

## Chapter 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.

### Expressing Algorithms

There are many ways of expressing algorithms; the order of ease of expression is natural language, pseudo code and real programming language syntax. In this course I inter mix the natural language and pseudo code convention.

### 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 timethan 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:**  $n^{\text{th}}$  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.  $O(1)$ .

**Example 2: Greatest Common Divisor****Inputs:** Two numbers  $a$  and  $b$ **Output:** G.C.D of  $a$  and  $b$ .**Algorithm:** assume (for simplicity)  $a > b \geq 0$ 

```
gcd(a,b)
{
    While(b != 0)
    {
        d = a/b ;
        temp = b ;
        b = a - b * d ;
        a = temp ;
    }
    return a ;
}
```

**Efficiency**

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

Another way of analyzing:

For Simplicity Let us assume that

$$b = 2^n$$

$$\Rightarrow n = \log b$$

$$\Rightarrow \text{Loop executes } \log b \text{ times in worst case}$$

$$\Rightarrow \text{Time Complexity} = o(\log b)$$

**Space Complexity:** The only allocated spaces are for variables so 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:

$$1. \log_a b = \log_c b / \log_c a ; c > 0$$

$$2. \log ab = \log a + \log b$$

$$3. \log a/b = \log a - \log b$$

$$4. \log (a^b) = b \log a$$

$$5. \log x < x \text{ for all } x > 0$$

$$6. \log 1 = 0, \log 2 = 1, \log 1024 = 10.$$

$$7. {}_a \log b^n = n {}_a \log b^a$$

### Series

$$1. \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

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

$$= a^{n+1} - 1 / a-1 ; \text{ else}$$

3.  $\sum_{i=1}^n i = n(n+1) / 2$
4.  $\sum_{i=0}^n i^2 = n(n+1)(2n+1) / 6$
5.  $\sum_{i=0}^n i^k \approx n^{k+1} / (k+1) ; k \neq -1$
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  $x \geq x_0$ ,  $0 \leq f(x) \leq c \cdot g(x)$

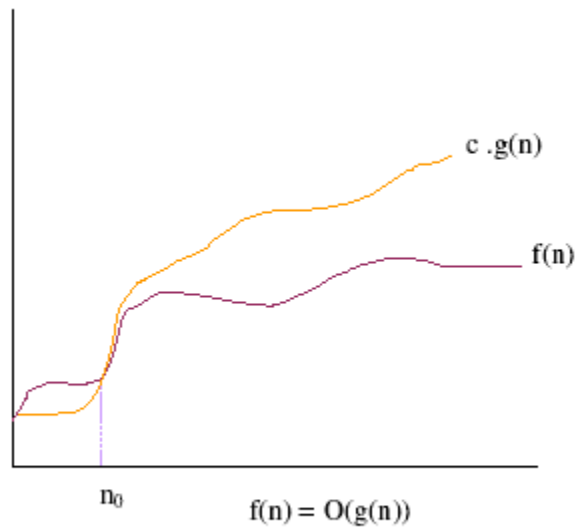
The above relation says that g(x) is an upper bound of f(x)

#### Some properties:

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

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

$O(1)$  is used to denote constants.



For all values of  $n \geq n_0$ , plot shows clearly that  $f(n)$  lies below or on the curve of  $c \cdot g(n)$

### 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

$$f(n) \leq c \cdot g(n), n \geq n_0 \text{ as}$$

$$3n^2 + 4n + 7 \leq 14n^2 \text{ for all } n \geq 1$$

The above inequality is trivially true

$$\text{Hence } f(n) = O(g(n))$$

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

$$n \log(n^3) = O(\sqrt{n^3}).$$

3. Is  $2^{n+1} = O(2^n)$  ?

Is  $2^{2n} = O(2^n)$  ?

### Big Omega ( $\Omega$ ) notation

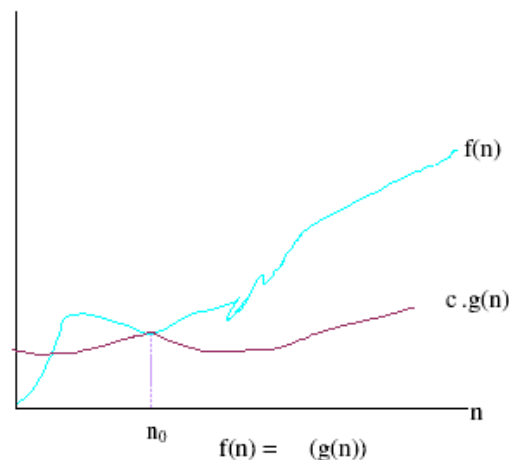
Big omega notation gives asymptotic lower bound. A function  $f(x) = \Omega(g(x))$  (read as  $g(x)$  is big omega of  $f(x)$ ) iff there exists two positive constants  $c$  and  $x_0$  such that for all  $x \geq x_0$ ,  $0 < c \cdot g(x) \leq f(x)$ .

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

#### Some properties:

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

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



For all values of  $n \geq n_0$ , plot shows clearly that  $f(n)$  lies above or on the curve of  $c \cdot g(n)$ .

### 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

$$f(n) \geq c \cdot g(n), n \geq n_0 \text{ as}$$



$$3n^2 + 4n + 7 \geq 1 \cdot n^2 \text{ for all } n \geq 1$$

The above inequality is trivially true

$$\text{Hence } f(n) = \Omega(g(n))$$

### Big Theta ( $\Theta$ ) notation

When we need asymptotically tight bound then we use notation. A function  $f(x) = \Theta(g(x))$  (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  $x \geq x_0$ ,  $c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x)$

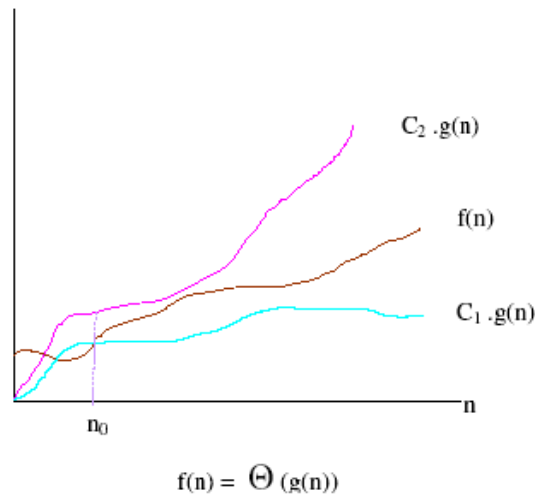
The above relation says that  $f(x)$  is order of  $g(x)$

#### Some properties:

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

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

Symmetry:  $f(x) = \Theta(g(x))$  iff  $g(x) = \Theta(f(x))$



For all values of  $n \geq n_0$ , plot shows clearly that  $f(n)$  lies between  $c_1 \cdot g(n)$  and  $c_2 \cdot g(n)$ .

### Examples

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

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

**Proof:** let us choose  $c_1$ ,  $c_2$  and  $n_0$  values as 14, 1 and 1 respectively then we can have,

$$f(n) \leq c_1 \cdot g(n), n \geq n_0 \text{ as } 3n^2 + 4n + 7 \leq 14 \cdot n^2, \text{ and}$$

$$f(n) \geq c_2 * g(n), n \geq n_0 \text{ as } 3n^2 + 4n + 7 \geq 1 * n^2$$

for all  $n \geq 1$  (in both cases).

So  $c_2 * g(n) \leq f(n) \leq c_1 * g(n)$  is trivial.

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

2. Show  $(n + a)^b = \Theta(n^b)$ , for any real constants  $a$  and  $b$ , where  $b > 0$ .

Here, using Binomial theorem for expanding  $(n + a)^b$ , we get ,

$$C(b,0)n^b + C(b,1)n^{b-1}a + \dots + C(b,b-1)na^{b-1} + C(b,b)a^b$$

We can obtain some constants such that  $(n + a)^b \leq c_1 * (n^b)$ , for all  $n \geq n_0$

And  $(n + a)^b \geq c_2 * (n^b)$ , for all  $n \geq n_0$

Here we may take  $c_1 = 2^b$        $c_2 = 1$        $n_0 = |a|$ ,

Since  $1 * (n^b) \leq (n + a)^b \leq 2^b * (n^b)$ .

Hence the problem is solved.

### Exercises

1. Show that  $2^n$  is  $O(n!)$ .

2.  $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ , where  $a_0, a_2, \dots, a_n$  are real numbers with  $a_n \neq 0$ .

Then show that  $f(x)$  is  $O(x^n)$ ,  $f(x)$  is  $\Omega(x^n)$  and then show  $f(x)$  is order of  $x^n$ .

## 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

$$T(n) = 1 \quad \text{when } n = 1$$

$$T(n) = T(n-1) + O(1) \quad \text{when } n > 1$$

Recursive algorithm for finding Nth Fibonacci number

$$\begin{array}{ll}
 T(1)=1 & \text{when } n=1 \\
 T(2)=1 & \text{when } n=2 \\
 T(n)=T(n-1) + T(n-2) + O(1) & \text{when } n>2
 \end{array}$$

Recursive algorithm for binary search

$$\begin{array}{ll}
 T(1)=1 & \text{when } n=1 \\
 T(n)=T(n/2) + O(1) & \text{when } n>1
 \end{array}$$

### Technicalities

Consider the recurrence

$$\begin{array}{ll}
 T(n) = k & n=1 \\
 T(n) = 2T(n/2) + kn & n>1
 \end{array}$$

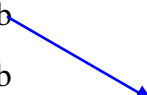
What is Odd about above? In next iteration  $n$  may not be integral.

More accurate is:

$$\begin{array}{ll}
 T(n) = k & n=1 \\
 T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + kn & n>1
 \end{array}$$

This difference rarely matters, so we usually ignore this detail

Again consider the recurrence

$$\begin{array}{ll}
 T(n) = \Theta(1) & n < b \\
 T(n) = a \cdot T(n/b) + f(n) & n \geq b
 \end{array}$$


For constant-sized problem, sizes can bound algorithm by some constant value.

