UNIT:- 1

Principles of Analyzing algorithms and Problems

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

- You are given two numbers, how do you find the Greatest Common Divisor.
- Given an array of numbers, how do you 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).

You need designing concepts of the algorithms because if you only study the algorithms then you are bound to those algorithms and selection among the available algorithms. However if you have knowledge about design then you 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.

Random Access Machine 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.

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 insures 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).

In our study we concentrate on worst case complexity only.

Example 1: Fibonacci Numbers

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 ;
}
```

Efficiency

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).

Mathematical Foundation

Since mathematics can provide clear view of an algorithm. Understanding the concepts of mathematics is aid in the design and analysis of good algorithms. Here we present some of the mathematical concepts that are helpful in our study.

Exponents

Some of the formulas that are helpful are:

$x^a x^b = x^{a+b}$
$x^a / x^b = x^{a-b}$
$(x^a)^b = x^{ab}$
$x^n + x^n = 2x^n$
$2^n + 2^n = 2^{n+1}$

Logarithmes

Some of the formulas that are helpful are:

1. $\log_a b = \log_c b / \log_c a$; c>0
2. log ab = log a + log b
3. log a/b = log a - log b
4. log (a^b) = b log a
5. Log x < x for all x>0
6. Log 1 = 0, log 2 = 1, log 1024 = 10.
7. $_a \log b^n = {}_n \log b^a$

Series

1. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
2. $\sum_{i=0}^{n} a^i \leq 1 / 1\text{-}a$; if 0<a<1

   $= a^{n+1} - 1 / a\text{-}1$; else
3. $\sum_{i=1}^{n} i = n(n+1) / 2$
4. $\sum_{i=0}^{n} i^2 = n(n+1)(2n+1) / 6$
5. $\sum_{i=0}^{n} i^k \approx n^{k+1} / |k+1|$; k != -1
6. $\sum_{i=1}^{n} 1/i \approx \log_e n$

Asymptotic Notation

Complexity analysis of an algorithm is very hard if we try to analyze exact. we know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input. So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier. For this purpose we need the concept of asymptotic notations. The figure below gives upper and lower bound concept.

Big Oh (O) notation

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)

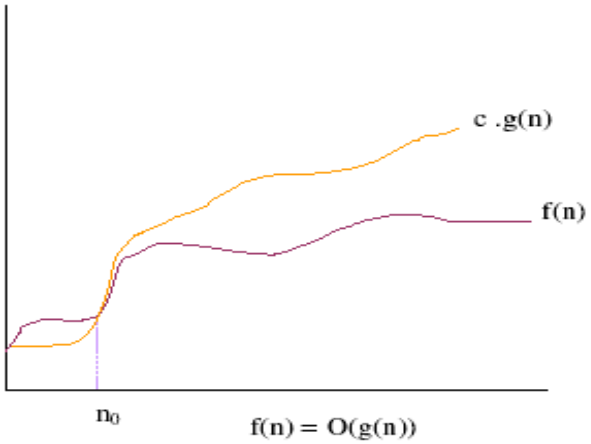
The above relation says that g(x) is an upper bound of f(x)

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.



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))
2. Prove that n log (n^3) is O($\sqrt{n^3}$).
   Proof: we have n log (n^3) = 3n log n
   Again, $\sqrt{n^3} = n\sqrt{n}$,
   If we can prove log n = O($\sqrt{n}$) then problem is solved
   Because n log n = n O($\sqrt{n}$) that gives the question again.
   We can remember the fact that $\log^a n$ is O (n^b) for all a,b>0.
   In our problem a = 1 and b = 1/2 ,
   hence log n = O($\sqrt{n}$).
   So by knowing log n = O($\sqrt{n}$) we proved that
   n log (n^3) = O($\sqrt{n^3}$)).

Big Omega (Ω) notation

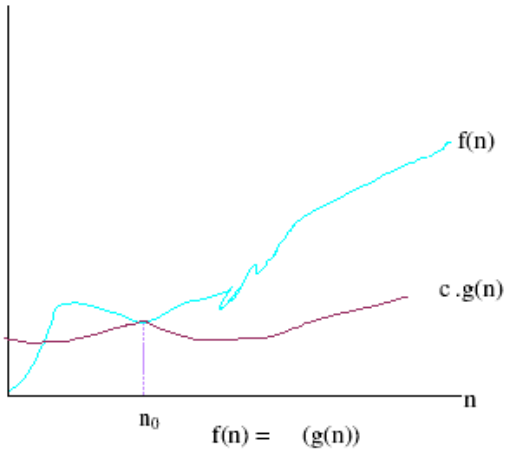
Big omega notation gives asymptotic lower bound. A function f(x) =Ω (g(x)) (read as g(x) is big omega of g(x)) iff there exists two positive constants c and x_0 such that for all x >= x_0, 0 <= c*g(x) <= f(x).

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

Some properties:

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

Reflexivity: f(x) = O(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 Theta (Θ) notation

When we need asymptotically tight bound then we use notation. A function f(x) = (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, $c_1*g(x) <= f(x) <= c_2*g(x)$

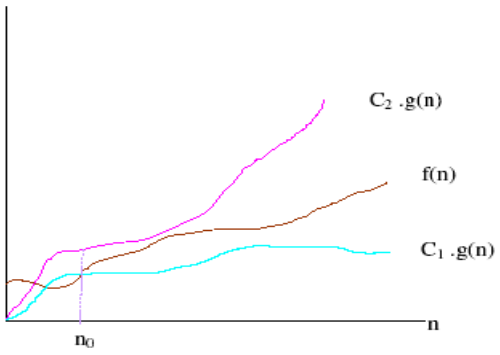
The above relation says that f(x) is order of g(x)

Some properties:

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

Reflexivity: $f(x) = \Theta (f(x))$

Symmetry: $f(x) = \Theta$ 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).

Example

1. $f(n) = 3n^2 + 4n + 7$
   $g(n) = n^2$, then prove that $f(n) = (g(n))$.
   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 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
$T(n)=T(n/2) + O(1)$ when n>1

Techniques for Solving Recurrences

We'll use four techniques:

- Iteration method
- Recursion Tree
- Substitution
- Master Method – for divide & conquer
- Characteristic Equation – for linear

Iteration method

- Expand the relation so that summation independent on n is obtained.
- Bound the summation

e.g.

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

$T(n) = 2T(n/2) +1$
$= 2 \{ 2T(n/4) + 1\} +1$
$= 4T(n/4) + 2 + 1$

$= 4 \{ T(n/8) +1 \} + 2 + 1$

$= 8 T(n/8) + 4 + 2 + 1$

……

……

$= 2^k T(n/2^k) + 2^{k-1} T(n/2^{k-1}) + \ldots\ldots + 4 + 2 + 1.$

For simplicity assume:

$n= 2^k$

- ⇨ $k=\log n$
- ⇨ $T(n)= 2^k + 2^{k-1} + \ldots\ldots + 2^2 + 2^1 + 2^0$
- ⇨ $T(n)= (2^{k+1} -1)/ (2-1)$
- ⇨ $T(n)= 2^{k+1} -1$
- ⇨ $T(n)= 2.2^k -1$
- ⇨ $T(n)= 2n-1$
- ⇨ $T(n)= O(n)$

Second Example:

$T(n) = T(n/3) + O(n)$ when $n>1$

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

$T(n) = T(n/3) + O(n)$

- ⇨ $T(n) = T(n/3^2) + O(n/3) + O(n)$
- ⇨ $T(n) = T(n/3^3) + O(n/3^2) + O(n/3) + O(n)$
- ⇨ $T(n) = T(n/3^4) + O(n/3^3) + O(n/3^2) + O(n/3) + O(n)$
- ⇨ $T(n)= T(n/3^k) + O(n/3^{k-1}) + \ldots\ldots + O(n/3) + O(n)$

For Simplicity assume

$n= 3^k$

- ⇨ $k = \log_3 n$
- ⇨ $T(n) <= T(1) + c\, n/3^{k-1} \ldots\ldots + c.n/3^2 + c.n/3 + c.n$
- ⇨ $T(n) <= 1 + \{ c.n/3^{k-1} \ldots\ldots + c.n/3^2 + c.n/3 + c.n\}$
- ⇨ $T(n) <= 1 + c.n \{ 1/(1-1/3) \}$
- ⇨ $T(n) <= 1 + 3/2\ c.n$
- ⇨ $T(n) = O(n)$

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:

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

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

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

More specifically:

$T(n) \le cn^3$, for all large enough n.

Prove by strong induction on n.

Assume: $T(k) \le ck^3$ for $\forall k<n$.

Show: $T(n) \le cn^3$ for $\forall n>n0$.

Base case,

For n=1:

$T(n) = 1$ Definition

$1 \le c$ Choose large enough c for conclusion

Inductive case, n>1:

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

$\le 4c\cdot(n/2)^3 + n$ Induction.

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

While this is $O(n^3)$, we're not done.
Need to show $c/2 \cdot n^3 + n \le c\cdot n^3$.
Fortunately, the constant factor is shrinking, not growing.

$T(n) \le c/2 \cdot n^3 + n$ From before.

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

$\le cn^3$ Since n>0, if c≥2

Proved:

$T(n) \le 2n^3$ for $\forall n>0$

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

Second Example

$T(n) = 1$ n=1

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

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

Same recurrence, but now try tighter bound.

More specifically:

$T(n) \le cn^2$ for $\forall n>n0$.

Assume $T(k) \le ck^2$, for $\forall k<n$.

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

$\le 4c\cdot(n/2)$

$= cn^2 + n$

Not $\le cn^2$!
Problem is that the constant isn't shrinking.

Solution: Use a tighter guess & inductive hypothesis.
Subtract a lower-order term – a common technique.
Guess:
$T(n) \le cn^2 - dn$ for $\forall n>0$

Assume $T(k) \le ck^2 - dk$, for $\forall k<n$. Show $T(n) \le cn^2 - dn$.

Base case, n=1

$T(n) = 1$ Definition.

$1 \le c-d$ Choosing c, d appropriately.

Inductive case, n>1:

$T(n) = 4T(n/2) + n$ Definition.
$\leq 4(c(n/2)^2 - d(n/2)) + n$ Induction.
$= cn^2 - 2dn + n$ Algebra.
$= cn^2 - dn - (dn - n)$ Algebra.
$\leq cn^2 - dn$ Choosing $d \geq 1$.
$T(n) \leq 2n^2 - dn$ for $\forall n > 0$
Thus, $T(n) = O(n^2)$.

Ability to guess effectively comes with experience.

Changing Variables:

Sometimes a little algebric manipulation can make a unknown recurrence similar to one we have seen

Consider the example

$T(n) = 2T(\lfloor n^{1/2} \rfloor) + \log n$

Looks Difficult: Rearrange like

Let $m=\log n \Rightarrow n= 2^m$

Thus,

$T(2^m)= 2T(2^{m/2})+m$

Again let

$S(m)= T(2^m)$ $S(m)= 2S(m/2) + m$

We can show that

$S(m)=O(\log m)$

⇨ $T(n)=T(2^m) =S(m)= O(m\log m) = O(\log n \log\log n)$

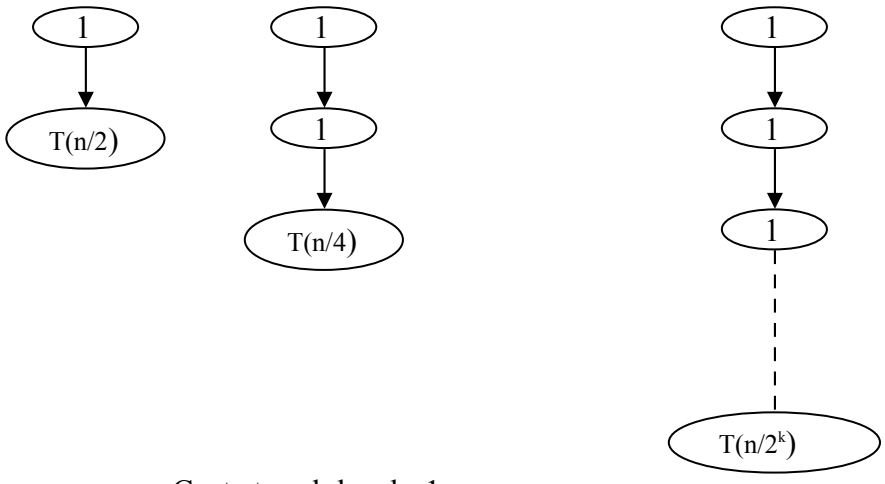
Recursion Tree

Just Simplification of Iteration method:

Consider the recurrence

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

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



Cost at each level =1

For simplicity assume that $n= 2^K$

⇨ $k= \log n$

Summing the cost at each level,

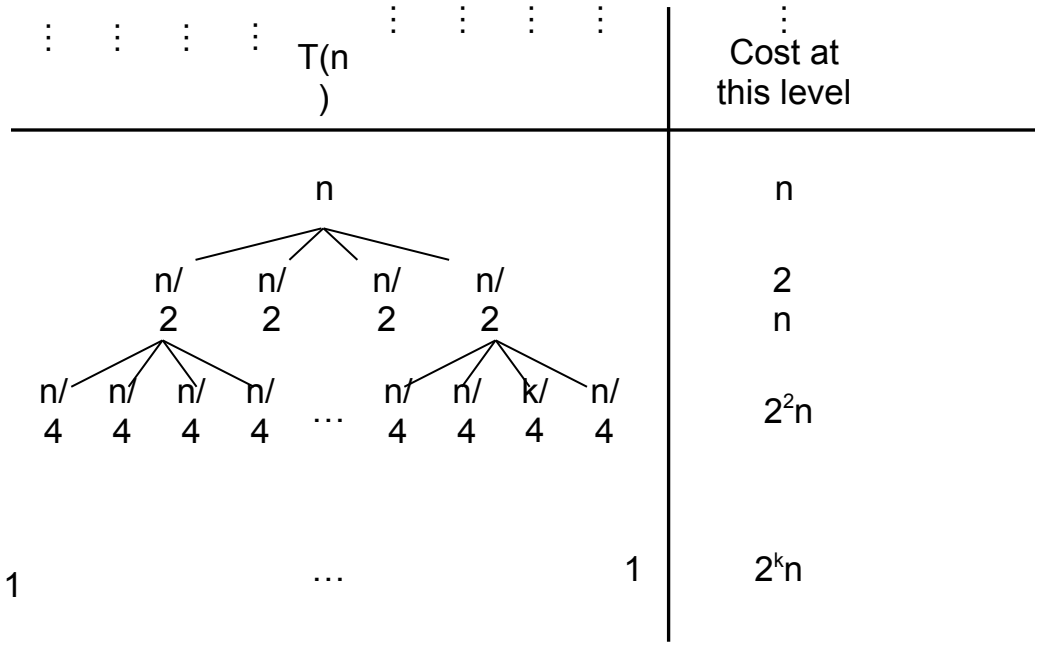
Total cost = 1 + 1 + 1 + ……. Up to logn terms

⇨ complexity = O (logn)

Second Example

T(n) = 1
n=1

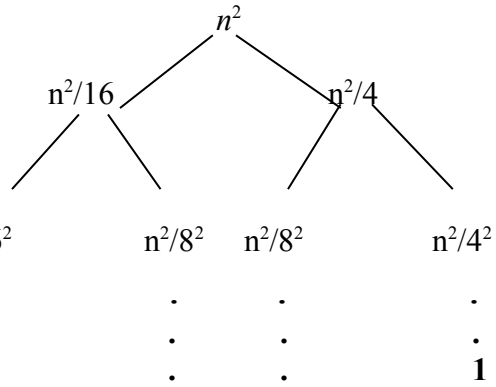
T(n) = 4T(n/2) + n
n>1



Asume: $n= 2^k$
$k= \log n$
$T(n) = n + 2n + 4n + \ldots + 2^{k-1}n + 2^k n$
$= n(1 + 2 + 4 + \ldots + 2^{k-1} + 2^k$
$= n(2^{k+1} -1)/(2-1)$
$= n (2^{k+1} -1)$
$\leq n\ 2^{k+1}$
$=2n\ 2^k$
$= 2n . n$
$= O(n^2)$

Third 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 ≤? Why not =?}

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

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

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

$= O(n^2)$

Master Method

Cookbook solution for some 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

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

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

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

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 $\varepsilon \leq 0.8$.

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

Master Method Case 2

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

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

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

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

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

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

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

$a \cdot f(n/b) \leq c \cdot f(n)$ for some $c < 1$ and all large enough n

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

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

Again, $4(n/2)^3 = \frac{1}{2} \cdot n^3 \leq cn^3$, for any $c \geq \frac{1}{2}$.

$T(n) = \Theta(n^3)$

Master Method Case 4

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

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

No.

Case 3? $n^2/\lg n = {}^{?} \Omega(n^{\log b\, a + \varepsilon}) = \Omega(n^{\log 2\, 4 + \varepsilon}) = \Omega(n^{2 + \varepsilon})$

No, since 1/lg n is asymptotically less than n^{ε}.

Exercises

- Show that the solution of $T(n) = 2T(n/2) + n$ is $\Omega(n\log n)$. Conclude that solution is $\Theta(n\log n)$.
- Show that the solution to $T(n) = 2T(n/2 + 17) + n$ is $O(n\log n)$.
- Write recursive Fibonacci number algorithm derive recurrence relation for it and solve by substitution method. {Guess 2^n}
- Argue that the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + n$ is $(n\log n)$ by appealing to a recursion tree.
- Use iteration to solve the recurrence $T(n) = T(n-a) + T(a) + n$, where $a >= 1$ is a constant.
- The running time of an algorithm A is described by the recurrence $T(n) = 7T(n/2) + n^2$. A competing algorithm A' has a running time of $T'(n) = aT'(n/4) + n2$. What is the largest integer value for 'a' such that A' is asymptotically faster than A?

Review of Data Structures

This part is to introduce some of the data structures if you want rigorous study you can consult the book on Data Structures.

Simple Data structures

The basic structure to represent unit value types are bits, integers, floating numbers, etc. The collection of values of basic types can be represented by arrays, structure, etc. The access of the values are done in constant time for these kind of data structured

Linear Data Structures

Linear data structures are widely used data structures we quickly go through the following linear data structures.

Lists

List is the simplest general-purpose data structure. They are of different variety. Most fundamental representation of a list is through an array representation. The other representation includes linked list. There are also varieties of representations for lists as linked list like singly linked, doubly linked, circular, etc. There is a mechanism to point to the first element. For this some pointer is used. To traverse there is a mechanism of pointing the next (also previous in doubly linked). Lists require linear space to collect and store the elements where linearity is proportional to the number of items. For e.g. to store n items in an array nd space is required were d is size of data. Singly linked list takes $n(d + p)$, where p is size of pointer. Similarly for doubly linked list space requirement is $n(d + 2p)$.

Array representation

- ✓ Operations require simple implementations.
- ✓ Insert, delete, and search, require linear time, search can take O(logn) if binary search is used. To use the binary search array must be sorted.
- ✓ Inefficient use of space

Singly linked representation (unordered)

1. Insert and delete can be done in O(1) time if the pointer to the node is given, otherwise O(n) time.
2. Search and traversing can be done in O(n) time
3. Memory overhead, but allocated only to entries that are present.

Doubly linked representation

4. Insert and delete can be done in O(1) time if the pointer to the node is given, otherwise O(n) time.
5. Search and traversing can be done in O(n) time

6. Memory overhead, but allocated only to entries that are present, search becomes easy.

boolean isEmpty ();

Return true if and only if this list is empty.

- int size ();

Return this list's length.

- boolean get (int i);

Return the element with index i in this list.

- boolean equals (List a, List b);

Return true if and only if two list have the same length, and each element of the lists are equal

- void clear ();
  Make this list empty.
- void set (int i, int elem);
  Replace by elem the element at index i in this list.
- void add (int i, int elem);
  Add elem as the element with index i in this list.
- void add (int elem);
  Add elem after the last element of this list.
- void addAll (List a List b);
  Add all the elements of list b after the last element of list a.
- int remove (int i);

Remove and return the element with index i in this list.

- void visit (List a);

Prints all elements of the list

Operation	Array representation	SLL representation
get	$O(1)$	$O(n)$
set	$O(1)$	$O(n)$
add(int,data)	$O(n)$	$O(n)$
add(data)	$O(1)$	$O(1)$
remove	$O(n)$	$O(n)$
equals	$O(n^2)$	$O(n^2)$
addAll	$O(n^2)$	$O(n^2)$

Stacks and Queues

These types of data structures are special cases of lists. Stack also called LIFO (Last In First Out) list. In this structure items can be added or removed from only one end. Stacks are generally represented either in array or in singly linked list and in both cases insertion/deletion time is O(1), but search time is O(n).

Operations on stacks

- **boolean** isEmpty ();
  Return true if and only if this stack is empty. Complexity is O(1).
- int getLast ();
  Return the element at the top of this stack. Complexity is O(1).

- **void** clear ();
  Make this stack empty. Complexity is O(1).
- **void** push (int elem);
  Add elem as the top element of this stack. Complexity is O(1).
- int pop ();
  Remove and return the element at the top of this stack. Complexity is O(1).

The queues are also like stacks but they implement FIFO(First In First Out) policy. One end is for insertion and other is for deletion. They are represented mostly circularly in array for O(1) insertion/deletion time. Circular singly linked representation takes O(1) insertion time and O(1) deletion time. Again Representing queues in doubly linked list have O(1) insertion and deletion time.

Operations on queues

3. boolean isEmpty ();
   Return true if and only if this queue is empty. Complexity is O(1).
4. int size ();
   Return this queue's length. Complexity is O(n).
5. int getFirst ();
   Return the element at the front of this queue. Complexity is O(1).
6. void clear ();
   Make this queue empty. Complexity is O(1).
7. void insert (int elem);
   Add elem as the rear element of this queue. Complexity is O(1).
8. int delete ();
   Remove and return the front element of this queue. Complexity is O(1).

Tree Data Structures

Tree is a collection of nodes. If the collection is empty the tree is empty otherwise it contains a distinct node called root (r) and zero or more sub-trees whose roots are directly connected to the node r by edges. The root of each tree is called child of r, and r the parent. Any node without a child is called leaf. We can also call the tree as a connected graph without a cycle. So there is a path from one node to any other nodes in the tree. The main concern with this data structure is due to the running time of most of the operation require O(logn). We can represent tree as an array or linked list.

Some of the definitions

- Level *h* of a full tree has d^{h-1} nodes.
- The first *h* levels of a full tree have

$1 + d + d^2 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots d^{h-1} = (d^h - 1)/(d-1)$

Binary Search Trees

BST has at most two children for each parent. In BST a key at each vertex must be greater than all the keys held by its left descendents and smaller or equal than all the keys held by its right descendents. Searching and insertion both takes O(h) worst case time, where h is height of tree and the relation between height and number of nodes n is given by $\log n < h+1 <= n$. for e.g. height of binary tree with 16 nodes may be anywhere between 4 and 15.

When height is 4 and when height is 15?

So if we are sure that the tree is height balanced then we can say that search and insertion has O(log n) run time otherwise we have to content with O(n).

Operation	Algorithm	Time complexity
search	BST search	$O(\log n)$ best $O(n)$ worst
add	BST insertion	$O(\log n)$ best $O(n)$ worst
Remove	BST deletion	$O(\log n)$ best $O(n)$ worst

AVL Trees

Balanced tree named after Adelson, Velskii and Landis. AVL trees consist of a special case in which the sub-trees of each node differ by at most 1 in their height. Due to insertion and deletion tree may become unbalanced, so rebalancing must be done by using left rotation, right rotation or double rotation.

Operation	Algorithm	Time complexity
Search	AVL search	$O(\log n)$ best, worst
Add	AVL insertion	$O(\log n)$ best, worst
Remove	AVL deletion	$O(\log n)$ best, worst

Priority Queues

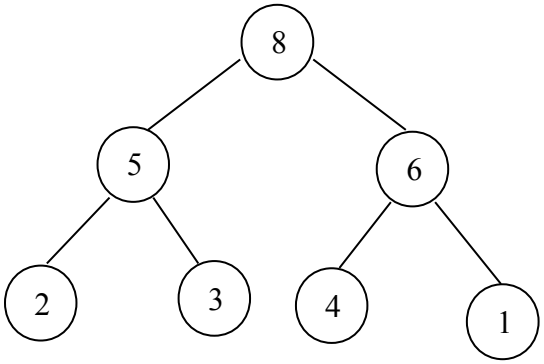
Priority queue is a queue in which the elements are prioritized. The least element in the priority queue is always removed first. Priority queues are used in many computing applications. For example, many operating systems used a scheduling algorithm where the next process executed is the one with the shortest execution time or the highest priority. Priority queues can be implemented by using arrays, linked list or special kind of tree (I.e. heap).

- boolean isEmpty ();
  Return true if and only if this priority queue is empty.
- int size ();
  Return the length of this priority queue.
- int getLeast ();
  Return the least element of this priority queue. If there are several least elements, return any of them.
- void clear ();
  Make this priority queue empty.
- void add (int elem);
  Add elem to this priority queue.
- int delete(); Remove and return the least element from this priority queue. (If there are several least elements, remove the same element that would be returned by getLeast.
-

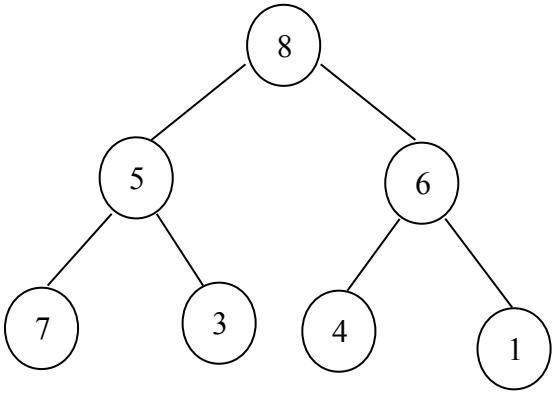
Operation	Sorted SLL	Unsorted SLL	Sorted Array	Unsorted Array
add	$O(n)$	$O(1)$	$O(n)$	$O(1)$
removeLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$
getLeast	$O(1)$	$O(n)$	$O(1)$	$O(n)$

Heap

A heap is a complete tree with an ordering-relation R holding between each node and its descendant. Note that the complete tree here means tree can miss only rightmost part of the bottom level. R can be smaller-than, bigger-than.
E.g. Heap with degree 2 and R is "bigger than".



Heap



Not a heap

Heap Sort Build a heap from the given set (O(n)) time, then repeatedly remove the elements from the heap (O(n log n)).

Implementation

Heaps are implemented by using arrays. Insertion and deletion of an element takes O(log n) time. More on this later

Operation	Algorithm	Time complexity
add	insertion	$O(\log n)$
delete	deletion	$O(\log n)$
getLeast	access root element	$O(1)$

UNIT:-2
Divide and Conquer Algorithms
(Sorting Searching and Selection)

Chapter:1

Sorting

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.

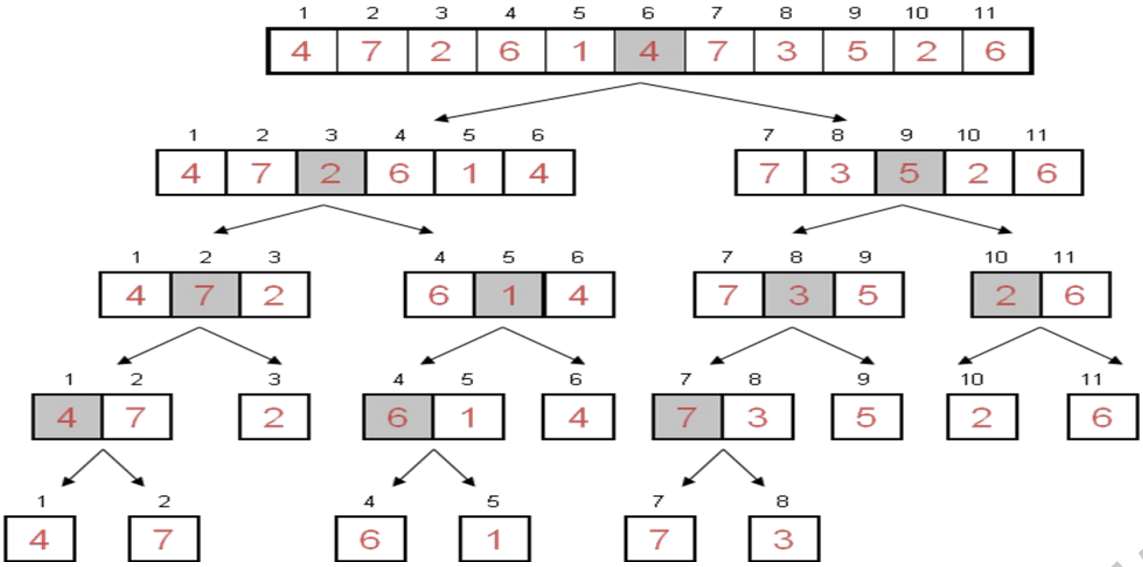
Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key,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

Merge Sort

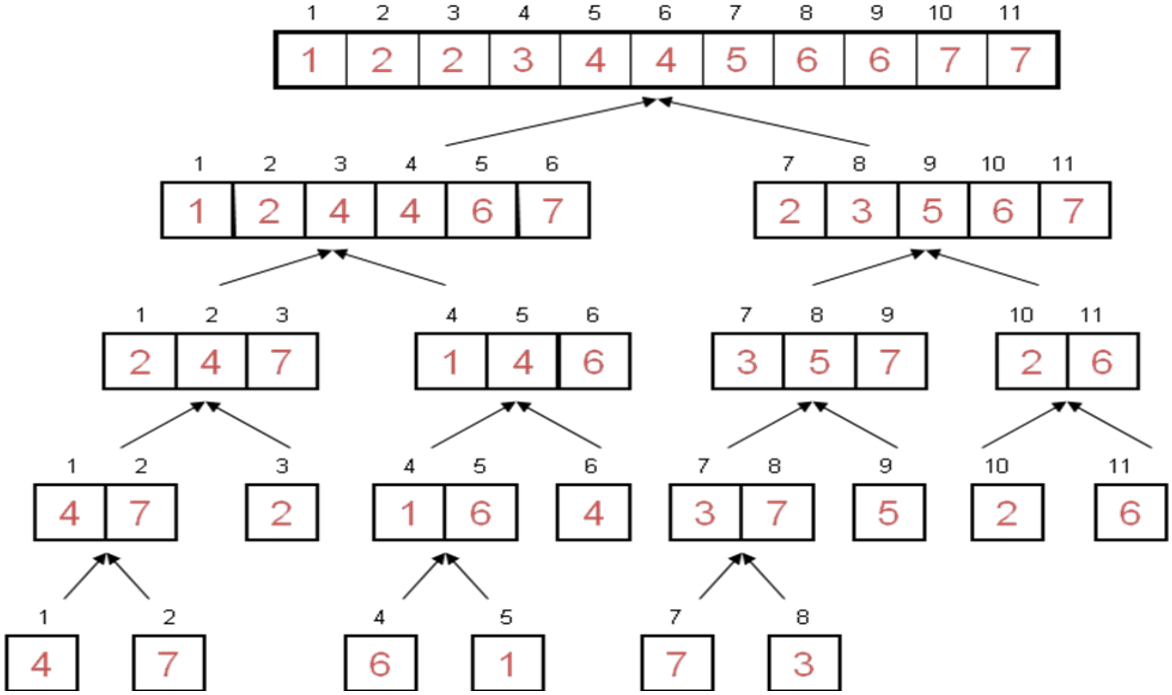
To sort an array A[l . . r]:

- 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:



Algorithm:

