hard to find a vertex-cover that is near optimal.

Algorithm:
```
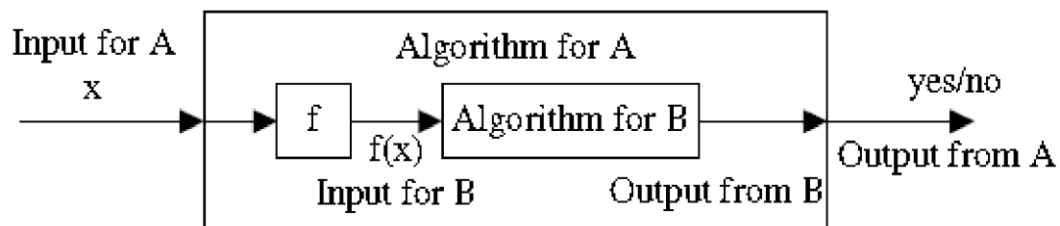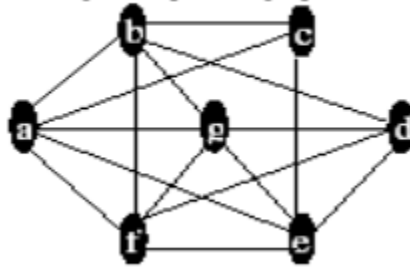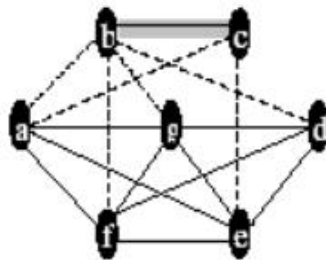ApproxVertexCover (G)
{
        C  {};
        E' = E
        while E` is not empty
        do Let (u, v) be an arbitrary edge of E`
        C = C ≈ {u, v}
        Remove from E` every edge incident on either u or v
        return C
}
```

Example: (vertex cover running example for graph below)

Solution:

Edge chosen is (b,c) C = {b,c}

Edge chosen is (f,e) C= {b,c,e,f}

Edge chosen is (g,d) C = {b,c,d,e,f,g}

Optimal vertex cover as lightly shaded vertices

Analysis:
If E' is represented using the adjacency lists the above algorithm takes O (V+E) since each edge is processed only once and every vertex is processed only once throughout the whole operation.