This constant value is irrelevant for asymptote. Thus, we often skip writing the base case equation

## 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.

$$\begin{aligned} T(n) &= 2T(n/2) + 1 && \text{when } n > 1 \\ T(n) &= 1 && \text{when } n = 1 \end{aligned}$$

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 2 \{ 2T(n/4) + 1 \} + 1 \\ &= 4T(n/4) + 2 + 1 \\ &= 4 \{ T(n/8) + 1 \} + 2 + 1 \\ &= 8T(n/8) + 4 + 2 + 1 \\ &\dots\dots\dots \\ &\dots\dots\dots \\ &= 2^k T(n/2^k) + 2^{k-1} T(n/2^{k-1}) + \dots\dots\dots + 4 + 2 + 1. \end{aligned}$$

For simplicity assume:

$$\begin{aligned} n &= 2^k \\ \Rightarrow k &= \log n \\ \Rightarrow T(n) &= 2^k + 2^{k-1} + \dots\dots\dots + 2^2 + 2^1 + 2^0 \\ \Rightarrow T(n) &= (2^{k+1} - 1) / (2 - 1) \\ \Rightarrow T(n) &= 2^{k+1} - 1 \end{aligned}$$

$$\Rightarrow T(n) = 2 \cdot 2^k - 1$$

$$\Rightarrow T(n) = 2n - 1$$

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

**Second Example:**

$$T(n) = T(n/3) + O(n) \quad \text{when } n > 1$$

$$T(n) = 1 \quad \text{when } n = 1$$

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

$$\Rightarrow T(n) = T(n/3^2) + O(n/3) + O(n)$$

$$\Rightarrow T(n) = T(n/3^3) + O(n/3^2) + O(n/3) + O(n)$$

$$\Rightarrow T(n) = T(n/3^4) + O(n/3^3) + O(n/3^2) + O(n/3) + O(n)$$

$$\Rightarrow T(n) = T(n/3^k) + O(n/3^{k-1}) + \dots + O(n/3) + O(n)$$

For Simplicity assume

$$n = 3^k$$

$$\Rightarrow k = \log_3 n$$

$$\Rightarrow T(n) \leq T(1) + c \cdot n/3^{k-1} + \dots + c \cdot n/3^2 + c \cdot n/3 + c \cdot n$$

$$\Rightarrow T(n) \leq 1 + \{ c \cdot n/3^{k-1} + \dots + c \cdot n/3^2 + c \cdot n/3 + c \cdot n \}$$

$$\Rightarrow T(n) \leq 1 + c \cdot n \{ 1/(1-1/3) \}$$

$$\Rightarrow T(n) \leq 1 + 3/2 \cdot c \cdot n$$

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

**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:

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

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

$$\text{Guess: } T(n) = O(n^3).$$

More specifically:

$$T(n) \leq cn^3, \text{ for all large enough } n.$$

Prove by strong induction on  $n$ .

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

Show:  $T(n) \leq cn^3$  for  $\forall n > n_0$ .

Base case,

For  $n=1$ :

$$T(n) = 1$$

Definition

$$1 \leq c$$

Choose large enough  $c$  for conclusion

Inductive case,  $n > 1$ :

$$T(n) = 4T(n/2) + n \quad \text{Definition.}$$

$$\leq 4c \cdot (n/2)^3 + n \quad \text{Induction.}$$

$$= c/2 \cdot n^3 + n \quad \text{Algebra.}$$

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

Need to show  $c/2 \cdot n^3 + n \leq c \cdot n^3$ .

Fortunately, the **constant factor** is shrinking, not growing.

$$T(n) \leq c/2 \cdot n^3 + n$$

From before.

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

Algebra.

$$\leq cn^3$$

Since  $n > 0$ , if  $c \geq 2$

Proved:

$$T(n) \leq 2n^3 \text{ for } \forall n > 0$$

$$\text{Thus, } T(n) = O(n^3).$$

## Second Example

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

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

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

Same recurrence, but now try tighter bound.

More specifically:

$$T(n) \leq cn^2 \text{ for } \forall n > n_0.$$

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

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c \cdot (n/2)^2 + n \\ &= cn^2 + n \end{aligned}$$



Not  $\leq cn^2$  !

Problem is that the constant isn't shrinking.

Solution: Use a tighter guess & inductive hypothesis.

Subtract a lower-order term – a common technique.

Guess:

$$T(n) \leq cn^2 - dn \text{ for } \forall n > 0$$

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

Base case,  $n=1$

$T(n) = 1$  Definition.

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

Inductive case,  $n > 1$ :

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

*Ability to guess effectively comes with experience.*

Changing Variables:

Sometimes a little algebraic 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

$$\text{Let } m = \log n \Rightarrow n = 2^m$$

Thus,

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

Again let

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

We can show that

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

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

## Recursion Tree

Jus Simplification of Iteration method:

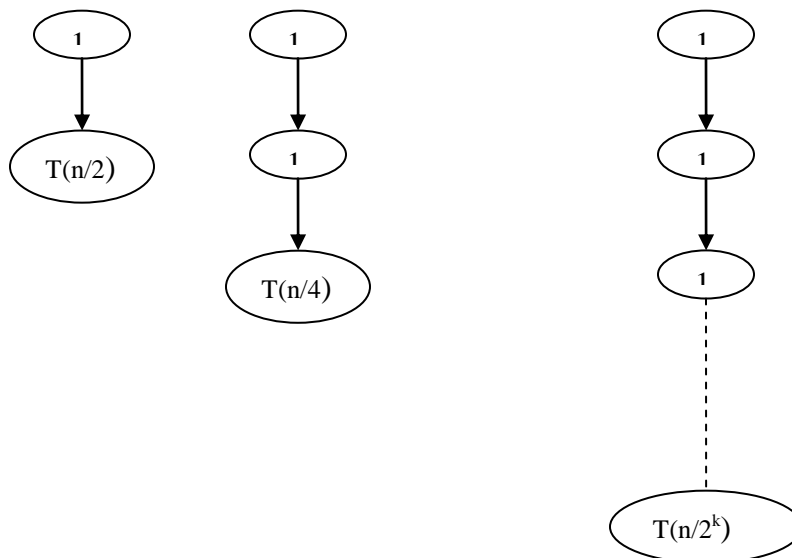
Consider the recurrence

$$T(1) = 1$$

when  $n=1$

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

when  $n > 1$



Cost at each level = 1

For simplicity assume that  $n = 2^k$

$$\Rightarrow k = \log n$$



Summing the cost at each level,

Total cost =  $1 + 1 + 1 + \dots$  Up to  $\log n$  terms

$\Rightarrow$  complexity =  $O(\log n)$

### Second Example

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

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

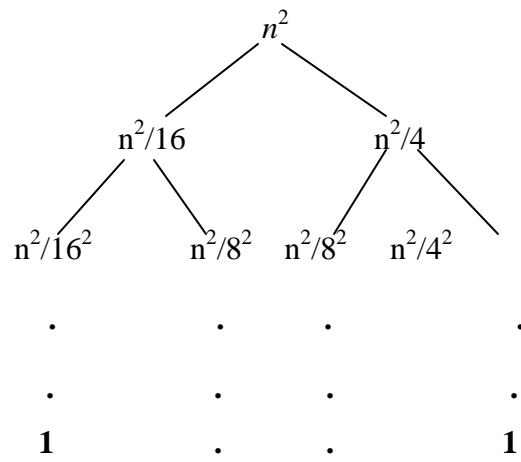
Third Example $T(n)$	Cost at this level
$n$ 	$n$
$2n$	$2n$
$2^2n$	$2^2n$
$\dots$	$\dots$
$1$	$2^k n$

Assume:  $n = 2^k$

$\Rightarrow k = \log n$

$$\begin{aligned}
 T(n) &= n + 2n + 4n + \dots + 2^{k-1}n + 2^k n \\
 &= n(1 + 2 + 4 + \dots + 2^{k-1} + 2^k) \\
 &= n(2^{k+1} - 1)/(2 - 1) \\
 &= n(2^{k+1} - 1) \\
 &\leq n 2^{k+1} \\
 &= 2n 2^k \\
 &= 2n \cdot n \\
 &= O(n^2)
 \end{aligned}$$

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



$$\text{Total Cost} \leq n^2 + 5 n^2/16 + 5^2 n^2/16^2 + 5^3 n^2/16^3 + \dots + 5^k n^2/16^k$$

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

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

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

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

$$= O(n^2)$$

## 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

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

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

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

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

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

$$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8})$$

## Master Method Case 2

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

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

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

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

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

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

## Master Method Case 3

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

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

I.e., is the constant factor shrinking?

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

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

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

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

## Master Method Case 4

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

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

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

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

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

No.

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

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

## 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  $T(n) = 2T(n/2) + n$  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  $a \geq 1$  is a constant.
- Use the master method to give tight asymptotic bounds for the following recurrences.
  - ✓  $T(n) = 9T(n/3) + n$
  - ✓  $T(n) = 3T(n/4) + n \log n$

- ✓  $T(n) = 2T(2n/3) + 1$
- ✓  $T(n) = 2T(n/2) + n \log n$
- ✓  $T(n) = 2T(n/4) + \sqrt{n}$
- ✓  $T(n) = T(\sqrt{n}) + 1$

- 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  $T'(n) = aT'(n/4) + n^2$ . What is the largest integer value for 'a' such that A' is asymptotically faster than A?

## Chapter 2

# Review of 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 where  $d$  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  $O(\log n)$  if binary search is used. To use the binary search array must be sorted.
- ✓ Inefficient use of space

### Singly linked representation (unordered)

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

**Doubly linked representation**

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

**Some Operation with List**

- `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**

➤ boolean isEmpty ();

Return true if and only if this queue is empty. Complexity is O(1).

➤ int size ();

Return this queue's length. Complexity is O(n).

➤ int getFirst ();

Return the element at the front of this queue. Complexity is O(1).

➤ void clear ();

Make this queue empty. Complexity is O(1).

➤ void insert (int elem);

Add elem as the rear element of this queue. Complexity is O(1).