```
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)

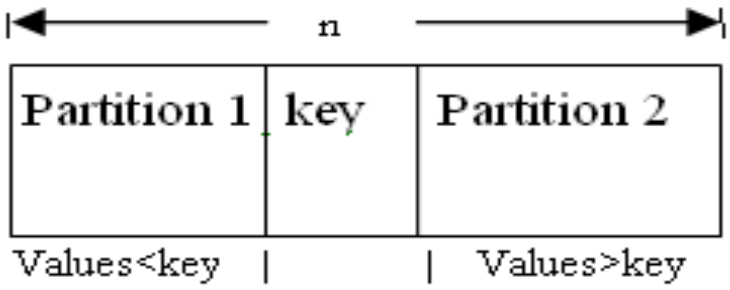
Space Complexity:

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

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

Quick Sort

- **Divide**
  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:

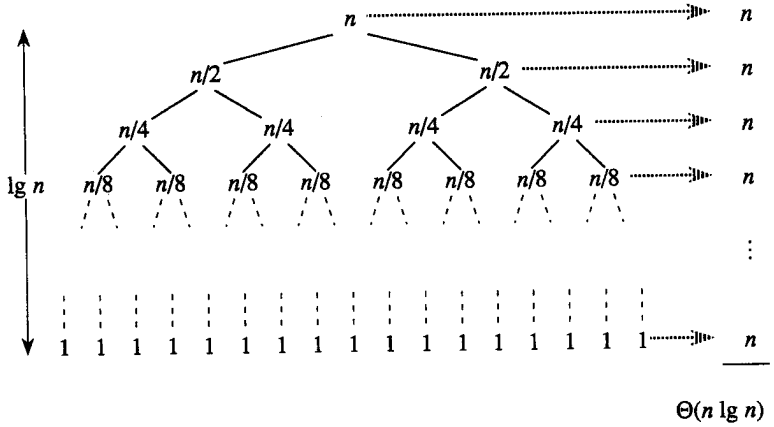
$T(n) = T(k) + T(n-k-1) + O(n)$

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)



tion contains n-1 items and

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

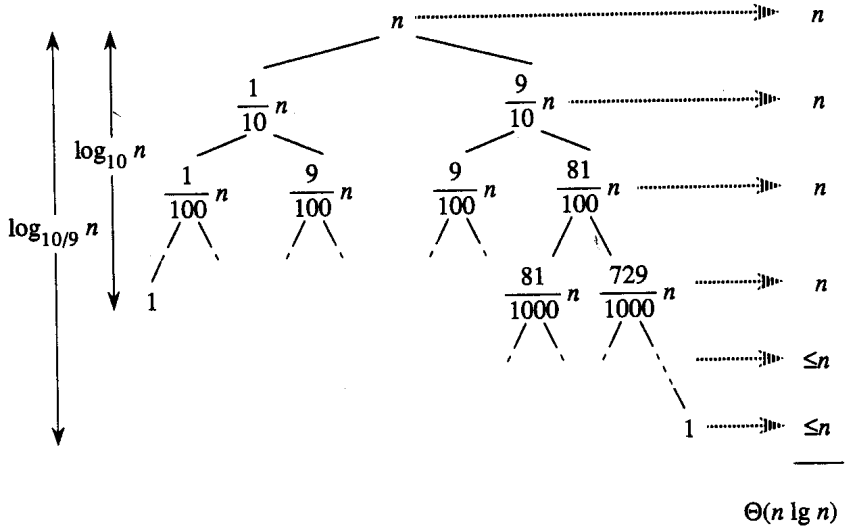
⇨ Time Complexity = $O(n^2)$

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(nolgn)



Average case:

All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree

Suppose we are alternate: Balanced, Unbalanced,Balanced, ….