➤ 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

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

**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  $\log n < h+1 \leq n$ . 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();`

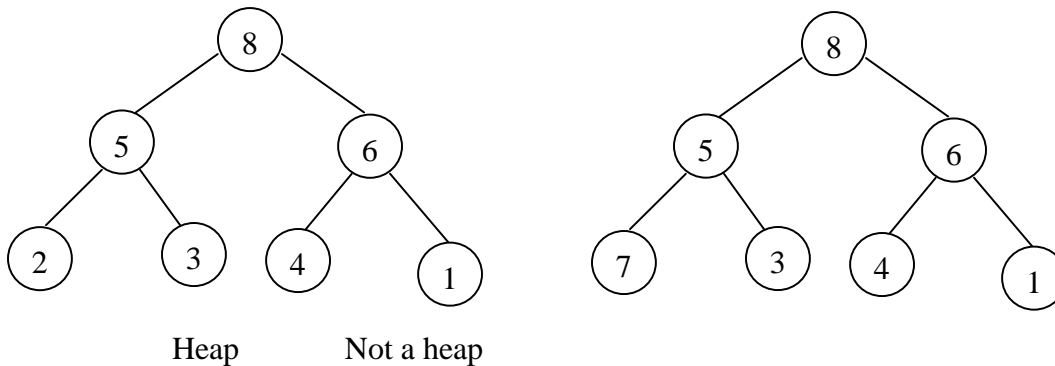
Remove and return the least element from this priority queue. (If there are several least elements, remove the same element that would be returned by `getLeast`.)

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 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	$O(1)$

## 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();

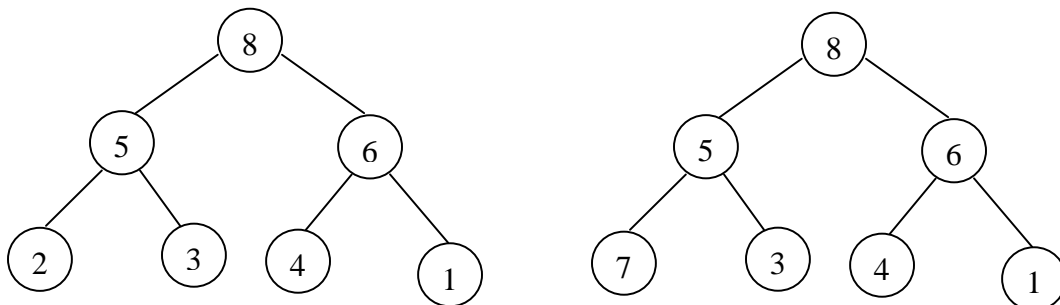
Remove and return the least element from this priority queue. (If there are several least elements, remove the same element that would be returned by getLeast.

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

## Implementation

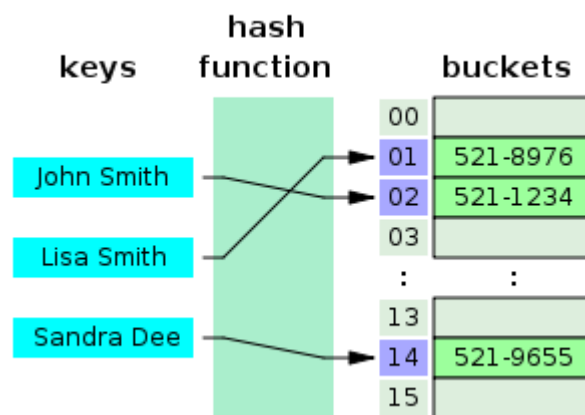
Heaps are implemented by using arrays. Insertion and deletion of an element takes  $O(\log n)$  time. More on this later

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

## Hash Table

A hash table or hash map is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number). The hash function is used to transform the key into the index of an array element where the corresponding value is to be sought. Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that *hash collisions*—different keys that map to the same hash value—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average (i.e  $O(1)$ ). In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.



## Chapter 3

### Divide and Conquer Algorithms

#### 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.

#### Bubble Sort

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

**Algorithm**

BubbleSort(A, n)

```
{
    for(i = 0; i < n-1; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

**Time Complexity:**

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

$$\begin{aligned}\text{Time complexity} &= (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ &= O(n^2)\end{aligned}$$

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

**Space Complexity:**

Since no extra space besides 3 variables is needed for sorting

$$\text{Space complexity} = O(n)$$

**Selection Sort**

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



**Algorithm:**

```

SelectionSort(A)
{
    for( i = 0; i < n ; i++)
    {
        least=A[i]; p=i;
        {
            for ( j = i + 1; j < n ; j++)
                if (A[j] < A[i])
                    least= A[j]; p=j;
        }
    }
    swap(A[i],A[p]);
}

```

**Time Complexity:**

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

$$\begin{aligned}
 \text{Time complexity} &= (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\
 &= O(n^2)
 \end{aligned}$$

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

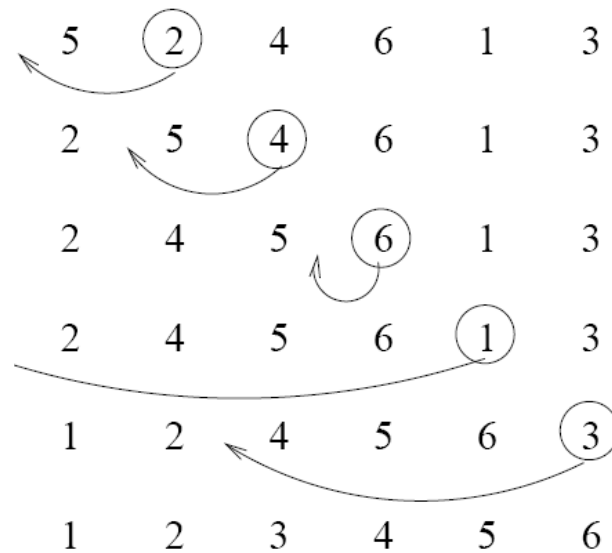
**Space Complexity:**

Since no extra space besides 5 variables is needed for sorting

$$\text{Space complexity} = O(n)$$

**Insertion Sort**

Idea: like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted

**Algorithm:**

InsertionSort(A)

```

{
    for (i=1; i<n; i++)
    {
        key = A[ i]
        for(j=i; j>0 && A[j] >key; j--)
        {
            A[j + 1] = A[j]
        }
        A[j + 1] = key
    }
}

```

**Time Complexity:****Worst Case Analysis:**

Array elements are in reverse sorted order

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

$$\begin{aligned} \text{Time complexity} &= 1 + 2 + 3 + \dots + (n-2) + (n-1) \\ &= O(n^2) \end{aligned}$$

**Best case Analysis:**

Array elements are already sorted

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

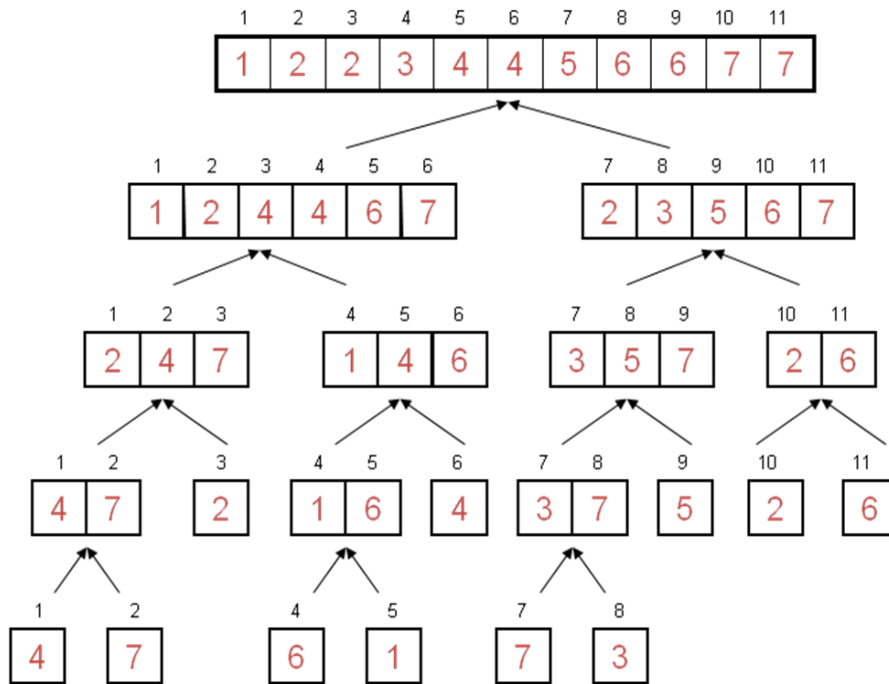
$$\begin{aligned} \text{Time complexity} &= 1 + 2 + 3 + \dots + 1 + 1 \\ &= O(n) \end{aligned}$$

Since no extra space besides 5 variables is needed for sorting

## Merge Sort

- 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

The diagram shows the recursive splitting of an array of 11 elements into individual elements. The array is split into two halves: [4, 7, 2, 6, 1] and [4, 7, 3, 5, 2, 6]. The left half is further split into [4, 7, 2] and [6, 1, 4], and so on, until all elements are isolated in single-element boxes.

**Merging:**

MergeSort(A, l, r)

```
{
    If ( l < r)
    {
        //Check for base case
        m =  $\lfloor (l + r) / 2 \rfloor$  //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:

$$T(n) = 1 \quad \text{if } n=1$$

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

Solving this recurrence we get

$$\text{Time Complexity} = O(n \log n)$$

**Space Complexity:**

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

$$\text{Space Complexity} = 2n + c = O(n)$$

**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:

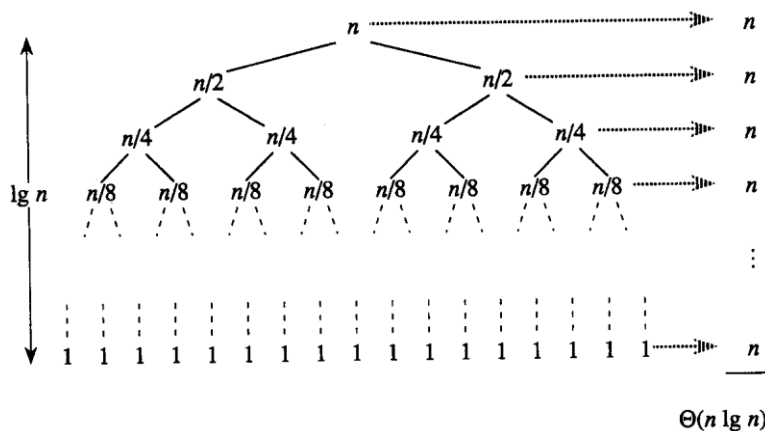
$$T(n) = T(k) + T(n-k-1) + O(n)$$

**Best Case:**

Divides the array into two partitions of equal size, therefore

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

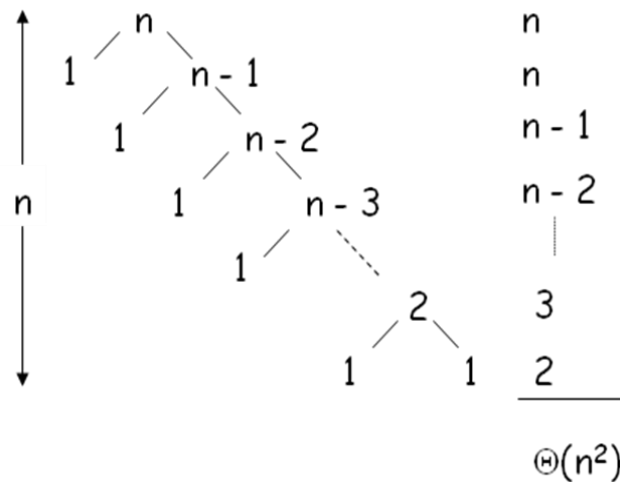
$$\Rightarrow \text{Time Complexity} = O(n \log n)$$



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

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

$\Rightarrow$  Time Complexity =  $O(n^2)$

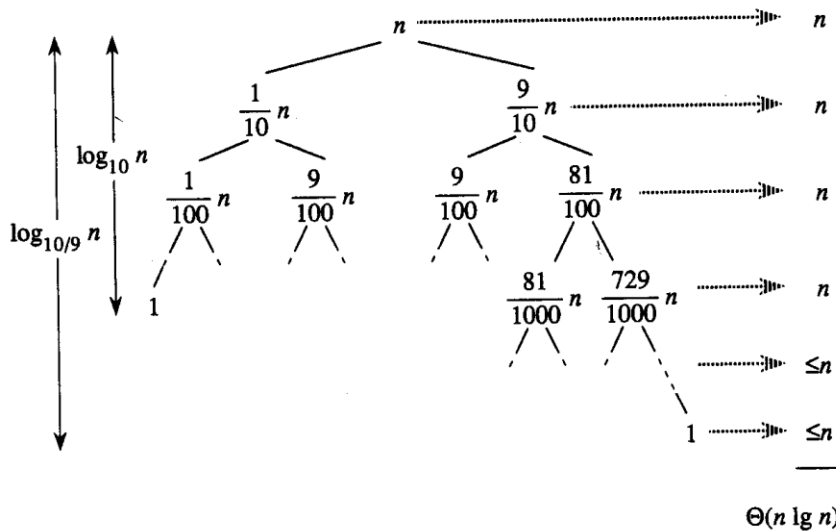


**Case between worst and best:**

9-to-1 partitions split

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

Time Complexity =  $O(n \lg n)$





**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, ....

$$B(n) = 2UB(n/2) + \Theta(n) \text{ Balanced}$$

$$UB(n) = B(n-1) + \Theta(n) \text{ Unbalanced}$$

Solving:

$$\begin{aligned} B(n) &= 2(B(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\ &= 2B(n/2 - 1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

**Randomized Quick Sort:**

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

**Algorithm:**

```

RandQuickSort(A,l,r)
{
    if(l<r)
    {
        m = RandPartition(A,l,r);
        RandQuickSort(A,l,m-1);
        RandQuickSort(A,m+1,r);
    }
}

```

```

RandPartition(A,l,r)
{
    k = random(l,r); //generates random number between i and j including both.
    swap(A[l],A[k]);
    return Partition(A,l,r);
}

```

**Time Complexity:**

Worst Case:

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

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

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

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

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

Proof of induction goal:

$$\begin{aligned}
 T(n) &\leq \max_{1 \leq q \leq n-1} (cq^2 + c(n-q)^2) + \Theta(n) \\
 &= c \cdot \max_{1 \leq q \leq n-1} (q^2 + (n-q)^2) + \Theta(n)
 \end{aligned}$$

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

$$\begin{aligned}
 \max_{1 \leq q \leq n-1} (q^2 + (n-q)^2) &= 1^2 + (n-1)^2 = n^2 - 2(n-1) \\
 T(n) &\leq cn^2 - 2c(n-1) + \Theta(n) \\
 &\leq cn^2
 \end{aligned}$$

**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

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

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

For some  $k = 1, 2, \dots, n-1$ ,  $T(k)$  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

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

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

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

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

$$\text{Let } A_n = T(n)/(n+1)$$

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

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

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

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

$$\Rightarrow A_n \approx 2 \log n$$

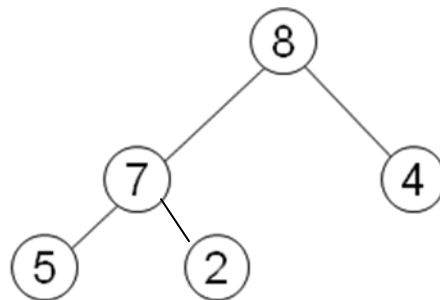
$$\text{Since } A_n = T(n)/(n+1)$$

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

## 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  $x$ ,  $\text{Parent}(x) \geq x$

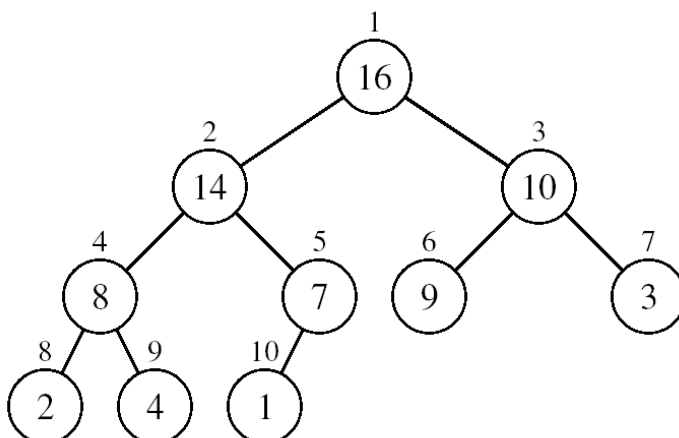
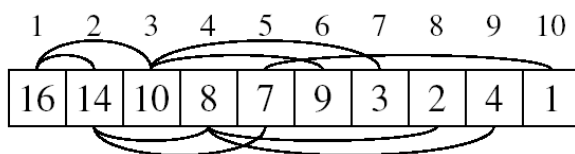


### Array Representation of Heaps

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  $A[i] = A[\lfloor i/2 \rfloor]$
- $\text{Heapsize}[A] \leq \text{length}[A]$

The elements in the subarray  $A[(\lfloor n/2 \rfloor + 1) .. n]$  are leaves



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

- for all nodes  $i$ , excluding the root:

$$A[\text{PARENT}(i)] \geq A[i]$$

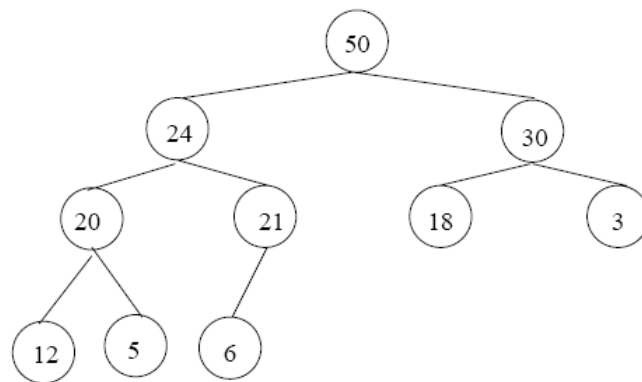
**Min-heaps** (smallest element at root), have the min-heap property:

- for all nodes  $i$ , excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$

### Adding/Deleting Nodes

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



### Operations on Heaps

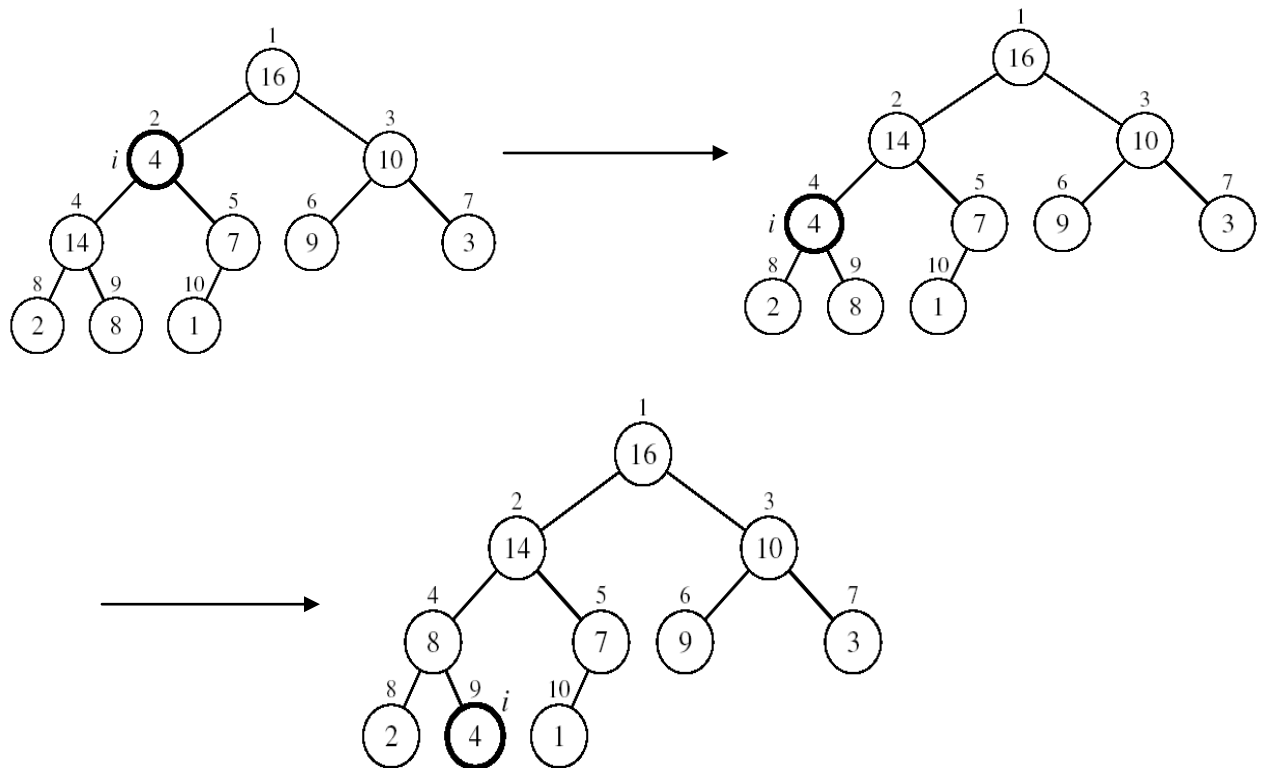
- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