B(n)= 2UB(n/2) + Θ(n) Balanced

UB(n)= B(n –1) + Θ(n) Unbalanced

Solving:

B(n)= 2(B(n/2 –1) + Θ(n/2)) + Θ(n)

= 2B(n/2 –1) + Θ(n)

= Θ(nlogn)

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);
}

    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:

Worst Case:

T(n) = worst-case running time

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

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

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

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

Proof of induction goal:

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

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

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

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

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

$\le cn^2$

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

- ⇨ $A_n = A_{n-1} + (2n+1)/n(n-1)$
- ⇨ $An = \sum_{i=1}^{n} 2i - 1/i(i+1)$
- ⇨ $An \approx \sum_{i=1}^{n} 2i/i(i+1)$
- ⇨ $An \approx 2\sum_{i=1}^{n} 1/(i+1)$
- ⇨ $An \approx 2\log n$

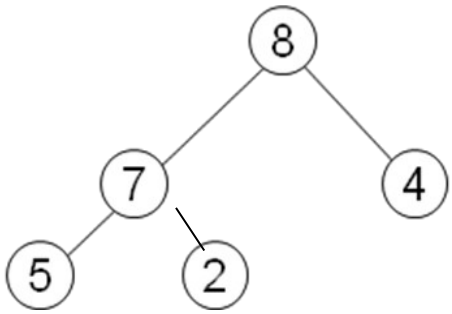
Since $An = T(n)/(n+1)$

$T(n) = n\log n$

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

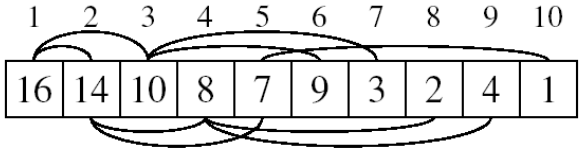


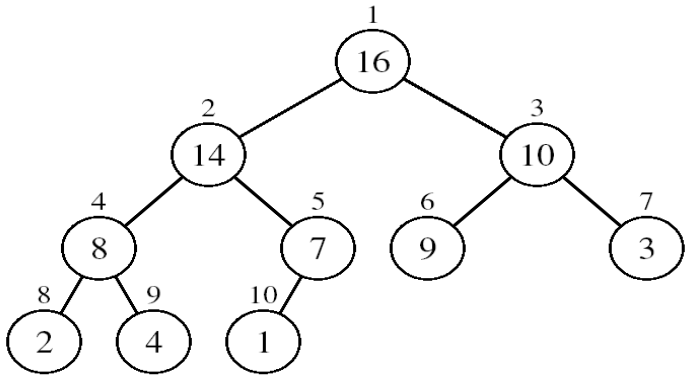
Array Representation of Heaps

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:

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

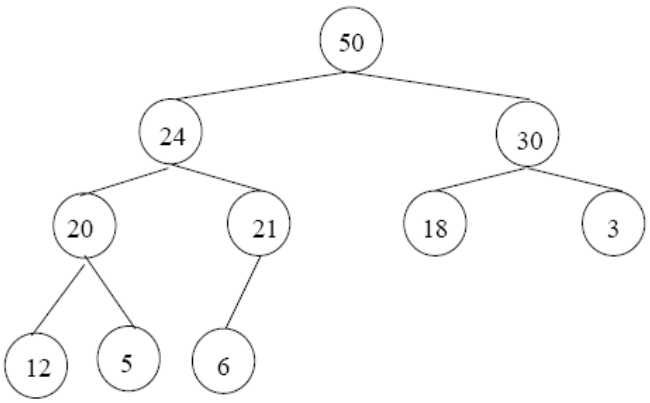
Min-heaps (smallest element at root), have the min-heap property:

- 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).



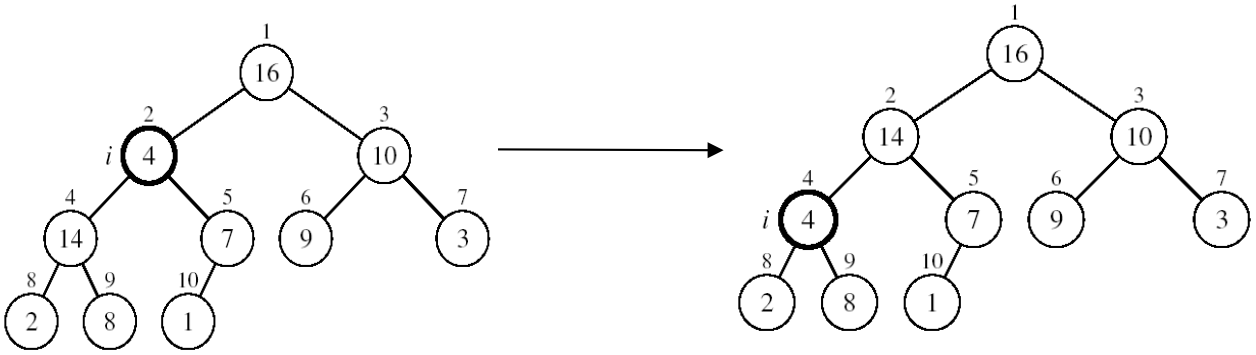
Operations on Heaps

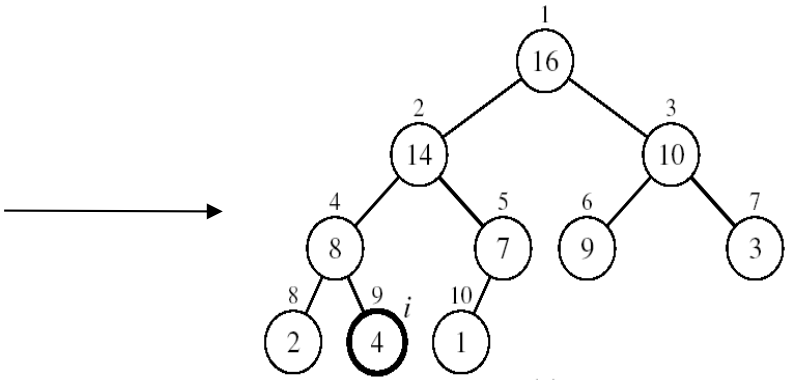
- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

Heapify 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)
}
```

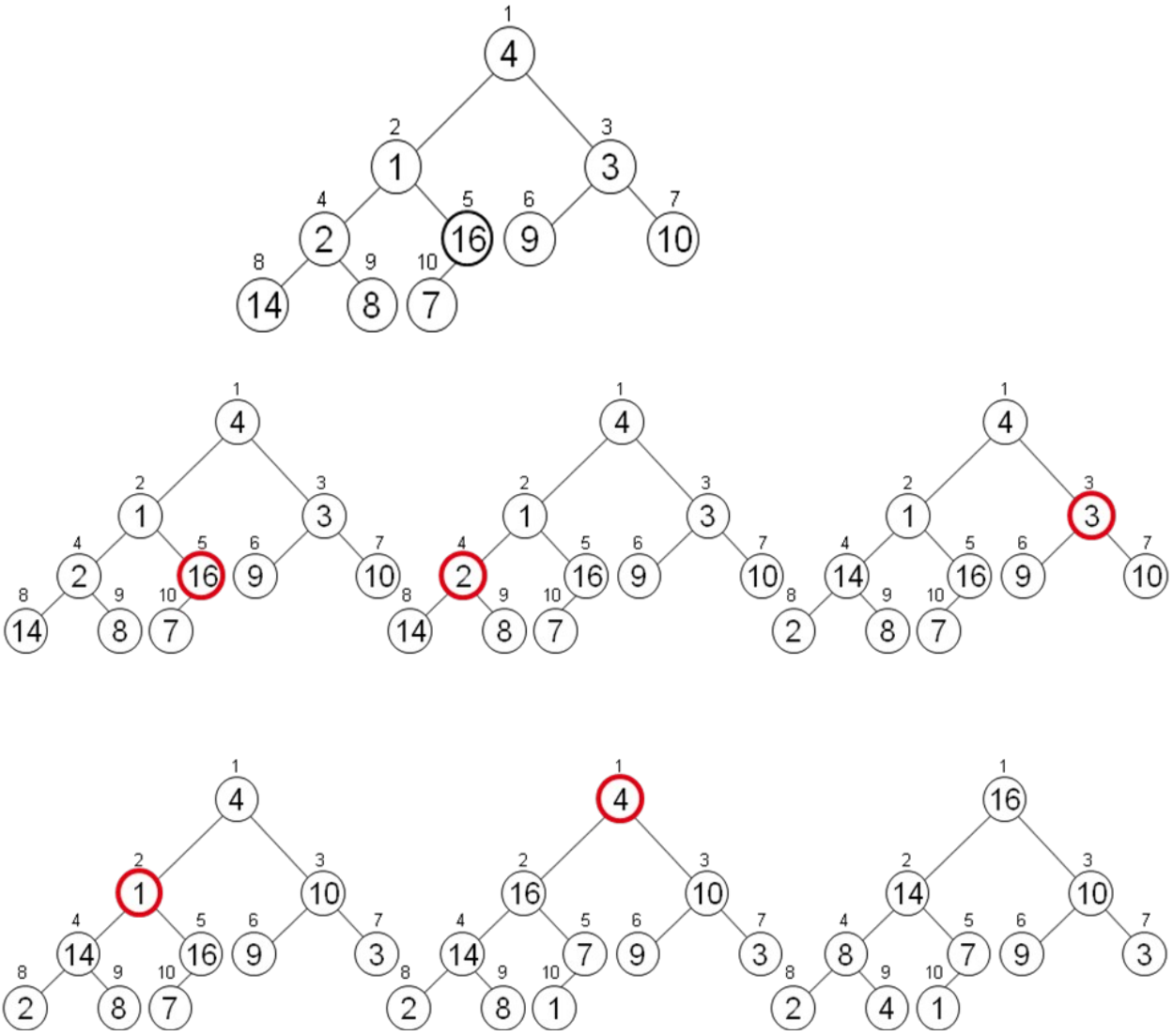
Analysis:

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 sub-array 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⌋ down to 1
            do MAX-HEAPIFY(A, i, n)
```

Time Complexity:

Running time: Loop executes O(n) times and complexity of Heapify is O(lgn), therefore complexity of Build-Max-Heap is O(nlogn).

This is not an asymptotically tight upper bound

Heapify takes O(h)

⇒ 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,

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

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

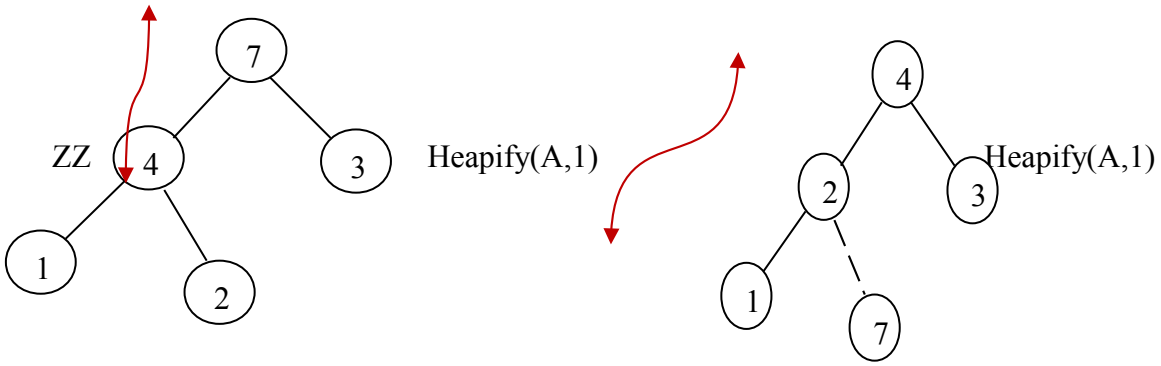
Put x=1/2

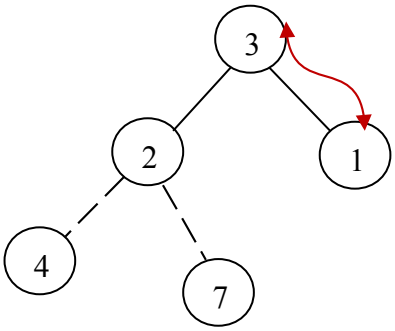
$\sum_{i=0}^{\infty} k /2^k = 1/(1-x)^2 = 2$

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

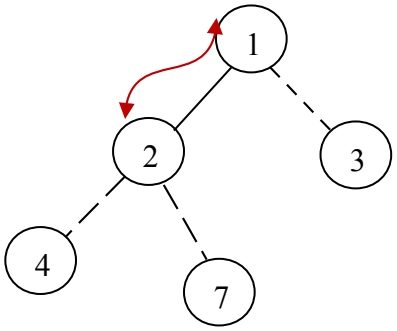
Heapsort

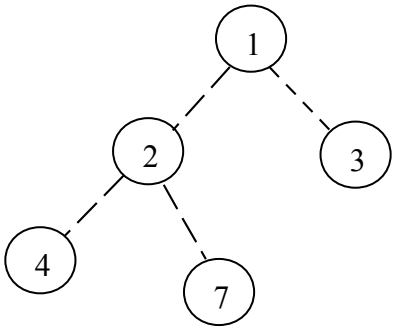
- Build a max-heap from the array
- Swap the root (the maximum element) with the last element in the array
- "Discard" this last node by decreasing the heap size
- Call Max-Heapify on the new root
- Repeat this process until only one node remains





Heapify(A,1)





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);
        }
}
```

Sorting comparisons:

Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Requires Memory
Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Small constants

Searching

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:

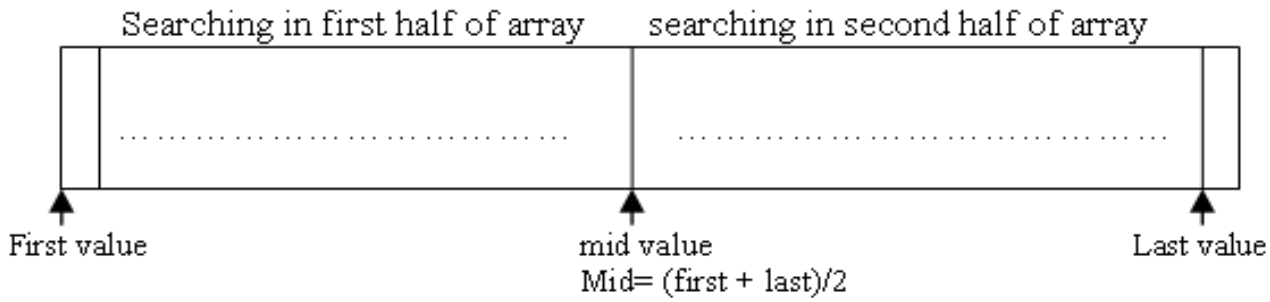
```
LinearSearch(A, n,key)
{
        for(i=0;i<n;i++)
        {
                if(A[i] == key)
                        return I;
        }

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

Analysis:

Time complexity = O(n)

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].



Take input array a[] = {2 , 5 , 7, 9 ,18, 45 ,53, 59, 67, 72, 88, 95, 101, 104}

For key = 2

low	high	mid	
0	13	6	key < A[6]
0	5	2	key < A[2]
0	1	0	

Terminating condition, since A[mid] = = 2, return 1(successful).

For key = 103

low	high	mid	
0	13	6	key > A[6]
7	13	10	key > A[10]
11	13	12	key > A[12]
13	13	-	

Terminating condition high = = low, since A[0] != 103, return 0(unsuccessful).

For key = 67

low	high	mid	
0	13	6	key > A[6]
7	13	10	key < A[10]
7	9	8	

Terminating condition, since A[mid] = 67, return 9(successful).

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 is:

$$T(n) = T(n/2) + Q(1)$$
$$= O(\log n) .$$

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).

Selection

i^{th} order statistic of a set of elements gives i^{th} largest(smallest) element. In general lets think of i^{th} 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 O(nlogn) in a very straightforward way. First sort the elements in Θ(nlogn) time and then pick up the i^{th} item from the array in constant time. What about the linear time algorithm for this problem? The next is answer to this.

Nonlinear general selection algorithm

We can construct a simple, but inefficient general algorithm for finding the *k*th smallest or *k*th largest item in a list, requiring O(*kn*) time, which is effective when *k* is small. To accomplish this, we simply find the most extreme value and move it to the beginning until we reach our desired index.

```
Select(A, k)
{
        for( i=0;i<k;i++)
        {
            minindex = i;
            minvalue = A[i];
           for(j=i+1;j<n;j++)
           {
              if( A[j] < minvalue)
              {
                    minindex = j;
                    minvalue = A[j];
              }
              swap(A[i],A[minIndex]);
         }
         return A[k];
}
```

Analysis:

When i=0, inner loop executes n-1 times
When i=1, inner loop executes n-2 times
When i=2, inner loop executes n-3 times
……...

When i=k-1 inner loop executes n-(k+1) times

Thus, Time Complexity = (n-1) + (n-2) + ……..(n-k-1)

$= O(kn) \approx O(n^2)$

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);
}
```

Analysis:

Since our algorithm is randomized algorithm no particular input is responsible for worst case however the worst case running time of this algorithm is $O(n^2)$. This happens if every time unfortunately the pivot chosen is always the largest one (if we are finding minimum element). Assume that the probability of selecting pivot is equal to all the elements i.e 1/n then we have the recurrence relation,

$$T(n) = 1/n\left(\sum_{j=1}^{n-1} T(\max(j,n-j))\right) + O(n)$$

Where, max(j,n-j) = j, if j>= ceil(n / 2)

and max(j,n-j) = n-j, otherwise.

Observe that every T(j) or T(n – j) will repeat twice for both odd and even value of n (one may not be repeated) one time form 1 to ceil(n / 2) and second time for ceil(n / 2) to n-1, so we can write,

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

Using substitution method,

Guess T(n) = O(n)

To show T(n) <= cn

Assume T(j) <= cj

Substituting on the relation

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

$$T(n) = 2/n \left\{\sum_{j=1}^{n-1} cj - \sum_{j=1}^{n/2-1} cj\right\} + O(n)$$

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

$T(n) \leq c(n-1) - c(n/2-1)/2 + O(n)$

$T(n) \leq cn - c - cn/4 + c/2 + O(n)$

$= cn - cn/4 - c/2 + O(n)$

$\leq cn$ {choose the value of c such that $(-cn/4-c/2 + O(n) \leq 0$ }

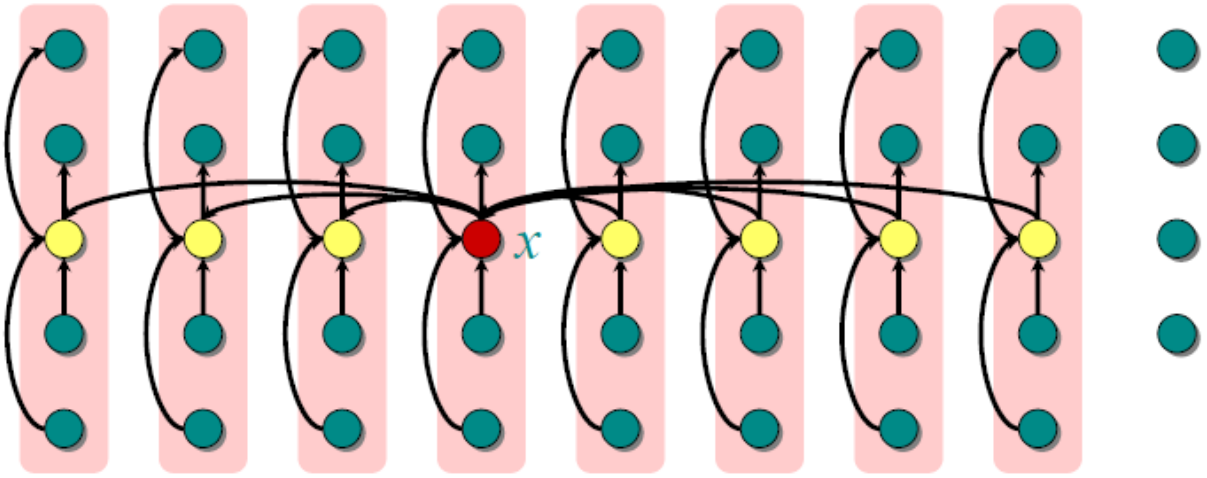
⇨ $T(n) = O(n)$

Selection in worst case linear time[Note:- this algorithm must be see in Class Note]

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 $\lfloor n/5 \rfloor$ 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 $T(n) \leq cn$

Assume that our guess is true for all $k<n$

Now,

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

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

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

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

⇨ $T(n) = O(n)$

Max and Min Finding

Here our problem is to find the minimum and maximum items in a set of n elements. **Iterative**

Divide and Conquer Algorithm for finding min-max:

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
                {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 $n \leq 2$

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

Matrix Multiplication

Given two A and B n-by-n matrices our aim is to find the product of A and B as C that is also n-by-n matrix. We can find this by using the relation

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$$

```
MatrixMultiply(A,B)
{
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            for(k=0;k<n;k++)
            {
                C[i][j] = C[i][j]+ A[i][k]*B[k][j];
            }
        }
    }
}
```

Analysis:

Using the above formula we need O(n) time to get C(i,j). There are n^2 elements in C hence the time required for matrix multiplication is $O(n^3)$. We can improve the above complexity by using divide and conquer strategy.

Divide and Conquer Algorithm for Matrix Multiplication

Divide the n x n square matrix into four matrices of size n/2 x n/2. The basic calculation is done for matrix of size 2 x 2 .

$$\begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix} = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}$$

Where

$C11 = A11 \times B11 + A12 \times B21$

$C12 = A11 \times B12 + A12 \times B22$

$C21 = A21 \times B11 + A22 \times B21$

$C22 = A21 \times B12 + A22 \times B22$

Now, We can write recurrence relation for this as

$T(n) = b$ if $n \leq 2$

$T(n) = 8T(n/2) + cn^2$ if $n > 2$

Solving this we get, $T(n) = O(n^3)$

Strassens's Matrix Multiplication

Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions.

The basic calculation is done for matrix of size 2 x 2 .

$$\begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix} = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}$$

Where;

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$P_2 = (A_{21} + A_{22}) * B_{11}$

$P_3 = A_{11} * (B_{12} - B_{22})$

$P_4 = A_{22} * (B_{21} - B_{11})$

$P_5 = (A_{11} + A_{12}) * B_{22}$

$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P_1 + P_4 - P_5 + P_7$

$C_{12} = P_3 + P_5$

$C_{21} = P_2 + P_4$

$C_{22} = P_1 + P_3 - P_2 + P_6$

Now, We can write recurrence relation for this as

$T(n) = b$ if $n \leq 2$

$T(n) = 7T(n/2) + cn^2$ if $n > 2$

Solving this we get, $T(n) = O(n^{2.81})$

Unit 2
Chapter:2
Dynamic Programming

Dynamic Programming: 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)

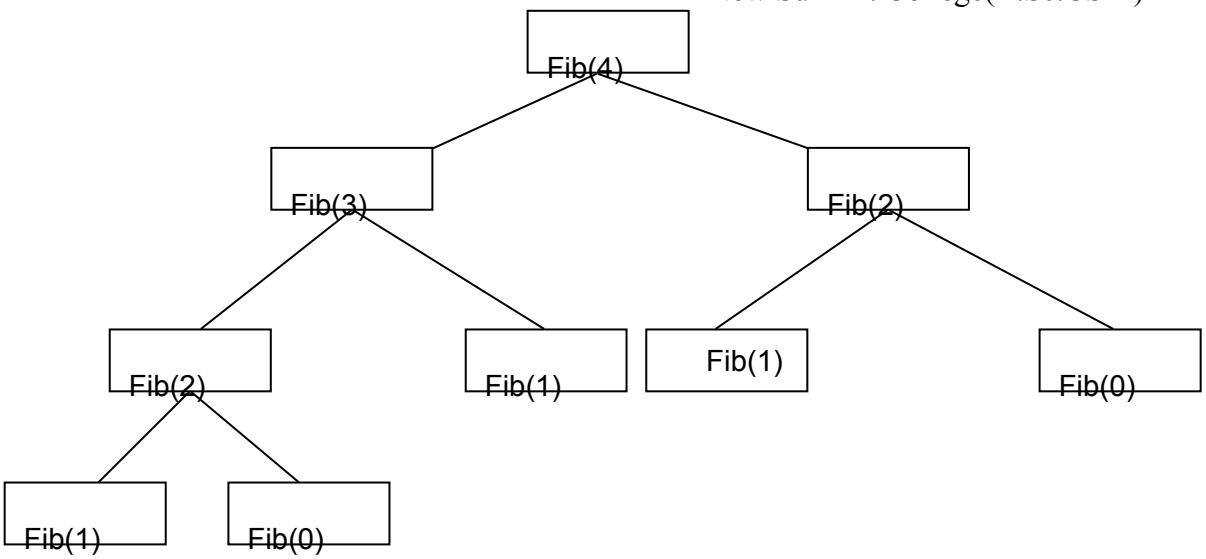
The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the formulation of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to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

Fibonacci numbers

Recursive Fibonacci revisited: In recursive version of an algorithm for finding Fibonacci number we can notice that for each calculation of the Fibonacci number of the larger number we have to calculate the Fibonacci number of the two previous numbers regardless of the computation of the Fibonacci number that has already be done. So there are many redundancies in calculating the Fibonacci number for a particular number. Let's try to calculate the Fibonacci number of 4. The representation shown below shows the repetition in the calculation.



In the above tree we saw that calculations of fib(0) is done two times, fib(1) is done 3 times, fib(2) is done 2 times, and so on. So if we somehow eliminate those repetitions we will save the running time.

Algorithm:

```
DynaFibo(n)
{
        A[0] = 0, A[1]= 1;
        for(i = 2 ; i <=n ; i++)
        A[i] = A[i-2] +A[i-1] ;
        return A[n] ;
}
```

Analysis

Analyzing the above algorithm we found that there are no repetition of calculation of the sub-problems already solved and the running time decreased from $O(2^{n/2})$ to O(n). This reduction was possible due to the remembrance of the sub-problem that is already solved to solve the problem of higher size.

0/1 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 i^{th} 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

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

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];
                                }
                        }
                        else
                        {
                                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

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} x 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} x 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))

Longest Common Subsequence Problem

Given two sequences $X = (x_1, x_2, \ldots\ldots\ldots, x_m)$ and $Z = (z_1; z_2; \ldots\ldots\ldots ; z_k)$, we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices $(i_1, i_2, \ldots\ldots\ldots i_k)$ $(1 \le i_1 < i_2 < \ldots\ldots\ldots < i_k)$ such that $Z = (X_{i1}, X_{i2} \ldots\ldots\ldots X_{ik})$.

For example, let X = (ABRACADABRA) and let Z = (AADAA), then Z is a subsequence of X. Given two strings X and Y , the longest common subsequence of X and Y is a longest sequence Z that is a subsequence of both X and Y . For example, let X = (ABRACADABRA) and let Y = (YABBADABBAD). Then the longest common subsequence is Z = (ABADABA)

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences X = (x1; : : : ; x_m) and Y = (y, ……., y_n) determine a longest common subsequence.

DP Formulation for LCS:

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ is called i^{th} prefix of X, here we have X_0 as empty sequence. Now in case of sequences X_i and Y_j:

If $x_i = y_j$ (i.e. last Character match) , we claim that the LCS must also contail charater x_i or y_j .

If $x_i \neq y_j$ (i.e Last Character do not match) , In this case x_i and y_j cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is not part of the LCS, or y_j is not part of the LCS (and possibly both are not part of the LCS). Let L[I,j] represents the length of LCS of sequences X_i and Y_j.

$$L[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ L[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{L[i-1,j], L[i,j-1]\} & \text{if } i>0 \text{ and } w>wi \end{cases}$$

Algorithm:

```
LCS(X,Y)
{
        m = length[X];
        n = length[Y];
        for(i=1;i<=m;i++)
                c[i,0] = 0;
        for(j=0;j<=n;j++)
                c[0,j] = 0;
        for(i = 1;i<=m;i++)
        for(j=1;j<=n;j++)
        {
                if(X[i]==Y[j])
                {
                        c[i][j] = c[i-1][j-1]+1; b[i][j] = "upleft";
                }
                else if(c[i-1][j]>= c[i][j-1])
```

```
        {
                c[i][j] = c[i-1][j]; b[i][j] = "up";
        }
        else
        {
        c[i][j] = c[i][j-1]; b[i][j] = "left";
        }
    }
    return b and c;
}
```

Analysis:
The above algorithm can be easily analyzed for running time as O(mn), due to two nested loops. The space complexity is O(mn).

Example:
Consider the character Sequences X=abbabba Y=aaabba

X \ Y	Φ	a	a	a	b	b	a
Φ	0	0	0	0	0	0	0
a	0	1 upleft	1upleft	1upleft	1 left	1 left	1 left
b	0	1 up	1 up	1 up	2 upleft	2 upleft	2 left
b	0	1 up	1 up	1 up	2 upleft	3 upleft	3 left
a	0	1 upleft	2 upleft	2 upleft	2 up	3 up	4 upleft
b	0	1 up	2 up	2 up	3 upleft	3 upleft	4 up
b	0	1 up	2 up	2 up	3 upleft	4 upleft	4 up
a	0	1 upleft	2 upleft	3 upleft	3 up	4 up	5 upleft

LCS = aabba

Chapter:3
Greedy Paradigm

Greedy method is the simple straightforward way of algorithm design. The general class of problems solved by greedy approach is optimization problems. In this approach the input elements are exposed to some constraints to get feasible solution and the *feasible solution* that meets some objective function best among all the solutions is called *optimal solution*. Greedy algorithms always makes optimal choice that is local to generate globally optimal solution however, it is not guaranteed that all greedy algorithms yield optimal solution. We generally cannot tell whether the given optimization problem is solved by using greedy method or not, but most of the problems that can be solved using greedy approach have two parts:

Greedy choice property

Globally optimal solution can be obtained by making locally optimal choice and the choice at present cannot reflect possible choices at future.

Optimal substructure

Optimal substructure is exhibited by a problem if an optimal solution to the problem contains optimal solutions to the sub-problems within it.
To prove that a greedy algorithm is optimal we must show the above two parts are exhibited. For this purpose first take globally optimal solution; then show that the greedy choice at the first step generates the same but the smaller problem, here greedy choice must be made at first and it should be the part of an optimal solution; at last we should be able to use induction to prove that the greedy choice at each step is best at each step, this is optimal substructure.

Fractional 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 i_{th} 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;
                tempW = W;
                for(i=1; i<=n; i++)
                {
                        if(w[i] > tempW) then break;
                                x[i] = 1.0;
                                tempW -= w[i];
                }
                if(i<=n)
                        x[i] = tempW/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).

Huffman Codes:

Huffman codes are used to compress data by representing each alphabet by unique binary codes in an optimal way. As an example consider the file of 100,000 characters with the following frequency distribution assuming that there are only 7 characters

f(a) = 40,000 , f(b) = 20,000 , f(c) = 15,000 , f(d) = 12,000 , f(e) = 8,000 , f(f) = 3,000 , f(g) = 2,000.

Here fixed length code for 7 characters we need 3 bits to represent all characters like

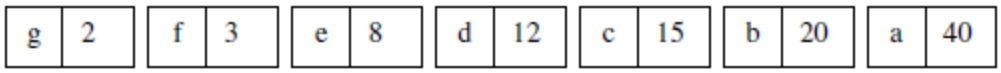
a = 000 , b = 001 , c = 010 , d = 011 , e = 100 , f = 101 , g = 110.

Total number of bits required due to fixed length code is 300,000.

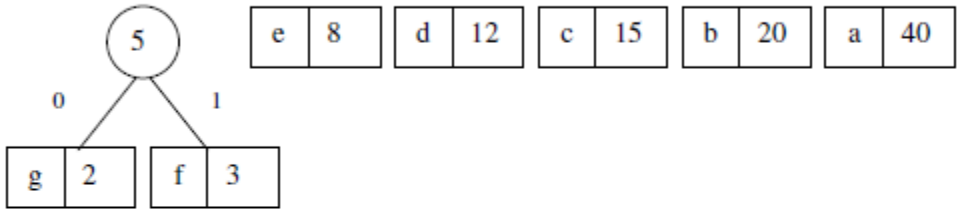
Now consider variable length character so that character with highest frequency is given smaller codes like

C = {a, b, c, d, e, f, g}; f(c) = 40, 20, 15, 12, 8, 3, 2; n = 7

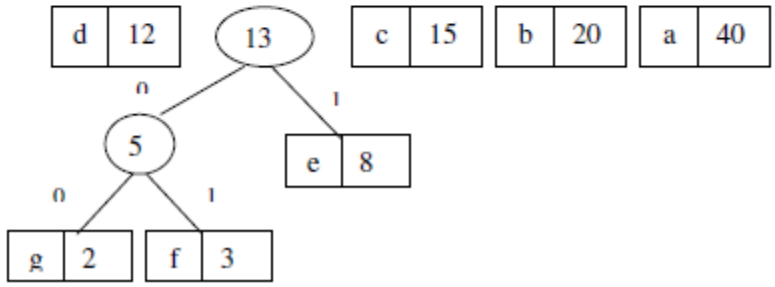
Initial priority queue is



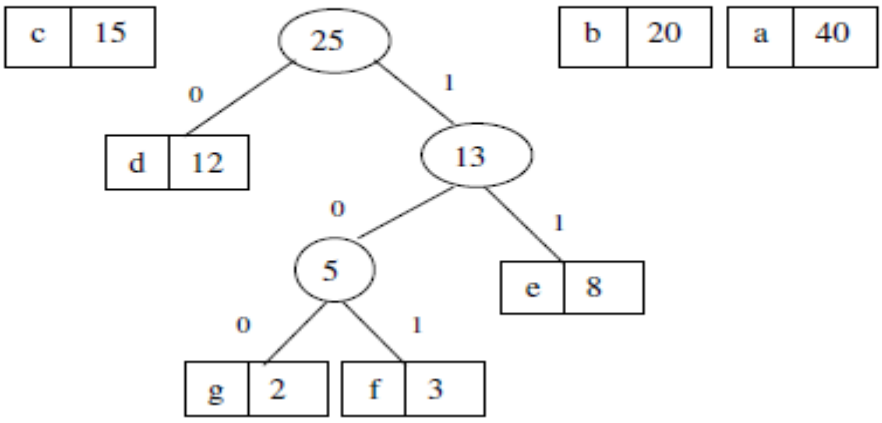
i = 1

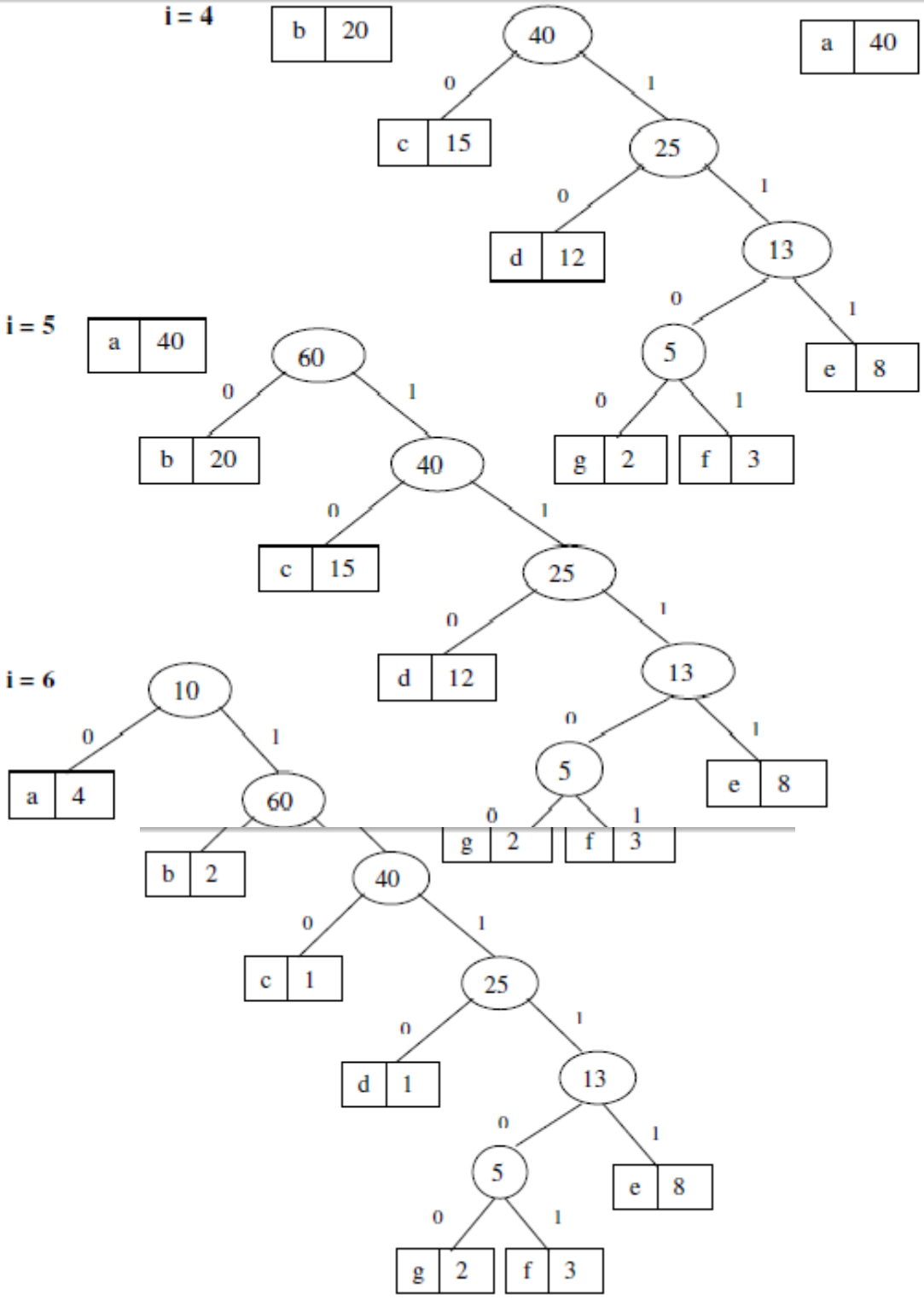


i = 2



i = 3





a = 0 , b = 10 , c = 110 , d = 1110 , e = 11111 , f = 111101 , g = 111100
Total number of bits required due to variable length code is
40,000*1 + 20,000*2 + 15,000*3 + 12,000*4 + 8,000*5 + 3,000*6 + 2,000*6.
i.e. 243,000 bits
Here we saved approximately 19% of the space.

Analysis

We can use *BuildHeap(C)* {see notes on sorting} to create a priority queue that takes O(n) time. Inside the for loop the expensive operations can be done in O(logn) time. Since operations inside for loop executes for n-1 time total running time of *HuffmanAlgo* is O(nlogn).

Job Sequencing with Deadline:

We are given a set of n jobs. Associated with each job I, di≥0 is an integer deadline and pi≥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, $(p_1,p_2,p_3,p_4)=(100,10,15,27)$, $(d_1,d_2,d_3,d_4)=(2,1,2,1)$

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	5
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=ϕ

- 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. Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.