### Maintaining the Heap Property

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

- Exchange with larger child

- Move down the tree
- Continue until node is not smaller than children

**Algorithm:**

Max-Heapify(A, i, n)

{

l = Left(i)

r = Right(i)

largest = i;

**if** l ≤ n and A[l] > A[largest]

largest = l

**if** r ≤ n and A[r] > A[largest]

largest = r

**if** largest ≠ i

exchange (A[i] , A[largest])

Max-Heapify(A, largest, n)

}

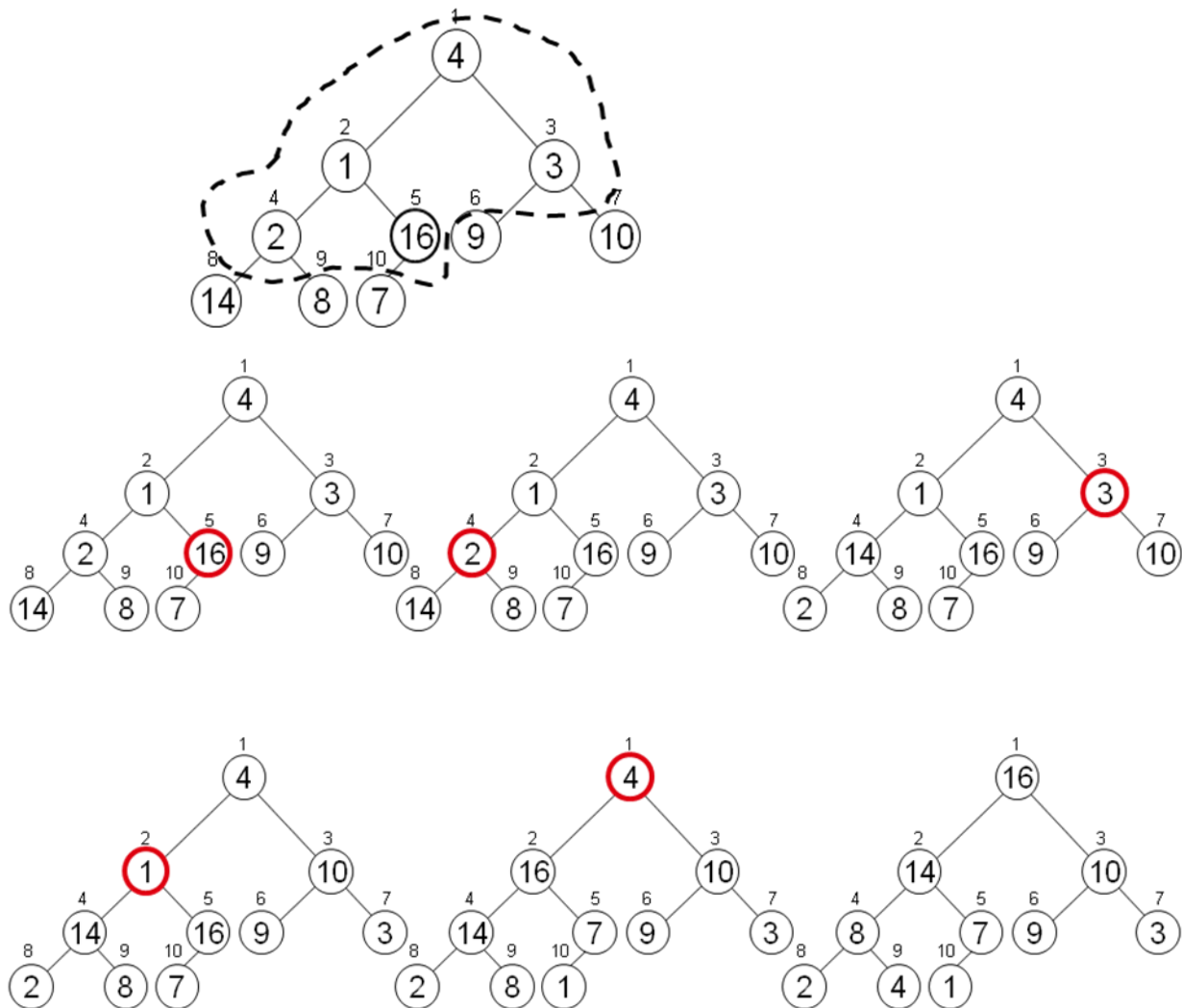
**Analysis:**

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

Time complexity =  $O(h) = O(\log n)$

**Building a Heap**

Convert an array  $A[1 \dots n]$  into a max-heap ( $n = \text{length}[A]$ ). The elements in the subarray  $A[(\lfloor n/2 \rfloor + 1) \dots n]$  are leaves. Apply MAX-HEAPIFY on elements between 1 and  $\lfloor n/2 \rfloor$ .



**Algorithm:**

Build-Max-Heap(A)

n = length[A]