```
JobSequencing(int d[], int j[], int n)
 {
        d[0] = J[0] = 0; // Initialize.
        J[1] = 1; // Include job 1.
```

```
        int k=1;
        for (int i=2; i<=n; i++)
        {//Consider jobs in nonincreasing order of p[i]. Find position for i and check
        feasibility of insertion.
                int r = k;
                while ((d[J[r]] > d[i]) && (d[J[r]] != r))
                        r--;
                if ((d[J[r]] <= d[i]) && (d[i] > r))
                {
                        // Insert i into J[].
                        for (int q=k; q>=(r+1); q--)
                                J[q+1] = J[q];
                        J[r+1] = i; k++;
                }
        }
    return (k);
}
```

Analysis

For loop executes O(n) line. While loop inside the for loop executes at most times and if the condition given inside if statement is true inner for loop executes O(k-r) times. Hence total time for each iteration of outer for loop is O(k). Thus time complexity is $O(n^2)$.

Unit 3:-

Graph Algorithms

Graph is a collection of vertices or nodes, connected by a collection of edges. Graphs are extremely important because they are a very flexible mathematical model for many application problems. Basically, any time you have a set of objects, and there is some "connection" or "relationship" or "interaction" between pairs of objects, a graph is a good way to model this. Examples of graphs in application include communication and transportation networks, VLSI and other sorts of logic circuits, surface meshes used for shape description in computer-aided design and geographic information systems, precedence constraints in scheduling systems etc.

A directed graph (or digraph) G = (V,E) consists of a finite set V , called the vertices or nodes, and E, a set of ordered pairs, called the edges of G.
An undirected graph (or graph) G = (V,E) consists of a finite set V of vertices, and a set E of unordered pairs of distinct vertices, called the edges.

Digraph and graph example.

We say that vertex v is adjacent to vertex u if there is an edge (u; v). In a directed graph, given the edge e = (u; v), we say that u is the origin of e and v is the destination of e. In undirected graphs u and v are the endpoints of the edge. The edge e is incident (meaning that it touches) both u and v.

In a digraph, the number of edges coming out of a vertex is called the out-degree of that vertex, and the number of edges coming in is called the in-degree. In an undirected graph we just talk about the degree of a vertex as the number of incident edges. By the degree of a graph, we usually mean the maximum degree of its vertices.

Notice that generally the number of edges in a graph may be as large as quadratic in the number of vertices. However, the large graphs that arise in practice typically have much fewer edges. A graph is said to be sparse if E = Θ(V), and dense, otherwise. When giving the running times of algorithms, we will usually express it as a function of both V and E, so that the performance on sparse and dense graphs will be apparent.

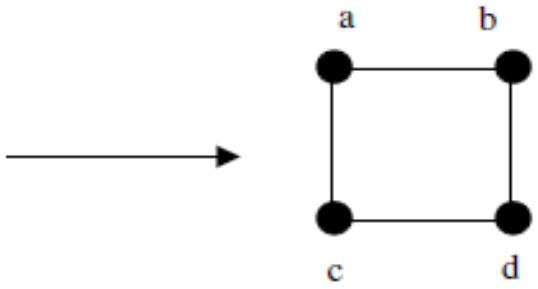
Representation of Graph

Generally graph can be represented in two ways namely adjacency lists(Linked list representation) and adjacency matrix(matrix).

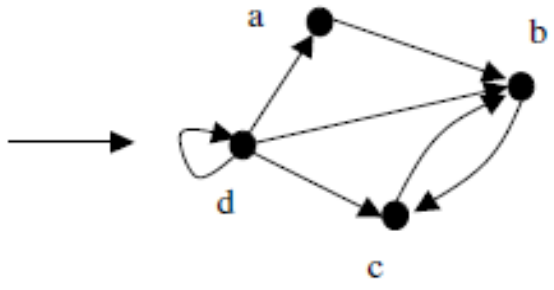
Adjacency List:

This type of representation is suitable for the undirected graphs without multiple edges, and directed graphs. This representation looks as in the tables below.

Edge List for Simple Graph	
Vertex	Adjacent Vertices
a	b, c
b	a, d
c	a, d
d	b, c



Edge List for Directed Graph	
Initial Vertex	End Vertices
a	b
b	c
c	b
d	a, b, c, d



If we try to apply the algorithms of graph using the representation of graphs by lists of edges, or adjacency lists it can be tedious and time taking if there are high number of edges. For the sake of the computation, the graphs with many edges can be represented in other ways. In this

class we discuss two ways of representing graphs in form of matrix.

Adjacency Matrix:

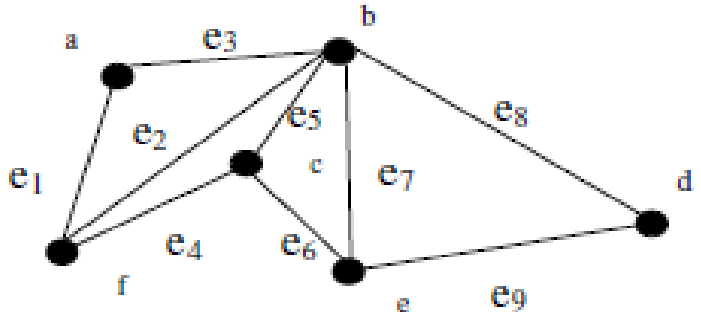
Given a simple graph G =(V, E) with |V| = n. assume that the vertices of the graph are listed in some arbitrary order like $v_1, v_2, \ldots, v_n$. The adjacency matrix A of G, with respect to the order of the vertices is n-by-n zero-one matrix ($A = [a_{ij}]$) with the condition,

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of G,} \\ 0 & \text{otherwise.} \end{cases}$$

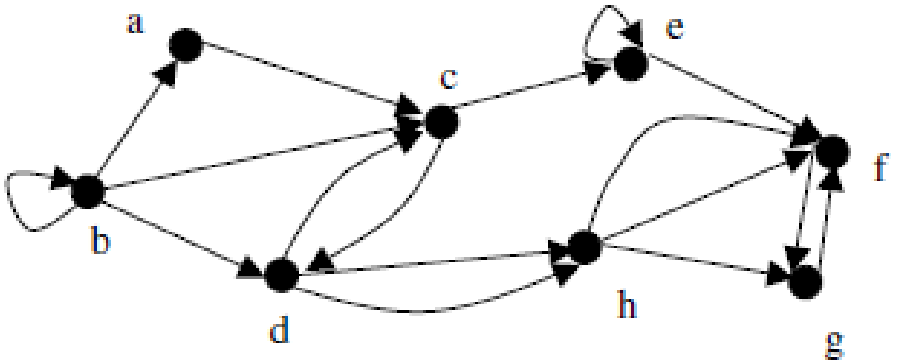
Since there are n vertices and we may order vertices in any order there are n! possible order of the vertices. The adjacency matrix depends on the order of the vertices, hence there are n! possible adjacency matrices for a graph with n vertices.
In case of the directed graph we can extend the same concept as in undirected graph as dictated by the relation

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of G,} \\ 0 & \text{otherwise.} \end{cases}$$

If the number of edges is few then the adjacency matrix becomes sparse. Sometimes it will be beneficial to represented graph with adjacency list in such a condition.
Solution:Let the order of the vertices be a, b, c, d, e, f



Let us take a directed graph



Solution:
Let the order of the vertices be a, b, c, d, e, f, g

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \end{bmatrix}$$

Graph Traversals

There are a number of approaches used for solving problems on graphs. One of the most important approaches is based on the notion of systematically visiting all the vertices and edge of a graph. The reason for this is that these traversals impose a type of tree structure (or generally a forest) on the graph, and trees are usually much easier to reason about than general graphs.

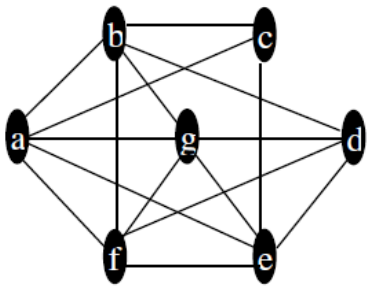
Breadth-first search

This is one of the simplest methods of graph searching. Choose some vertex arbitrarily as a root. Add all the vertices and edges that are incident in the root. The new vertices added will become the vertices at the level 1 of the BFS tree. Form the set of the added vertices of level 1, find other vertices, such that they are connected by edges at level 1 vertices. Follow the above step until all the vertices are added.

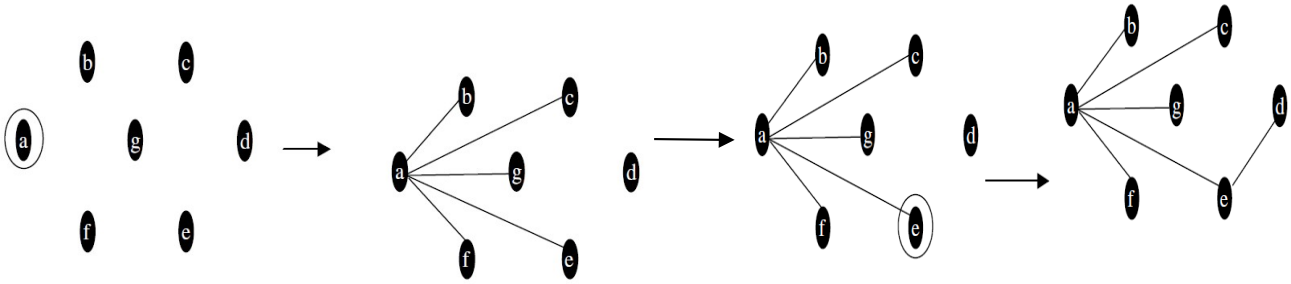
Algorithm:

```
BFS(G,s) //s is start vertex
{
        T = {s};
        L =Φ; //an empty queue
        Enqueue(L,s);
        while (L != Φ )
        {
                v = dequeue(L);
                for each neighbor w to v
                        if ( w∉ L and w ∉ T )
                        {
                                enqueue( L,w);
                                T = T U {w}; //put edge {v,w} also
                        }
        }
}
```

Example:
Use breadth first search to find a BFS tree of the following graph.



Solution:



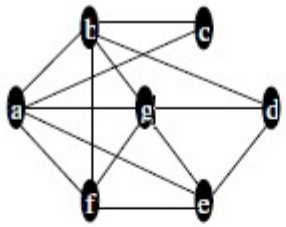
Analysis

From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are looked for and this can be done in O(n) time(for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue i.e. n, produce complexity of an algorithm as $O(n^2)$. Also from aggregate analysis we can write the complexity as O(E+V) because inner loop executes E times in total.

Depth First Search

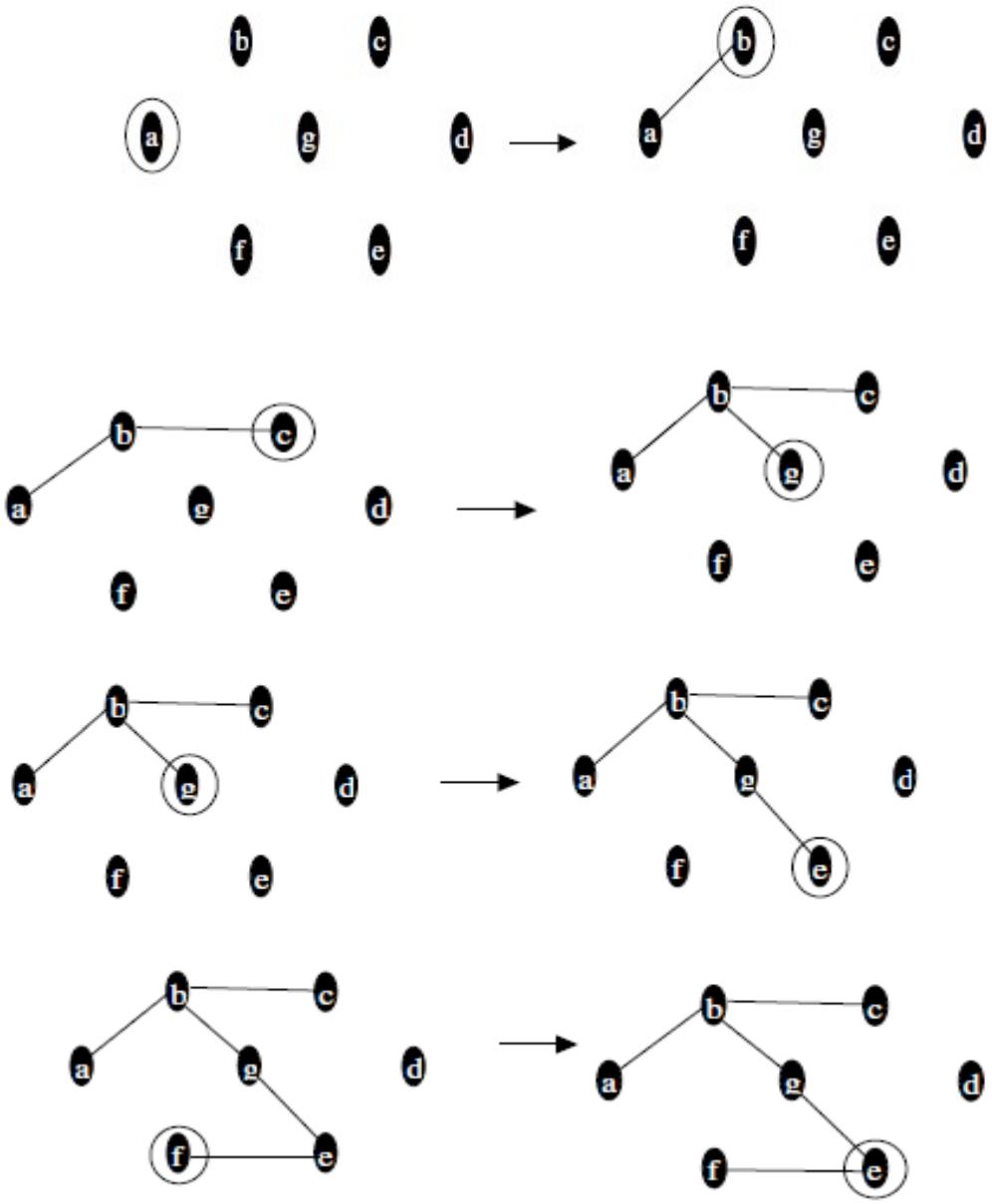
This is another technique that can be used to search the graph. Choose a vertex as a root and form a path by starting at a root vertex by successively adding vertices and edges. This process is continued until no possible path can be formed. If the path contains all the vertices then the tree consisting this path is DFS tree. Otherwise, we must add other edges and vertices. For this move back from the last vertex that is met in the previous path and find whether it is possible to find new path starting from the vertex just met. If there is such a path continue the process above. If this cannot be done, move back to another vertex and repeat the process. The whole process is continued until all the vertices are met. This method of search is also called **backtracking**.

Example:
Use depth first search to find a spanning tree of the following graph.



Solution:

Choose a as initial vertex then we have



Algorithm:

```
DFS(G,s)
{
        T = {s};
        Traverse(s);
}
Traverse(v)
{
        for each w adjacent to v and not yet in T
        {
                T = T U {w}; //put edge {v,w} also
                Traverse (w);
        }
}
```

Analysis:

The complexity of the algorithm is greatly affected by **Traverse** function we can write its running time in terms of the relation T(n) = T(n-1) + O(n), here O(n) is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find that the complexity of an algorithm is $O(n^2)$.

Also from aggregate analysis we can write the complexity as O(E+V) because traverse function is invoked V times maximum and for loop executes O(E) times in total.

Minimum Spanning Tree

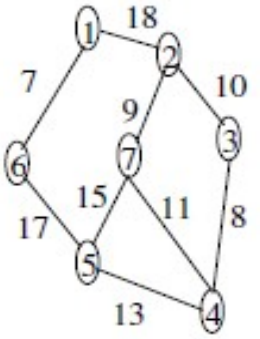
Given an undirected graph G = (V,E), a subgraph T =(V,E') of G is a spanning tree if and only if T is a tree. The MST is a spanning tree of a connected weighted graph such that the total sum of the weights of all edges eÎE' is minimum amongst all the sum of edges that would give a spanning tree.

Kruskal's Algorithm:

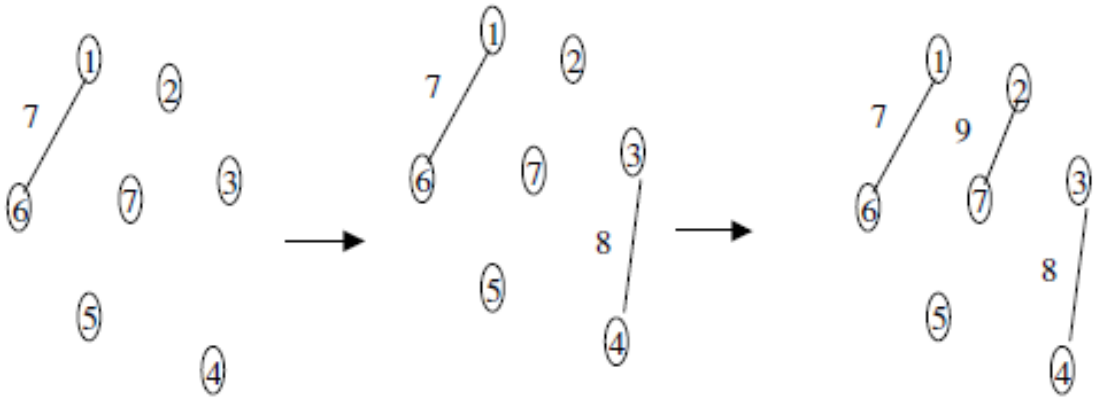
The problem of finding MST can be solved by using Kruskal's algorithm. The idea behind this algorithm is that you put the set of edges form the given graph G = (V,E) in nondecreasing order of their weights. The selection of each edge in sequence then guarantees that the total cost that would from will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

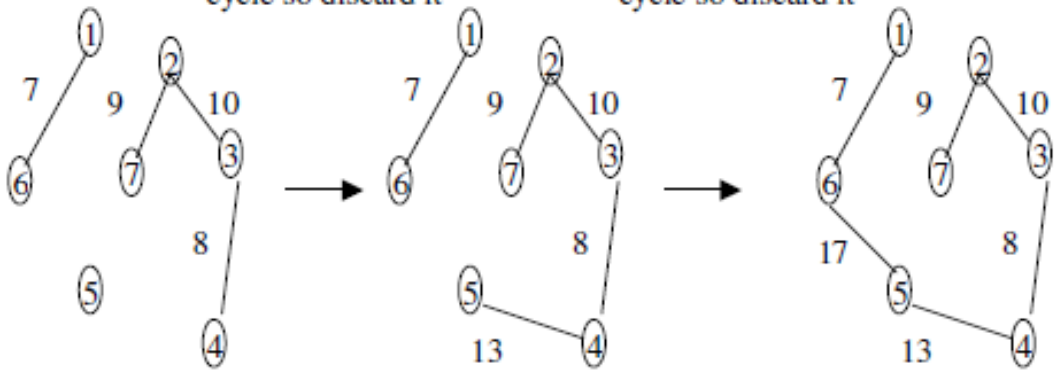


Solution:



Edge with weight 11 forms cycle so discard it

Edge with weight 15 forms cycle so discard it



The total weight of MST is 64.

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 !=∅)
    {
        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:

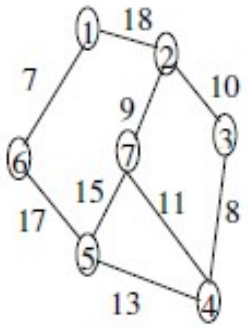
In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes O(ElogE) time and while loop execute O(n) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(ElogE) or asymptotically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

Prim's Algorithm

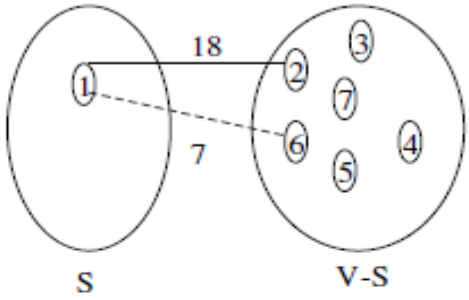
This is another algorithm for finding MST. The idea behind this algorithm is just take any arbitrary vertex and choose the edge with minimum weight incident on the chosen vertex. Add the vertex and continue the above process taking all the vertices added. Remember the cycle must be avoided.

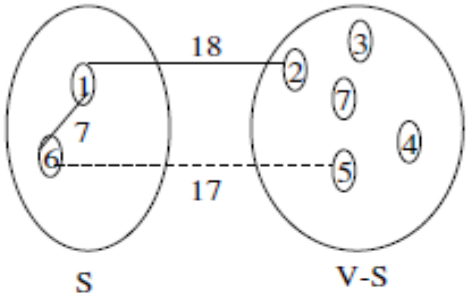
Example:

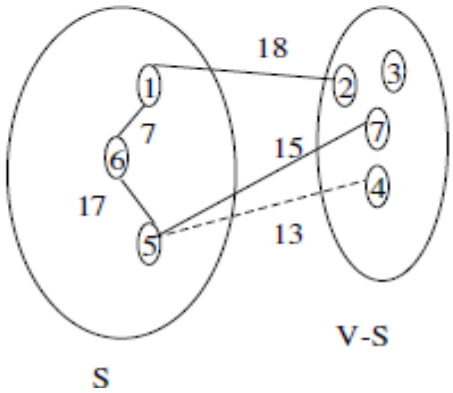
Find the minimum spanning tree of the following graph.

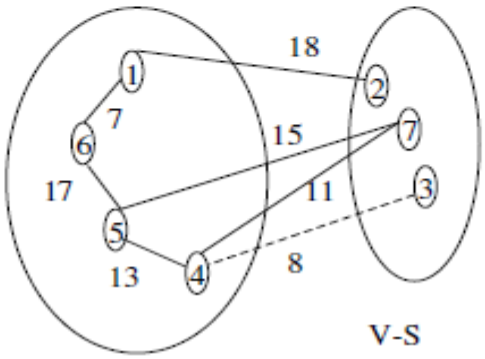


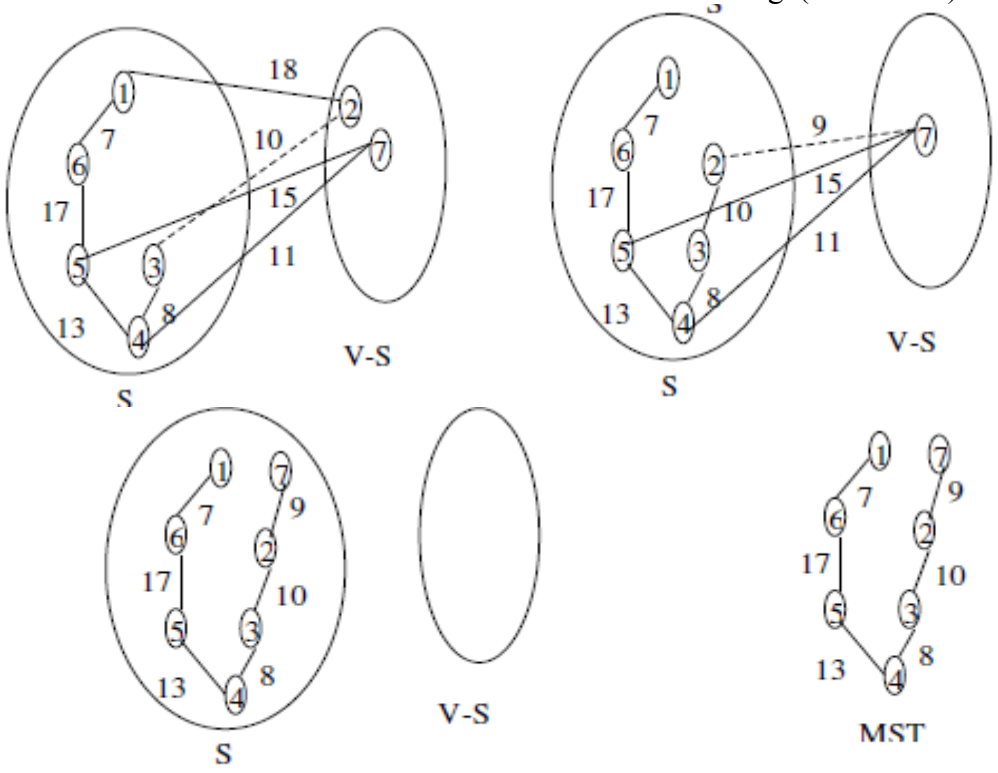
Solution: note: dotted edge is chosen.











The total weight of MST is 64.

Algorithm:

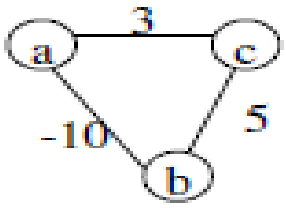
PrimMST(G)
{
T = ∅; // T is a set of edges of MST
S = {s} ; //s is randomly chosen vertex and S is set of vertices
while(S != V)
{
e = (u,v) an edge of minimum weight incident to vertices in T and not forming a simple circuit in T if added to T i.e. u ∈ S and v∈ V-S
T = T ∪ {(u,v)};
S = S ∪ {v};
}
}

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)!.

Shortest Path Problem

Given a weighted graph G =(V,E), then it has weight for every path p = $<v_0,v_1,\ldots v_k>$ as $w(p) = w(v_0,v_1) + w(v_1,v_2) + \ldots + w(v_{k-1},v_k)$. A shortest path from u to v is the path from u to v with minimum weight. Shortest path from u to v is denoted by d(u,v). It is important to remember that the shortest path may exist in a graph or may not i.e. if there is negative weight cycle then there is no shortest path. For e.g the below graph has no shortest path from **a** to **c** .You can notice the negative weight cycle for path a to b.



As a matter of fact even the positive weight cycle doesn't constitute shortest path but there will be shortest path. Some of the variations of shortest path problem include:

Single Source: This type of problem asks us to find the shortest path from the given vertex (source) to all other vertices in a connected graph

Single Destination: This type of problem asks us to find the shortest path to the given vertex (destination) from all other vertices in a connected graph.

Single Pair: This type of problem asks us to find the shortest path from the given vertex (source) to another given vertex (destination).

All Pairs: This type of problem asks us to find the shortest path from the all vertices to all other vertices in a connected graph

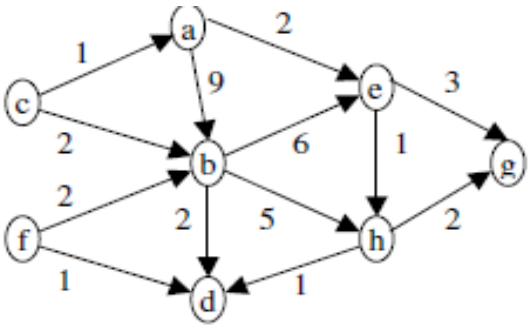
Single Source Problem

Relaxation: Relaxation of an edge (u,v) is a process of testing the total weight of the shortest path to v by going through u and if we get the weight less than the previous one then replacing the record of previous shortest path by new one.

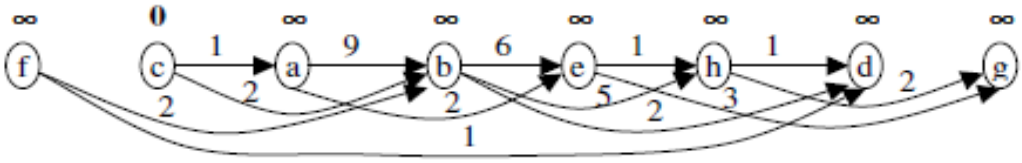
Directed Acyclic Graphs (Single Source Shortest paths)

Recall the definition of DAG, DAG is a directed graph G = (V,E) without a cycle. The algorithm that finds the shortest paths in a DAG starts by topologically sorting the DAG for getting the linear ordering of the vertices. The next step is to relax the edges as usual.

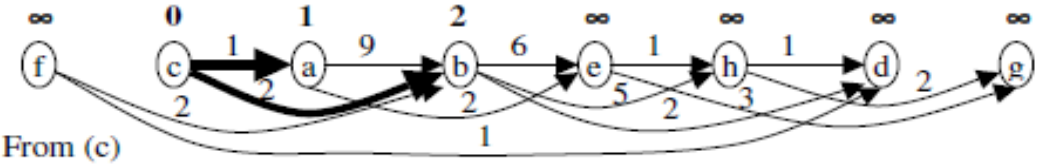
Example:
Find the shortest path from the vertex c to all other vertices in the following DAG.



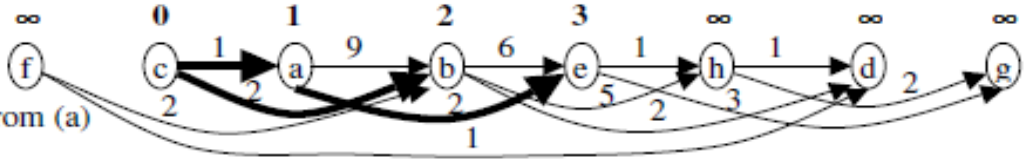
Solution:



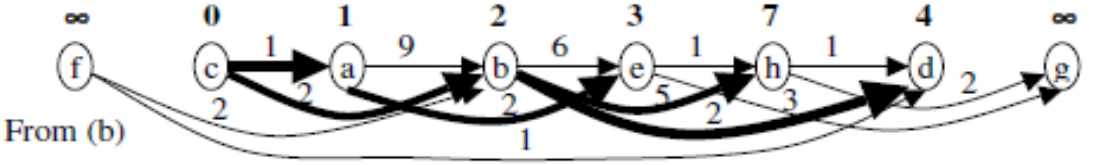
Topologically sorted and initialized.



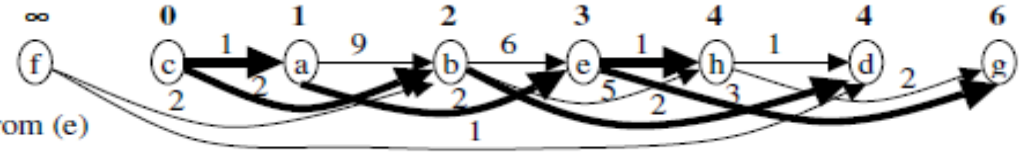
From (c)



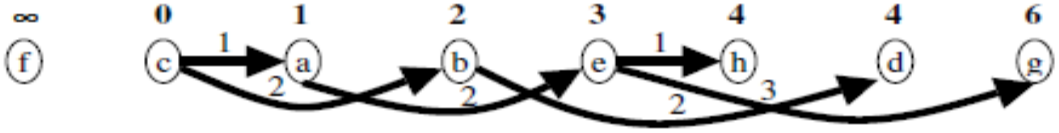
From (a)



From (b)



From (e)



From (h) (d) and (g) no change. So above is the shortest path tree.

Algorithm:

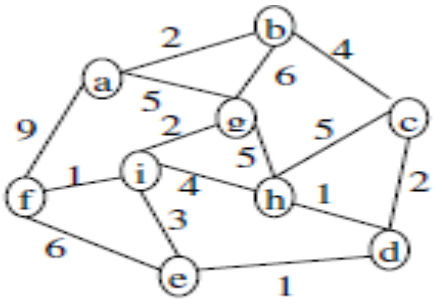
```
DagSP(G,w,s)
{
Topologically Sort the vertices of G
for each vertex v belongs to V
        do d[v] = ∞
d[s] = 0
for each vertex u, taken in topologically sorted order
        do for each vertex v adjacent to u
                do if d[v] > d[u] + w(u,v)
                        then d[v] = d[u] + w(u,v)
}
```

Dijkstra's Algorithm

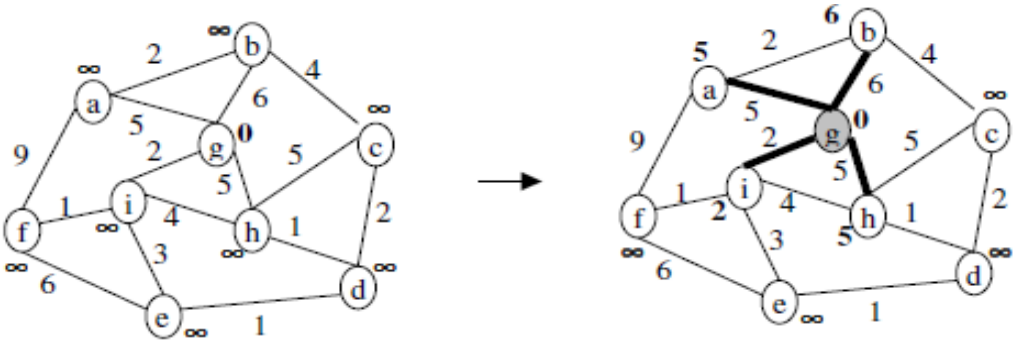
This is another approach of getting single source shortest paths. In this algorithm it is assumed that there is no negative weight edge. Dijkstra's algorithm works using greedy approach, as we will see later.

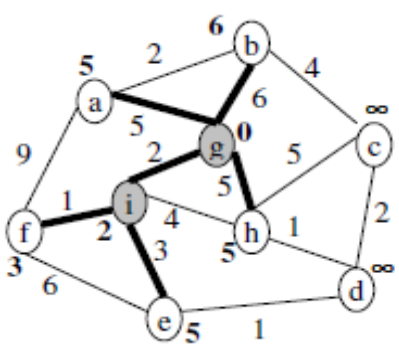
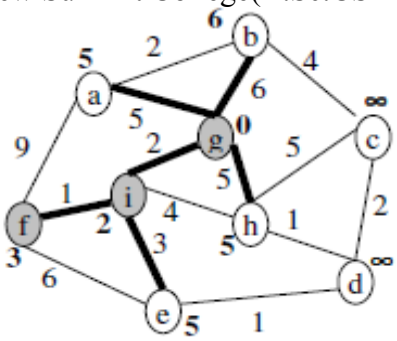
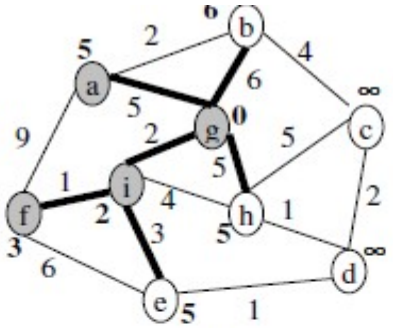
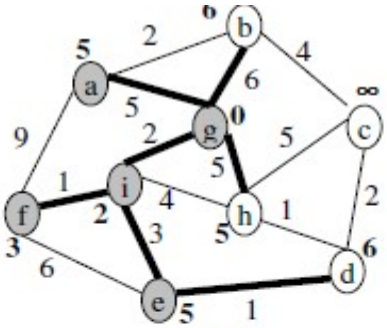
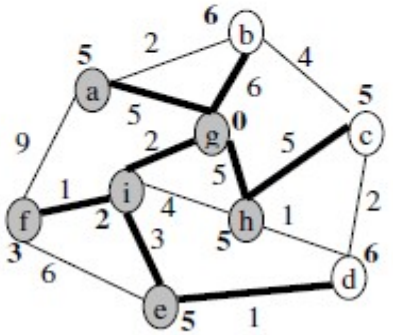
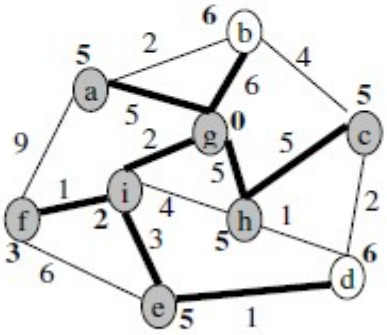
Example:

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



Solution:



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 = ∅
        Q = V
        While(Q!= ∅)
        {
        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 $O(V_2)$.

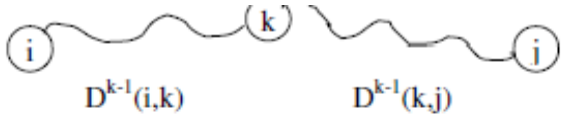
All Pairs Problem

As defined in above sections, we can apply single source shortest path algorithms |V| times to solve all pair shortest paths problem.

Flyod's Warshall 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 from 1,2,…,k as intermediate vertices in the ath. 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



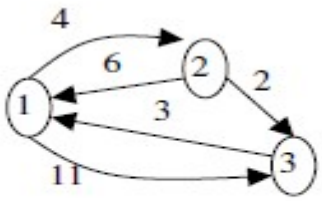
$D^k(i,j) = D^{k-1}(i,k) + D^{k-1}(k,j)$, when k is an intermediate vertex.

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 $D^1, D^2, \ldots, D^n$.

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

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

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

$D^1(1,2) = \min\{D^0(1,2), D^0(1,1)+ D^0(1,2)\}$
$= \min\{4, 0+ 4\} = 4$
$D^1(1,3) = \min\{D^0(1,3), D^0(1,1)+ D^0(1,3)\}$
$= \min\{11, 0+ 11\} = 11$
$D^1(2,1) = \min\{D^0(2,1), D^0(2,1)+ D^0(1,1)\}$
$= \min\{6, 6+ 0\} = 6$
$D^1(2,3) = \min\{D^0(2,3), D^0(2,1)+ D^0(1,3)\}$
$= \min\{2, 6+ 11\} = 2$
$D^1(3,1) = \min\{D^0(3,1), D^0(3,1)+ D^0(1,1)\}$
$= \min\{3, 3+ 0\} = 3$
$D^1(3,2) = \min\{D^0(3,2), D^0(3,1)+ D^0(1,2)\}$
$= \min\{\infty, 3+ 4\} = 7$

$D^2(1,2) = \min\{D^1(1,2), D^1(1,2)+ D^1(2,2)\}$
$= \min\{4, 4 + 0\} = 4$
$D^2(1,3) = \min\{D^1(1,3), D^1(1,2)+ D^1(2,3)\}$
$= \min\{11, 4+ 2\} = 6$
$D^2(2,1) = \min\{D^1(2,1), D^1(2,2)+ D^1(2,1)\}$
$= \min\{6, 0+ 6\} = 6$
$D^2(2,3) = \min\{D^1(2,3), D^1(2,2)+ D^1(2,3)\}$
$= \min\{2, 0+ 2\} = 2$
$D^2(3,1) = \min\{D^1(3,1), D^1(3,2)+ D^1(2,1)\}$
$= \min\{3, 7+ 6\} = 3$
$D^2(3,2) = \min\{D^1(3,2), D^1(3,2)+ D^1(2,2)\}$
$= \min\{7, 7+ 0\} = 7$

$D^3(1,2) = \min\{D^2(1,2), D^2(1,3)+ D^2(3,2)\}$
$= \min\{4, 6 + 7\} = 4$
$D^3(1,3) = \min\{D^2(1,3), D^2(1,3)+ D^2(3,3)\}$
$= \min\{6, 6+ 0\} = 6$
$D^3(2,1) = \min\{D^2(2,1), D^2(2,3)+ D^2(3,1)\}$
$= \min\{6, 2+ 3\} = 5$
$D^3(2,3) = \min\{D^2(2,3), D^2(2,3)+ D^2(3,3)\}$
$= \min\{2, 2+ 0\} = 2$
$D^3(3,1) = \min\{D^2(3,1), D^2(3,3)+ D^2(3,1)\}$
$= \min\{3, 0+ 3\} = 3$
$D^3(3,2) = \min\{D^2(3,2), D^2(3,3)+ D^2(3,2)\}$
$= \min\{7, 0+ 7\} = 7$

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.
}
```