**for** i ←  $\lfloor n/2 \rfloor$  **downto** 1

**do** MAX-HEAPIFY(A, i, n)

**Time Complexity:**

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

This is not an asymptotically tight upper bound

Heapify takes  $O(h)$

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

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

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

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

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

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

Let  $k = h-i$

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

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

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

Differentiating both sides we get,

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



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

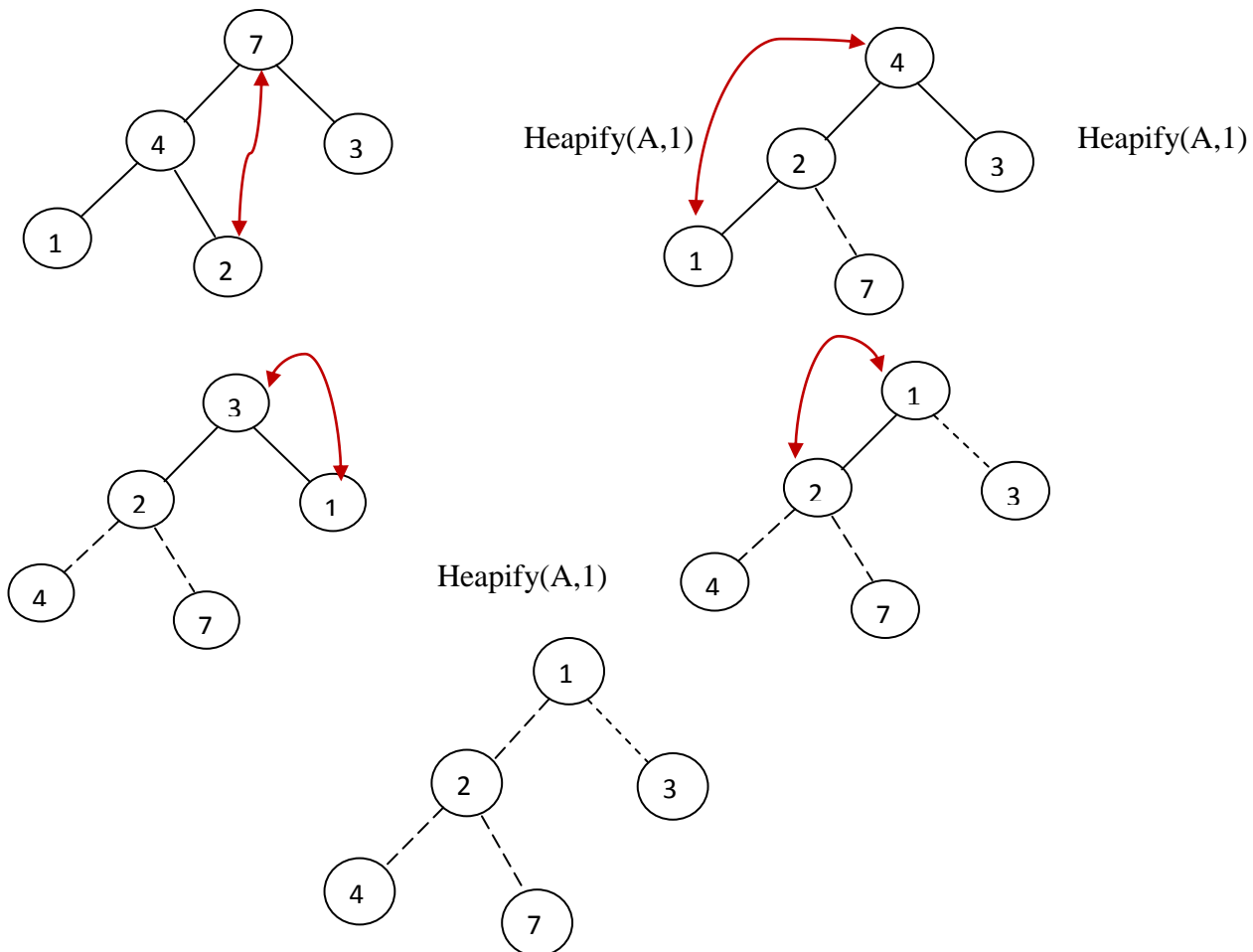
Put  $x=1/2$

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

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

## 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



**Algorithm:**

HeapSort(A)

```

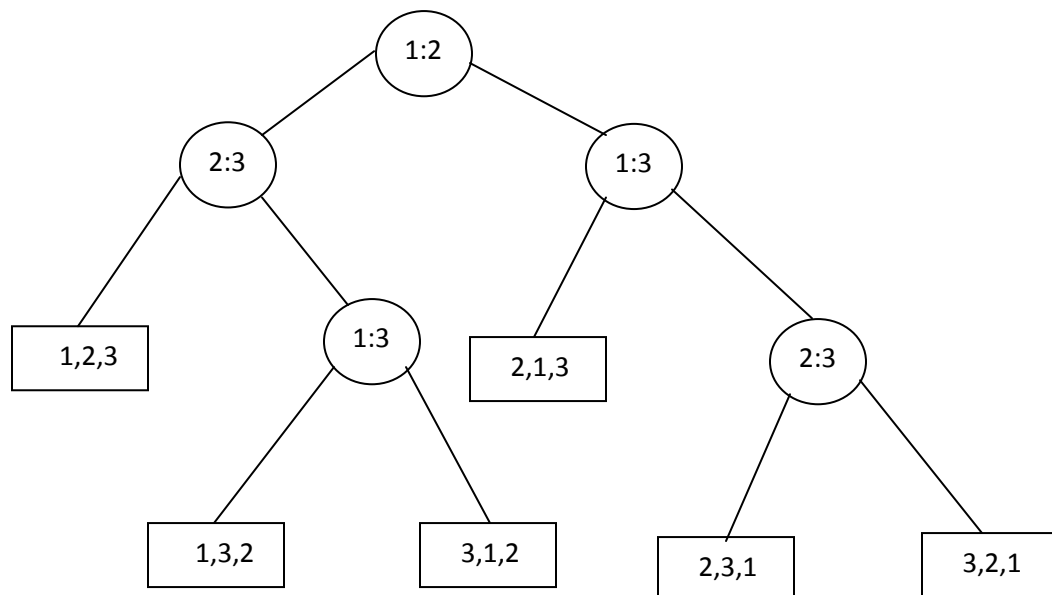
{
    BuildHeap(A); //into max heap
    n = length[A];
    for(i = n ; i >= 2; i--)
    {
        swap(A[1],A[n]);
        n = n-1;
        Heapify(A,1);
    }
}

```

**Lower Bound for Sorting**

All the sorting algorithms we have seen so far are *comparison sorts*: only use comparisons to determine the relative order of elements. The best worst-case running time that we've seen for comparison sorting is  $O(n \lg n)$ . E.g., insertion sort, merge sort, quicksort, heapsort.

Sort  $\langle a_1, a_2, \dots, a_n \rangle$



Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ . The left subtree shows subsequent comparisons if  $a_i \leq a_j$ . The right subtree shows subsequent comparisons if  $a_i \geq a_j$ . A decision tree

can model the execution of any comparison sort: The tree contains the comparisons along all possible instruction traces. The running time of the algorithm is the length of the path taken. Worst-case running time is height of tree.

The tree must contain  $\geq n!$  leaves, since there are  $n!$  possible permutations. A height- $h$  binary tree has  $\leq 2^h$  leaves.

$$\text{Thus, } n! \leq 2^h$$

$$\therefore h \geq \lg(n!)$$

$$\text{Since, } \lg(n!) = \Omega(n \lg n)$$

$$\Rightarrow h = \Omega(n \lg n).$$

## 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 file containing keys  $z[0; 1; \dots; n-1]$  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,  $z[0; \dots; n/2 - 1]$ , or on the second half,  $z[n/2; \dots; n-1]$ .

**Algorithm**

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

    else
    {
        m = (l + r) /2 ; //integer division
        if(key == A[m])
            return m+1;
        else if (key < A[m])
            return BinarySearch(l, m-1, key) ;
        else return BinarySearch(m+1, r, key) ;
    }
}
```

**Analysis:**

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

$$\begin{aligned} T(n) &= T(n/2) + Q(1) \\ &= O(\log n) . \end{aligned}$$

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  $O(\log n)$  time. In the average case also running time is  $O(\log n)$ . For unsuccessful search best, worst and average time complexity is  $O(\log n)$ .

## Max and Min Finding

Here our problem is to find the minimum and maximum items in a set of  $n$  elements. We will see two methods here first one is iterative version and the next one uses divide and conquer strategy to solve the problem.

### Iterative Algorithm:

```
MinMax(A,n)
{
    max = min = A[0];
    for(i = 1; i < n; i++)
    {
        if(A[i] > max)
            max = A[i];
        if(A[i] < min)
            min = A[i];
    }
}
```

The above algorithm requires  $2(n-1)$  comparison in worst, best, and average cases. The comparison  $A[i] < \min$  is needed only when  $A[i] > \max$  is not true. If we replace the content inside the for loop by

```
if(A[i] > max)
    max = A[i];
else if(A[i] < min)
    min = A[i];
```

Then the best case occurs when the elements are in increasing order with  $(n-1)$  comparisons and worst case occurs when elements are in decreasing order with  $2(n-1)$  comparisons. For the average case  $A[i] > \max$  is about half of the time so number of comparisons is  $3n/2 - 1$ .

We can clearly conclude that the time complexity is  $O(n)$ .

**Divide and Conquer Algorithm:**

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.

$$T(n) = 2T(n/2) + 1, \text{ if } n > 2$$

$$T(n) = 1, \text{ if } n \leq 2$$

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

**Selection**

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

This problem can be solved in  $\Theta(n \log n)$  in a very straightforward way. First sort the elements in  $\Theta(n \log n)$  time and then pick up the  $i^{\text{th}}$  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  $k^{\text{th}}$  smallest or  $k^{\text{th}}$  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 =  $(n-1) + (n-2) + \dots\dots\dots(n-k-1)$

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

**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  $j \geq \text{ceil}(n/2)$

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 from 1 to  $\text{ceil}(n/2)$  and second time for  $\text{ceil}(n/2)$  to  $n-1$ , so we can write,

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

Using substitution method,

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

To show  $T(n) \leq cn$

Assume  $T(j) \leq cj$

Substituting on the relation

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

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

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

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

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

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

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

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

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

### Selection in worst case linear time

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

#### Algorithm

Divide  $n$  elements into groups of 5

Find median of each group

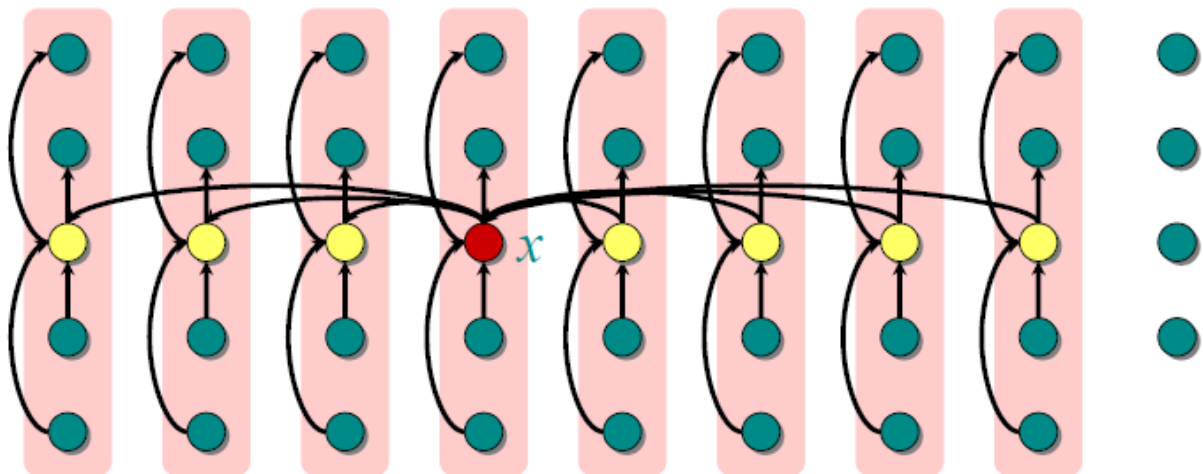
Use Select() recursively to find median  $x$  of the  $\lfloor n/5 \rfloor$  medians

Partition the  $n$  elements around  $x$ . Let  $k = \text{rank}(x)$  //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  $\leq x$ , 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  $\leq x$ .

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

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

Therefore, for  $n \geq 50$  the recursive call to SELECT in Step 4 is executed recursively on  $\leq 3n/4$  elements 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”

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

$$\text{Guess } T(n) = O(n)$$

$$\text{To Show } T(n) \leq cn$$

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

Now,

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

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

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

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

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

## Chapter 4

### 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 discuss a top-down alternative, called memoization)

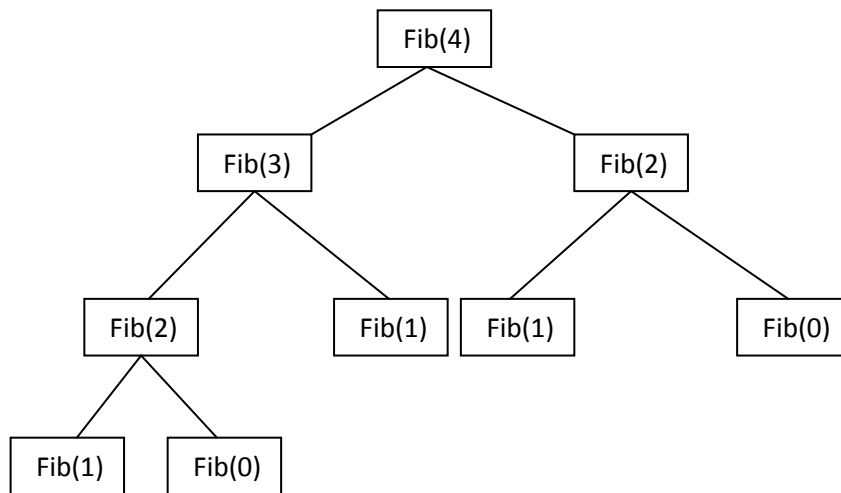
The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the formulation of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to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

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

```

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  $W = 9$ .