Analysis:

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

Exercises

1. Write an algorithm for Topological sorting the directed graph.
2. Explore the applications of DFS and BFS, Describe the biconnected component and algorithm for its detection in a graph
3. Give an example graph where bellman ford algorithm returns FALSE, Justify for the falsity of the return value
4. Give an example graph where Dijkstra's algorithm fails to work. Why the found graph does not work, give reason?
5. Maximum Spanning Tree of a weighted Graph G = (V,E) is a subgraph T = (V,E') of G such that T is a tree and the sum of weights of all the edges of E' is maximum among all possible set of edges that would form a spanning tree.
   Modify the prim's algorithm to solve for maximum spanning tree.

Chapter 2:

[Geometric Algorithms]

Computational Geometry

The field of computational geometry deals with the study of geometric problems. In our class we present few geometric problems for e.g. detecting the intersection between line segments, and try to solve them by using known algorithms. In this lecture, we discuss and present algorithms on context of 2-D.

Application Domains

- Computer graphics
- Robotics
- GIS
- CAD/CAM – IC Design, automobile, buildings.
- Molecular Modeling
- Pattern recognition

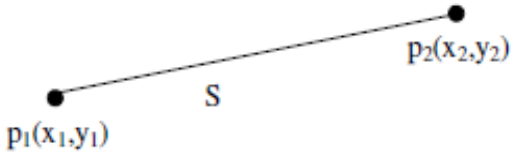
Some Definitions

Point:

A point is a pair of numbers. The numbers are real numbers, but in our usual calculation we concentrate on integers. For e.g. $p_1(x_1,y_1)$ and $p_2(x_2,y_2)$ are two points as shown

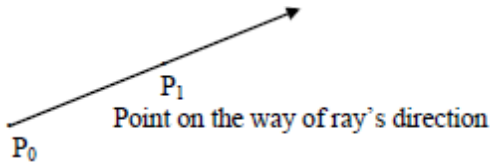
Line segment:

A line segment is a pair of points p_1 and p_2, where two points are end points of the segment. For e.g. $S(p_1,p_2)$ is shown below.



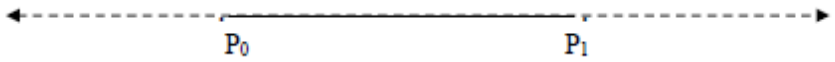
Ray: -

A ray is an infinite one dimensional subset of a line determined by two points: say P_0, P_1, where one point is denoted as the endpoint.
Thus, a ray consists of a bounded point & is extended to infinitely along a line segment.



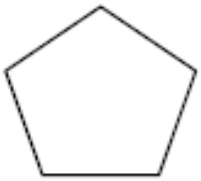
Line: - Line is represented by a pair of points P_0 and P_1 say, which is extended in both way to infinity along the segment represented by the pair of points P_0 & P_1.

Line: - Line is represented by a pair of points P_0 and P_1 say, which is extended in both way to infinity along the segment represented by the pair of points P_0 & P_1.



Polygon:

Simply polygon is a homeomorphic image of a circle, i.e. it is a certain deformation of circle



Simple Polygon:

A simple polygon is a region of plane bounded by a finite collection of line segments to form a simple closed curve. Mathematically, let V_0, V_1, V_2, ---------, V_{n-1} are n ordered vertices in the plane, then the line segments e_0 (V_0, V_1), e_1 (V_1, V_2), ……., e_{n-1} (V_{n-1}, V_0) form a simple polygon if and only if;

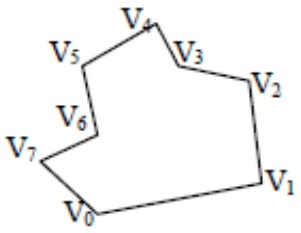
- the intersection of each pair of segments adjacent in cyclic ordering is a simple single point shared by them;

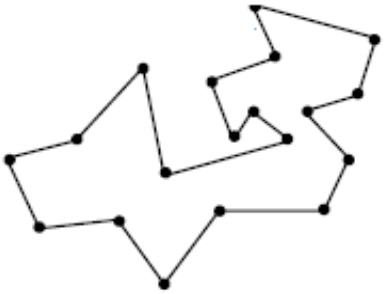
  $e_i \cap e_{i+1} = V_{i+1}$ &

- non-adjacent segments do not intersect;

  $e_i \cap e_j = \Phi$

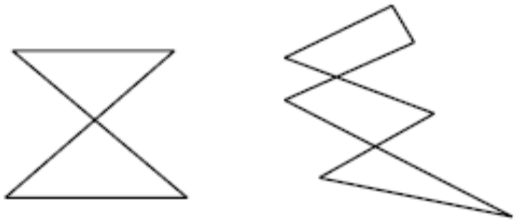
Thus, a polygon is simple if there are no points between non-consecutive linesegments, i.e. vertices are only intersection points. Vertices of simple polygon are assumed to be ordered into counterclockwise direction





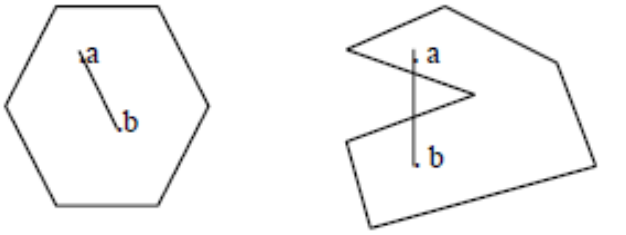
Non-Simple polygon (Self Intersecting)

A polygon is non-simple if there is no single interior region, i.e. non-adjacent edges intersect each other.



Convex Polygon:

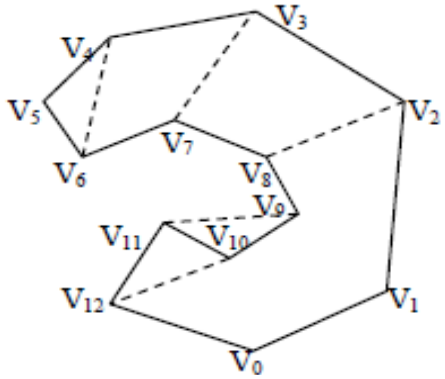
A simple polygon P is convex if and only if for any pair of points x, y in P the line segment between x and y lies entirely in P. We can notice that if all the interior angle is less than 180_0, then the simple polygon is a convex polygon.



Convex Polygon Non Convex (Concave)

Diagonal of a simple polygon:

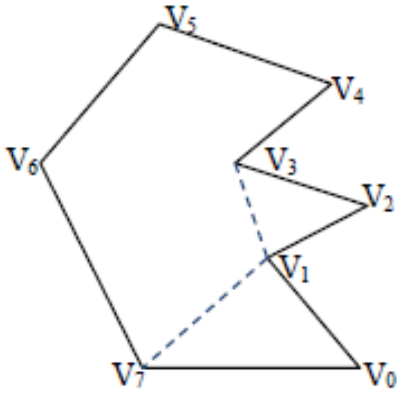
A diagonal of a simple polygon is a line segments connecting two non-adjacent vertices and lies completely inside the polygon.



Here all (V_2, V_8), (V_3, V_7), (V_4, V_6) & (V_{10}, V_{12}) are diagonals of the polygon but (V_9, V_{11}) is not a diagonal

Ear of Polygon:

Three consecutive vertices V_i, V_{i+1}, V_{i+2} of a polygon form an ear if (V_i, V_{i+2}) is a diagonal, V_{i+1} is the tip of the ear.



(V_1, V_2, V_3) is an ear.
But, (V_0, V_1, V_2) is not an ear.
(V_7, V_0, V_1) & (V_1, V_2, V_3) are non-overlapping ears
(V_1, V_2, V_3) & (V_3, V_4, V_5) are non-overlapping ears
(V_3, V_4, V_5) & (V_4, V_5, V_6) are overlapping ears.

Mouth:

Three consecutive vertices V_i, V_{i+1}, V_{i+2} of a polygon form a mouth if (V_i, V_{i+2}) is an external diagonal. In above figure, (V_0, V_1, V_2) & (V_2, V_3, V_4) are mouths of the polygon.

One-Mouth Theorem

Except for convex polygons, every simple polygon has at least one mouth.

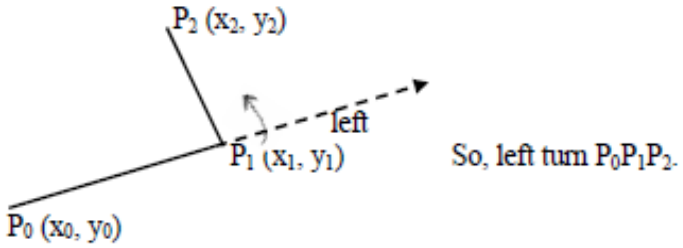
Two-Ears Theorem

Every polygon of $n \geq 4$ vertices has at least two non-overlapping ears.

Notion of Left Turn & Right Turns:

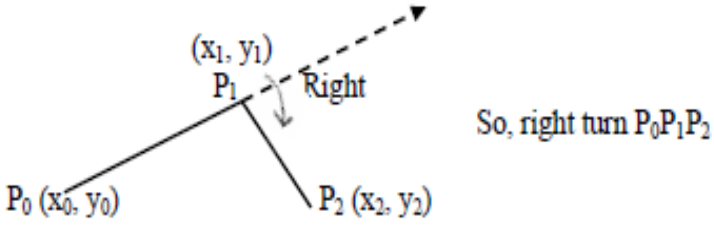
Left Turn:

For three points P_0, P_1, P_2 in a place, P_0, P_1, P_2 is said to be left turn if line segment (P_1, P_2) lies to the left of line segment (P_0, P_1).



Right Turn:

If P line segment (P_1, P_2) lies to the right of (P_0, P_1) then P_0, P_1, P_2 is a right turn.



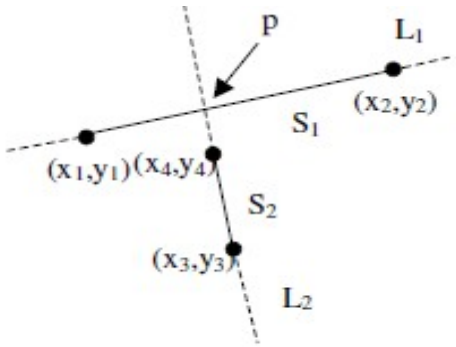
Computing point of intersection between two line segments

We can apply our coordinate geometry method for finding the point of intersection between two line segments. Let S_1 and S_2 be any two line segments. The following steps are used to calculate point of intersection between two line segments. We are not considering parallel line segments here in this discussion.

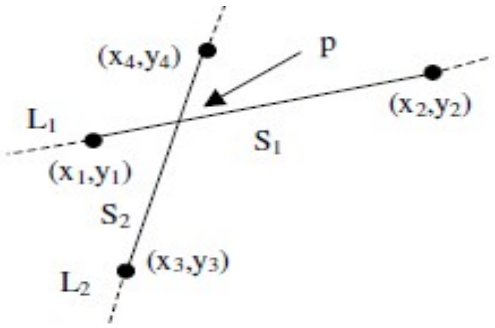
- Determine the equations of line through the line segment S_1 and S_2. Say the equations are $L_1 = (y = m_1x + c_1)$ and $L_2 = (y = m_2x + c_2)$ respectively. We can find the equation of line L_1 using the formula of slope $(m_1) = (y_2-y_1)/ (x_2-x_1)$, where (x_1,y_1) and (x_2,y_2) are two given end points of the line segment S_1. Similarly we can find the m_2 for L_2 also. The values of c_i's can be obtained by using the point of the line segment on the obtained equation after getting slope of the respective lines.

• Solve two equations of lines L_1 and L_2, let the value obtained by solving be $p = (x_i, y_i)$. Here we confront with two cases. The first case is, if p is the intersection of two line segments then p lies on both S1 and S_2. The second case is if p is not an intersection point then p does not lie on at least one of the line segments S1 and S_2.

The figure below shows both the cases.



Segments do not intersect



Segments intersect

Detecting point of intersection

In straightforward manner we can compute the point of intersection (p) between the lines passing through S1 and S_2 and see whether the line segments intersects or not as done in above discussion. However, the above method uses the division in the computation and we know that division is costly process. Here we try to detect the intersection without using division.

Left and Right Turn: Given points $p_0(x_0,y_0)$, $p_1(x_1,y_1)$, and $p_2(x_2,y_2)$. If we try to find whether the path $p_0p_1p_2$ make left or right turn, we check whether the vector $\mathbf{p_0p_1}$ is clockwise or counterclockwise with respect to vector $\mathbf{p_0p_2}$. We compute the cross product of the vectors given by two line segments as

$(p_1- p_0)\times(p_2- p_0) = (x_1- x_0, y_1- y_0) \times(x_2- x_0, y_2- y_0) = (x_1- x_0)(y_2- y_0)-(y_1- y_0)(x_2- x_0)$, this can be represented as

$$\Delta = \begin{vmatrix} 1 & 1 & 1 \\ x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \end{vmatrix}$$

Here we have,

- If $\Delta = 0$ then p_0,p_1,p_2 are collinear
- If $\Delta > 0$ then $p_0p_1p_2$ make left turn i.e. there is left turn at p_1. ($\mathbf{p_0p_1}$ is clockwise with respect to $\mathbf{p_0p_2}$).
- If $\Delta < 0$ then $p_0p_1p_2$ make right turn i.e. there is right turn at p_1. ($\mathbf{p_0p_1}$ is anticlockwise with respect to $\mathbf{p_0p_2}$).

See figure below to have idea on left and right turn as well as direction of points. The cross product's geometric interpretation is also shown below.

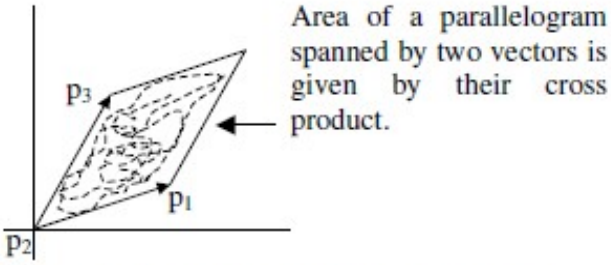


Fig: Cross product geometrical interpretation.

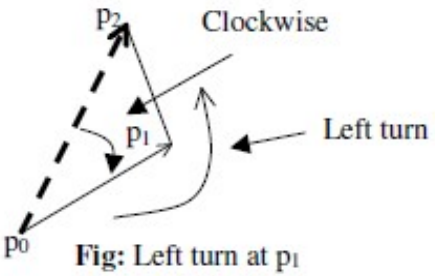


Fig: Left turn at p_1

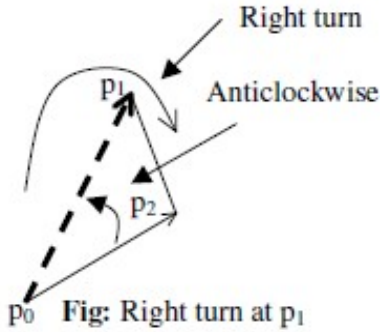


Fig: Right turn at p_1

Using the concept of left and right turn we can detect the intersection between the two line segments in very efficient manner.

Convex hull:

Definition:

- The convex hull of a finite set of points, S in plane is the smallest convex polygon P, that encloses S. (Smallest area)
- The convex hull of a set of points, S in the plane is the union of all the triangles determined by points in S.
  - The convex hull of a finite set of points, S, is the intersection of all the convex polygons (sets) that contain S.
  -

There are wide ranges of application areas where it comes use of convex hulls such as;

- In pattern recognition, an unknown shape may be represented by its convex hull, which is then matched to a database of known shapes.
- In motion planning, if the robot is approximated by its convex hull, then it is easier to plan collision free path on the landscape of obstacles.
  - Smallest box, fitting ranges & so on.

Graham's Scan Algorithm:

This algorithm computes convex hull of points by maintaining the feasible candidate points on the stack. If the candidate point is not extreme, then it is removed from the stack. When all points are examined, only the extreme points remain on the stack & which will result the final hull.

Input $\Rightarrow$ P = $\{p_0, p_1, \text{-------}, p_{n-1}\}$ of n-points.

Output $\Rightarrow$ Convex hull of P

Algorithm:

-Find a point pi with lowest y-coordinate, let it be q0.
- Sort the input points angularly about q0 let the sorted list is now {q0, q1, -------, qn-1}
- Push q0 into stack S and push q1 into the stack S.
- Initialize i = 2

```
while (i < n)
    if LeftTurn (next top (S), top (S), qi) is true
        push qi into stack S.
            i++
    else
        pop the stack
    end if
end while
```

- Each points popped from stack are not vertex of convex hull.
- Finally, at last when all elements are processed, the points that remain on stack are the vertices of the convex hull.

Complexity Analysis:

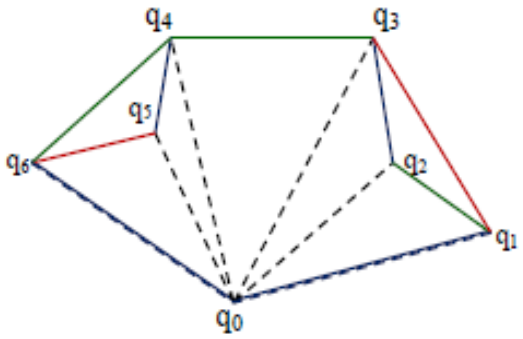
- Finding minimum y-coordinate point it takes O (n) time.
- Sorting angularly about the point takes O (nlogn) time.
- Pushing & popping takes constant time.
- The while loop runs for O (n) times

Hence, the complexity = O (n) + O (nlogn) + O (1) + O (n)
= O (nlogn).

Another way of constructin this Algorithm:

```
GrahamScan(P) // P = {p1, p2,..., pn}
{
    p0 = point with lowest y-coordinate value.
Angularly sort the other points with respect to p0. Let q = {q1,q2,..., qm} be sorted points.
Push(S, p0); // S is a stack
Push(S, q1);
Push(S, q2);
For(i=3;i<m;i++)
{
    a = NexttoTop(S);
    b= Top(S);
    while (a,b,qi makes non left turn)
    Pop(S);
Push(S,qi);
}
return S;
}
```

Consider an example:



(1) At First, Push q_0 & q_1 into stack $\Rightarrow$ | | | | q_1 | q_0 |

(2) **Scan q_2**:

$q_0q_1q_2$ is left turn so push q_2 into stack. $\Rightarrow$ | | | q_2 | q_1 | q_0 |

(3) **Scan q_3:**

$q_1q_2q_3$ is not left turn so pop (stack) i.e. pop q_2 $\Rightarrow$ | | | | q_1 | q_0 |

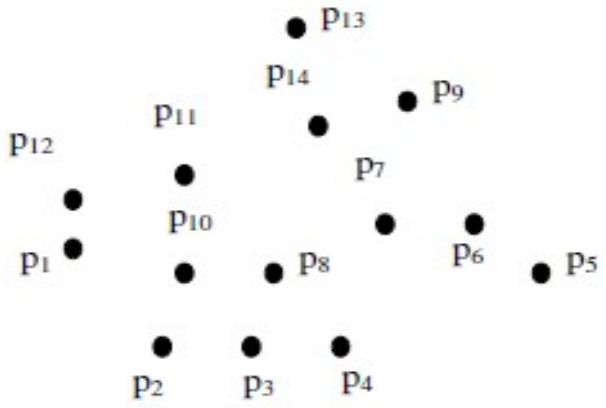
$q_0q_1q_3$ is left turn so push q_3 into stack. $\Rightarrow$ | | | q_3 | q_1 | q_0 |

& so on……

At last, final hull will be $\Rightarrow$ $\{q_0, q_1, q_3, q_4, q_6\}$

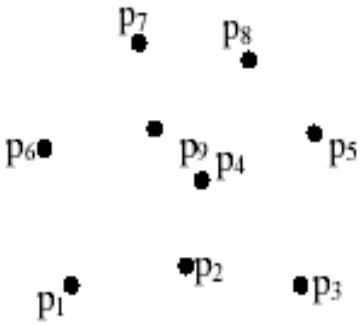
Exercises:

1. Write down the algorithm to find the point of intersection of two linesegments, if exists.
2. Use Graham's scan algorithm to find convex hull of the set of points below.

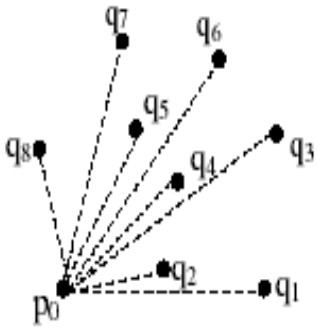


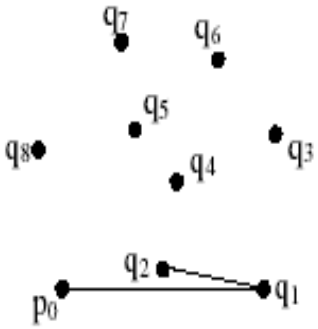
Example2:

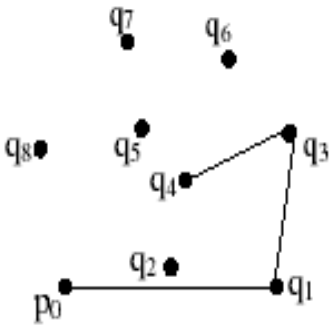
Find the convex hull of the set of points given below using graham's scan algorithm.

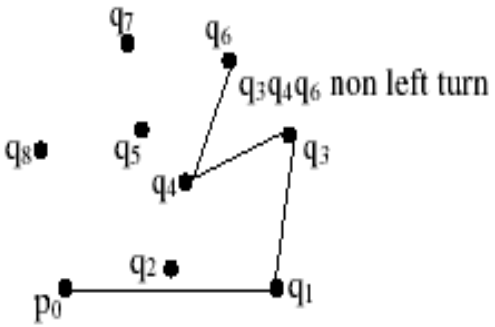


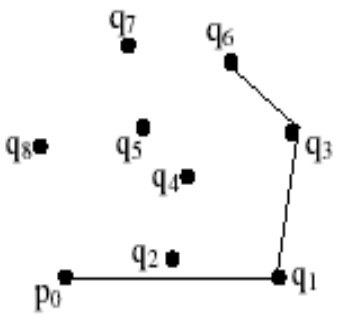
Solution:

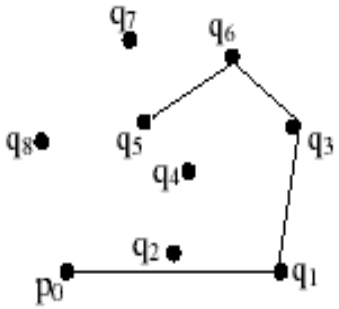


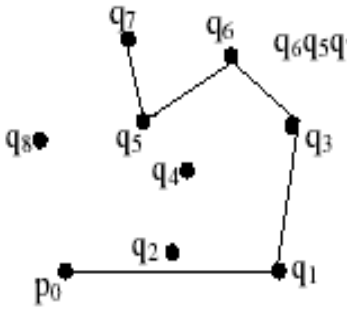




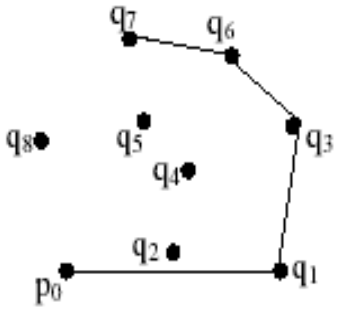


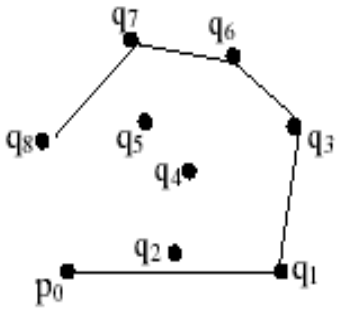


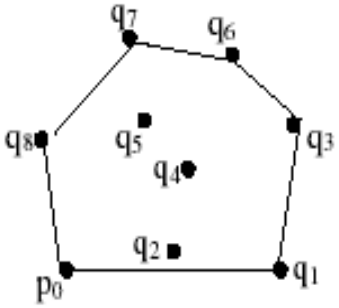




$q_6q_5q_7$ non left turn







Chapter 3
[NP Complete Problems&Approximation Algorithms]

Up to now we were considering on the problems that 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 superpolynomial 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.

P and NP classes and NP completeness:

The set of problems that can be solved using polynomial time algorithm is regarded asclass 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.

Problems:

Abstract Problems:

Abstract problem A is binary relation on set I of problem instances, and the set S of problem solutions. For e.g. Minimum spanning tree of a graph G can be viewed as a pair of the given graph G and MST graph T.

Decision Problems:

Decision problem D is a problem that has an answer as either “true”, “yes”, “1” or “false”, ”no”, “0”. For e.g. if we have the abstract shortest path with instances of the problem and the solution set as {0,1}, then we can transform that abstract problem by reformulating the problem as “Is there a path from u to v with at most k edges”. In this situation the answer is either yes or no.

Optimization Problems:

We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with the best value. This kind of problem is called optimization problem. For e.g. given the graph G, and the vertices u and v find the shortest path from u to v with minimum number of edges. The NP completeness does not directly deal with optimizations problems, however we can translate the optimization problem to the decision problem.

Encoding:

Encoding of a set S is a function e from S to the set of binary strings. With the help of encoding, we define **concrete problem** as a problem with problem instances as the set of binary strings i.e. if we encode the abstract problem, then the resulting encoded problem is concrete problem. So, encoding as a concrete problem assures that every encoded problem can be regarded as a language i.e. subset of {0,1}*.

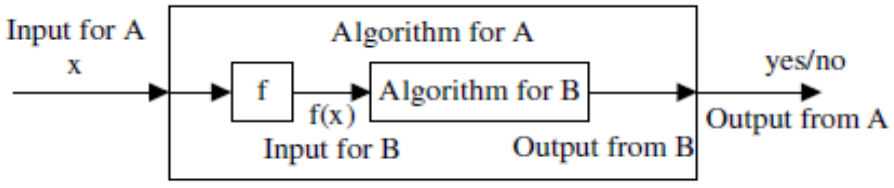
Complexity Class P:

Complexity class **P** is the set of concrete decision problems that are polynomial time solvable by deterministic algorithm. If we have an abstract decision problem A with instance set I mapping the set {0,1}, an encoding e: I→{0,1}* is used to denote the concrete decision problem e(A). We have the solutions to both the abstract problem instance i∈I and concrete problem instance e(i) ∈ {0,1}* as A(i)∈ {0,1}. It is important to understand that the encoding mechanism does greatly vary the running time of the algorithm for e.g. take some algorithm that runs in O(n) time, where the n is size of the input. Say if the input is just a natural number k, then its unary encoding makes the size of the input as **k** bits as k number of 1's and hence the order of the algorithm's running time is O(k). In other situation if we encode the natural number k as binary encoding then we can represent the number k with just **logk** bits (try to represent with 0 and 1only) here the algorithm runs in O(n) time. We can notice that if n = logk then O(k) becomes $O(2^n)$ with unary encoding. However in our discussion we try to discard the encoding like unary such that there is not much difference in complexity.

We define **polynomial time computable** function f:{0,1}*→{0,1}* with respect to some polynomial time algorithm PA such that given any input x ∈ {0,1}*, results in output f(x). For some set I of problem instances two encoding e_1 and e_2 are **polynomially related** if there are two polynomial time computable functions **f** and **g** such that for any i ∈I, both $f(e_1(i)) = e_2(i)$ and $g(e_2(i)) = e_1(i)$ are true i.e. both the encoding should computed from one encoding to another encoding in polynomial time by some algorithm.

Polynomial time reduction:

Given two decision 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.



Complexity Class NP:

NP is the set of decision problems solvable by nondeterministic algorithms in polynomial time. When we have a problem, it is generally much easier to verify that a given value is solution to the problem rather than calculating the solution of the problem. 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. V is the verification algorithm to the decision problem D if V takes input string x as an instance of the problem D and another binary string y, certificate, whose size is no more than the polynomial in the size of x. the algorithm V verifies an input x if there is a certificate y such that answer of D to the input x with certificate y is yes. *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*. We can always say **P** Í **NP**, since if we have the problem for which the polynomial time algorithm exists to solve (decide: *notice the difference between decide and accept*) the problem, then we can always get the verification algorithm that neglects the certificate and accepts the output of the polynomial time algorithm. From the above fact we are clear that P Í NP but the question, whether P = NP remains unsolved and is still the big question in theoretical computer science. Most of the computer scientists, however, believes that P ¹ 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 $\leq_p$ A, for every B ∈ NP.

We call the problem (or language) A satisfying property 2 is called NP-hard.

Cook's Theorem:

"***SAT is NP-hard"***

Proof: (This is not actual proof as given by cook, this is just a sketch)

Take a problem V NP, let A be the algorithm that verifies V in polynomial time (this must be true since V 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 A_x be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of A_x from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for A_x. Conversely, if A_x outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

Theorem 2: (Cook's Theorem)

"***SAT is NP-complete"***

Proof:

To show that SAT is NP-complete we have to show two properties as given by the definition of NP-complete problems. The first property i.e. SAT is in NP we showed above (see pg 5 italicized

part), so it is sufficient to show the second property holds for SAT. The proof for the second property i.e. SAT is NP-hard is from lemma 3. This completes the proof.

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.

An algorithm has an a**pproximate ratio** of ρ(n) if, for any problem of input size n, the cost C of solution by an algorithm and the cost C* of optimal solution have the relation as $\max(C/C^*, C^*, C) \leq \rho(n)$. Such an algorithm is called **ρ(n)-approximation algorithm**.

The relation applies for both maximization ($0 < C \leq C^*$) and minimization ($0 < C^* \leq C$) problems. ρ(n) is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have ρ(n) = 1.

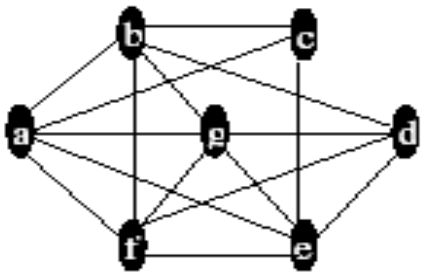
Vertex Cover Problem:

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.

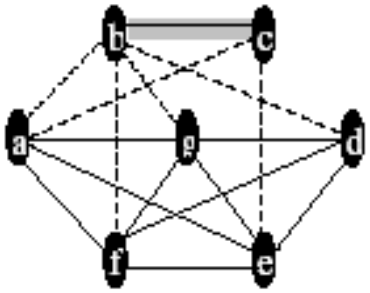
Algorithm:

```
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
}
```

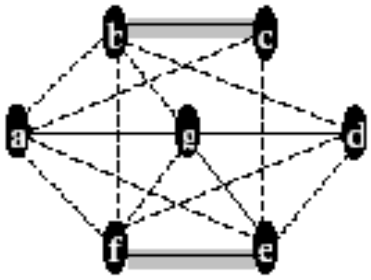
Example: (vertex cover running example for graph below)



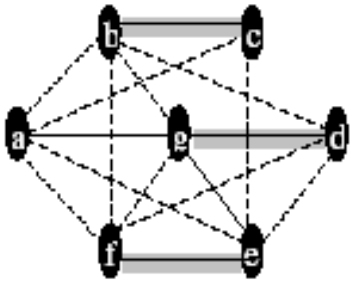
Solution:



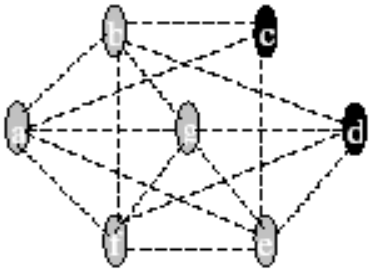
Edge chosen is (b,c) C = {b,c}



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.

Tribhuvan University
Bachelor of Science in Computer Science
And Information Technology
Examination, 2069
New Summit College
(Old Baneshowr , Kathmandu)

Subject: "Design and Analysis of Algorithm(DAA)" **FM:80**
Time: 3 hr. PM: 32
Attempt all the Questions {10 x 8 =80}

1. Why asymptotic notations are important in algorithm analysis? Describe big-O, big-Ω and big-theta notation with suitable examples. {2 + 2 + 2+2}

2. What is recurrence relation? Prove that the complexity of the recurrence relation "$T(n) = 8T(n/2) + n^2$" is $O(n^3)$ by using substitution method. {1+7}

3. Given the following block of code, write a recurrence relation for it and also find asymptotic upper bound (Assume that all dotted code takes constant time) { 4+4)

```
Fun(int n)
{
…….
if(condition1)
        x=Fun(n/2)
else if(condition2)
        x=Fun(2n/3)
else
        x= Fun(n/4)
……
}
```

4. What is the concept behind randomized quick sort? Write down its algorithm and give its average case analysis. {1 + 3 + 4)

5. What is meant by medial order statistics? Write the algorithm for expected liner time selection and analyze it. {1 + 3 + 4}

6. Devise a divide and conquer algorithm for finding minimum and maximum element among a set of given elements. Write recurrence relation for your algorithm and give its big-O estimate. {5+ 3}

7. What are the characteristics of problem that can be solved by using dynamic programming algorithm? Give the recursive definition of solving 0/1 knapsack problem. Trace the algorithm for w={3,4,2,2,3}, v={12,14,6,5,6} and knapsack of capacity 12. (2+1+5)

8. Write the recurrence relation for Longest Common subsequence problem(LCS). Trace the algorithm to find LCS of X={a,b,c,b,d,a,b} and Y={b,d,c,a,b,a}. (2+6)

9. Use master method to find the big-O estimates of the recurrences: {4+4)
a. $T(n) = 3T(n/2) + n$
b. $T(n) = 4T(n/2) + n^2$

10 Show all the steps required for sorting an array of size 10 by using Heap sort.
a[10]={5, 3, 2, 4, 7, 8, 1, 11, 9, 15}. (8)
Hint: At first construct a heap and then sort by using Heap sort properties.

A Complete Note in Design And Analysis of Algorithms

By Bhupendra S. Saud

Email: Saud.bhupendra427@gmail.com

The End