$i \backslash w$	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  $p_{i-1} \times p_i$ , determine the order of multiplication that minimizes the number of operations.

**Important Note:** This algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

Although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices:  $A_1$  be  $5 \times 4$ ,  $A_2$  be  $4 \times 6$  and  $A_3$  be  $6 \times 2$ .

$$\text{multCost}[(A_1 A_2) A_3] = (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180$$

$$\text{multCost}[A_1 (A_2 A_3)] = (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

Let  $A_{i...j}$  denote the result of multiplying matrices  $i$  through  $j$ . It is easy to see that  $A_{i...j}$  is a  $p_{i-1} \times p_j$  matrix. So for some  $k$  total cost is sum of cost of computing  $A_{i...k}$ , cost of computing  $A_{k+1...j}$ , and cost of multiplying  $A_{i...k}$  and  $A_{k+1...j}$ .

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

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



Matrix-Chain-Multiplication(p)

```
{
    n=length[p]
    for( i= 1 i<=n i++)
    {
        m[i, i]= 0
    }
    for(l=2; l<= n; l++)
    {
        for( i= 1; i<=n-l+1; i++)
        {
            j = i + l - 1
            m[i, j] = ∞
            for(k= i; k<= j-1; k++)
            {
                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  $O(n^2)$ .

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

**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

$(A_1A_2A_3A_4) \Rightarrow ((A_1A_2A_3)(A_4)) \Rightarrow (((A_1)(A_2A_3))(A_4))$

**Longest Common Subsequence Problem**

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

For example, let  $X = (\text{ABRACADABRA})$  and let  $Z = (\text{AADAA})$ , 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  $X = (\text{ABRACADABRA})$  and let  $Y = (\text{YABBADABBAD})$ . Then the longest common subsequence is  $Z = (\text{ABADABA})$

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences  $X = (x_1; \dots; x_m)$  and  $Y = (y_1, \dots, y_n)$  determine a longest common subsequence.

**DP Formulation for LCS:**

Given a sequence  $X = \langle x_1, x_2, \dots, x_m \rangle$ ,  $X_i = \langle x_1, x_2, \dots, x_i \rangle$  is called  $i^{\text{th}}$  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 contain character  $x_i$  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  $Y_j$ .

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

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])
```

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

```

    {
        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  $X=abbabba$   $Y=aaabba$

Y \ X	$\Phi$	a	A	a	b	b	a
$\Phi$	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 5

### Greedy Algorithms

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

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

- ➡ Greedy Choice Property
- ➡ Optimal Substructure Property

**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  $i^{\text{th}}$  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  $0 \leq x_i \leq 1$ . 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.  $v[1 \dots n]$  and  $w[1 \dots n]$  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,  $x[1 \dots n]$  is the solution vector that includes fractional amount of items and  $n$  is the number of items.

GreedyFracKnapsack( $W, n$ )

{

for( $i=1; i \leq n; i++$ )

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

```

        x[i] = 0.0;
        tw = W;
        for(i=1; i<=n; i++)
        {
            if(w[i] > tw)
                break;
            else
                x[i] = 1.0;

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

```

**Analysis:**

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

**Job Sequencing with Deadline**

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

**Example**

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

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

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

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

Greedy strategy using *total profit* as optimization function to above example.

Begin with  $J=\phi$

- Job 1 considered, and added to  $J \rightarrow J=\{1\}$
- Job 4 considered, and added to  $J \rightarrow J=\{1,4\}$
- Job 3 considered, but discarded because not feasible  $\rightarrow J=\{1,4\}$
- Job 2 considered, but discarded because not feasible  $\rightarrow J=\{1,4\}$

Final solution is  $J=\{1,4\}$  with total profit 127 and it is optimal

### Algorithm

Assume the jobs are ordered such that  $p[1] \geq p[2] \geq \dots \geq p[n]$

$d[i] \geq 1$ ,  $1 \leq i \leq n$  are the deadlines,  $n \geq 1$ . The jobs  $n$  are ordered such that  $p[1] \geq p[2] \geq \dots \geq p[n]$ .  $J[i]$  is the  $i$ th job in the optimal solution,  $1 \leq i \leq k$ .

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

```
{
    for(i=1;i<=n;i++)
        {
            //initially no jobs are selected
            J[i]=0;
```

Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU

```
    }  
    for (int i=1; i<=n; i++)  
    {  
        d=d[i];  
        for(k=d;k>=0;k--)  
        {  
            if(j[k]==0)  
            {  
                J[k]=i;  
            }  
        }  
    }  
}
```

### Analysis

First for loop executes for  $O(n)$  times .

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

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

## Huffman Coding

Huffman coding is an algorithm for the lossless compression of files based on the frequency of occurrence of a symbol in the file that is being compressed. In any file, certain characters are used more than others. Using binary representation, the number of bits required to represent each character depends upon the number of characters that have to be represented. Using one bit we can represent two characters, i.e., 0 represents the first character and 1 represents the second character. Using two bits we can represent four characters, and so on. Unlike ASCII code, which is a fixed-length code using seven bits per character, Huffman compression is a variable-length coding system that assigns smaller codes for more frequently used characters and larger codes for less frequently used characters in order to reduce the size of files being compressed and transferred.



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

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

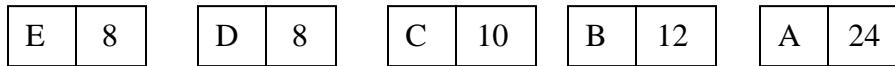
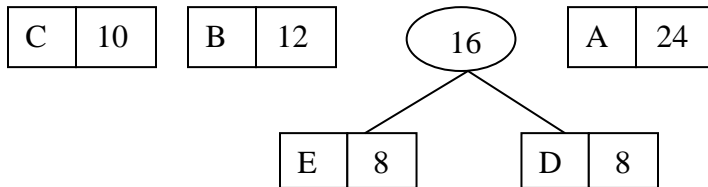
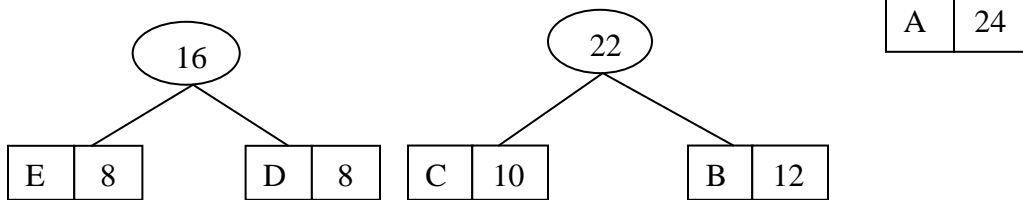
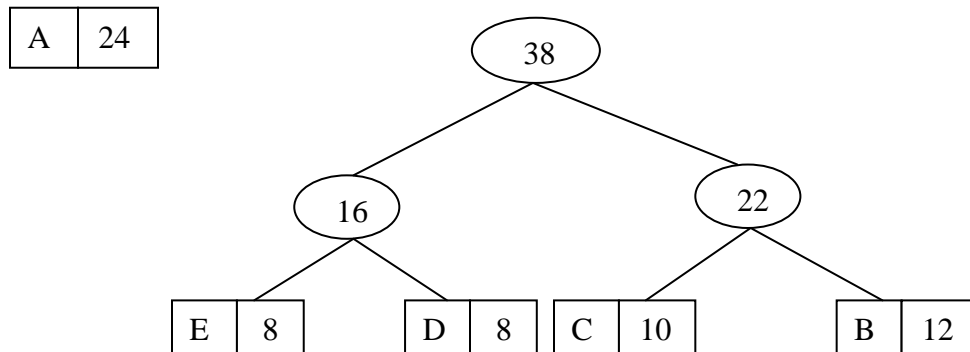
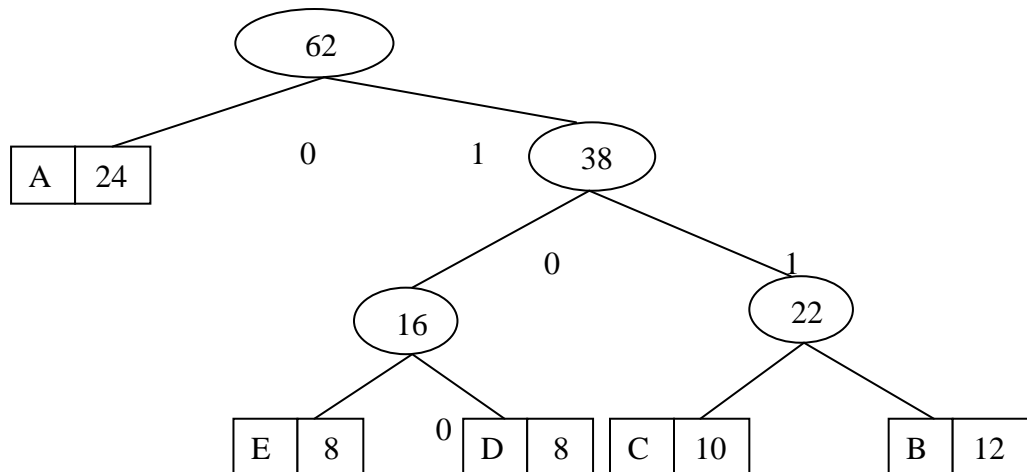
### Example

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

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

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

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

**Step1:****Step2:****Step3:****Step4:****Step5:**

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

-----  
Total length of message: 138 bit

### Algorithm

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

HuffmanAlgo( $C$ )

```
{
    n = |C|;  Q = C;
    For(i=1; i<=n-1; i++)
    {
        z = Allocate-Node();
        x = Extract-Min(Q);
        y = Extract-Min(Q);
        left(z) = x;  right(z) = y;
        f(z) = f(x) + f(y);
        Insert(Q,z);
    }
}
```

### Analysis

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

## Chapter 6

### 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.

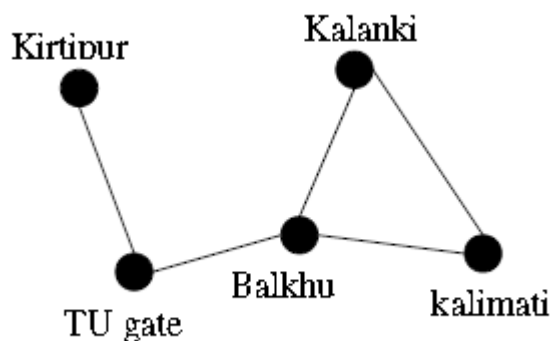
We say that vertex  $v$  is adjacent to vertex  $u$  if there is an edge  $(u; v)$ . In a directed graph, given the edge  $e = (u; v)$ , we say that  $u$  is the origin of  $e$  and  $v$  is the destination of  $e$ . In undirected graphs  $u$  and  $v$  are the endpoints of the edge. The edge  $e$  is 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  $E = \Theta(V)$ , 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.

## Graph Representation

Graph is a pair  $G = (V, E)$  where  $V$  denotes a set of vertices and  $E$  denotes the set of edges connecting two vertices. Many natural problems can be explained using graph for example modeling road network, electronic circuits, etc. The example below shows the road network.



### Representing Graphs

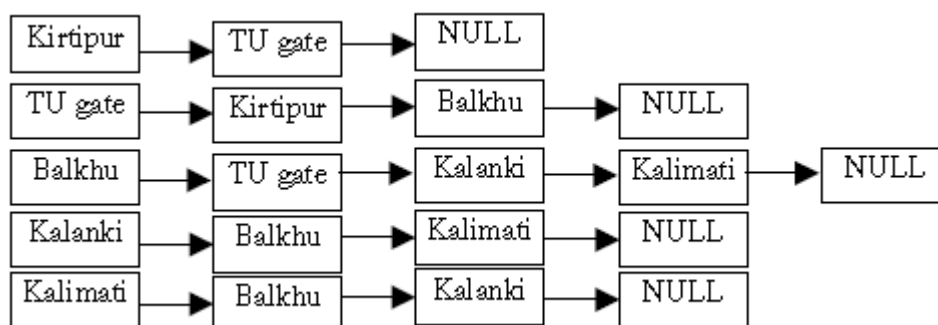
Generally we represent graph in two ways namely adjacency lists and adjacency matrix. Both ways can be applied to represent any kind of graph i.e. directed and undirected. An

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

**adjacency matrix** is an  $n \times n$  matrix  $M$  where  $M[i,j] = 1$  if there is an edge from vertex  $i$  to vertex  $j$  and  $M[i,j]=0$  if there is not. Adjacency matrices are the simplest way to represent graphs. This representation takes  $O(n^2)$  space regardless of the structure of the graph. So, if we have larger number of nodes say 100000 then we must have space for  $100000^2 = 10,000,000,000$  and this is quite a lot space to work on. The adjacency matrix representation of a graph for the above given road network graph is given below. Take the order {Kirtipur, TU gate, Balkhu, Kalanki, Kalimati}

0	1	0	0	0
1	0	1	0	0
0	1	0	1	1
0	0	1	0	1
0	0	1	1	0

It is very easy to see whether there is edge from a vertex to another vertex ( $O(1)$  time), what about space? Especially when the graph is sparse or undirected. If **adjacency list** representation of a graph contains an array of size  $n$  such that every vertex that has edge between the vertex denoted by the vertex with array position is added as a list with the corresponding array element. The example below gives the adjacency list representation of the above road network graph.



Searching for some edge  $(i,j)$  required  $O(d)$  time where  $d$  is the degree of  $i$  vertex.

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

**Some points:**

- To test if  $(x, y)$  is in graph adjacency matrices are faster.
- To find the degree of a vertex adjacency list is good
- For edge insertion and deletion adjacency matrix takes  $O(1)$  time where as adjacency list takes  $O(d)$  time.

## 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 =  $\Phi$ ; //an empty queue
    Enqueue(L,s);
    while (L !=  $\Phi$ )
    {
        v = dequeue(L);
        for each neighbor w to v
            if ( w  $\notin$  L and w  $\notin$  T )
```

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

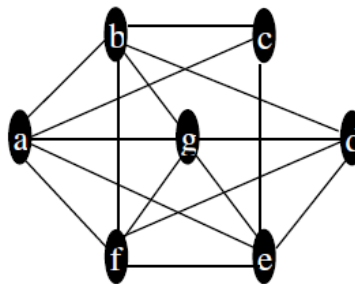
```

    {
        enqueue( L,w);
        T = T U {w}; //put edge {v,w} also
    }
}

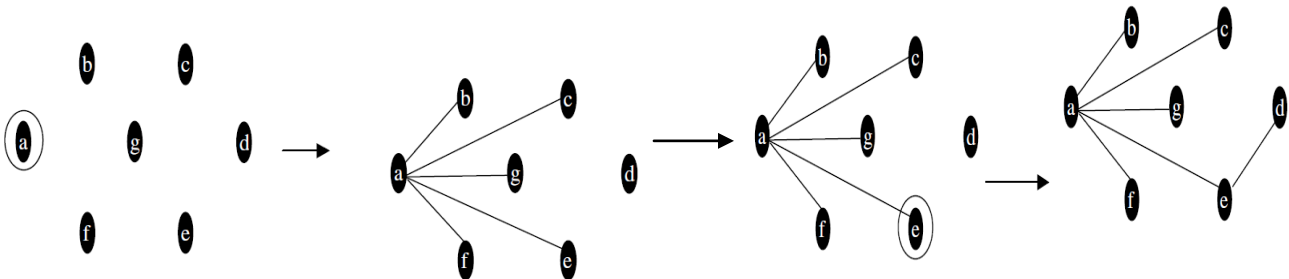
```

**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

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**



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**.

#### 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);
    }
}
```

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

**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.

for each execution the block inside the loop takes  $O(V)$  times . Hence the total running time is  $O(V^2)$ .

## Minimum Spanning Tree

A tree is defined to be an undirected, acyclic and connected graph (or more simply, a graph in which there is only one path connecting each pair of vertices). Assume there is an undirected, connected graph  $G$ . A spanning tree is a sub-graph of  $G$  that is tree and contains all the vertices of  $G$ . A minimum spanning tree is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

**Application of MST**

- Practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. MST can be used to determine the least costly paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- Another useful application of MST would be finding airline routes. MST can be applied to optimize airline routes by finding the least costly paths with no cycles

**Kruskal's algorithm**

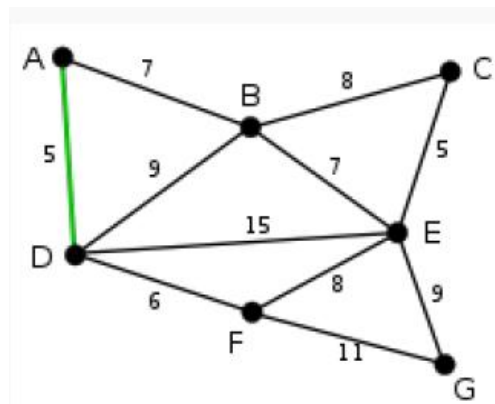
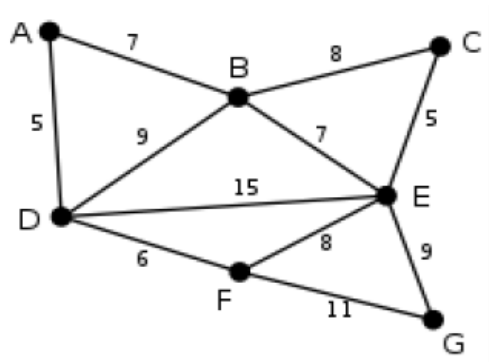
It is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected,

then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm. It works as follows:

- create a forest  $F$  (a set of trees), where each vertex in the graph is a separate tree
- create a set  $S$  containing all the edges in the graph
- while  $S$  is nonempty and  $F$  is not yet spanning
- remove an edge with minimum weight from  $S$
- if that edge connects two different trees, then add it to the forest, combining two trees into a single tree (i.e does not creates cycle)
- Otherwise discard that edge.

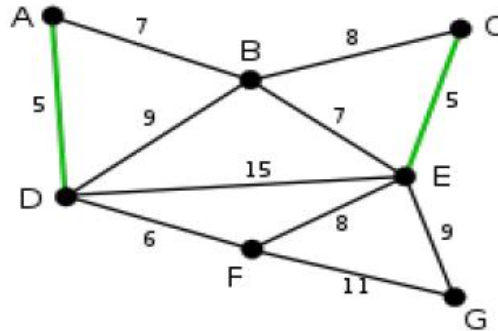
At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Consider the following Example, This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.

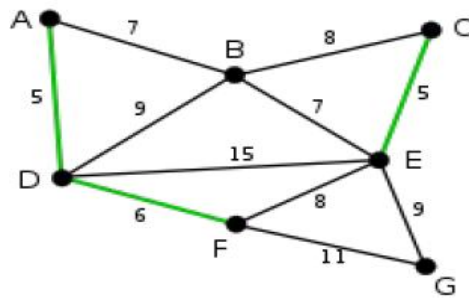


Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU

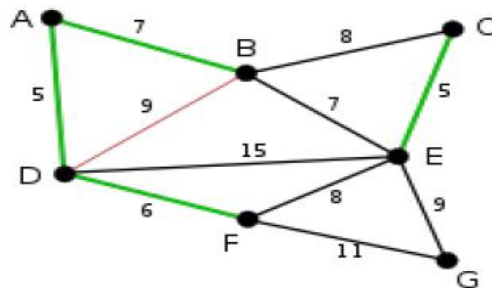
**AD** and **CE** are the shortest arcs, with length 5, and **AD** has been arbitrarily chosen, so it is highlighted.



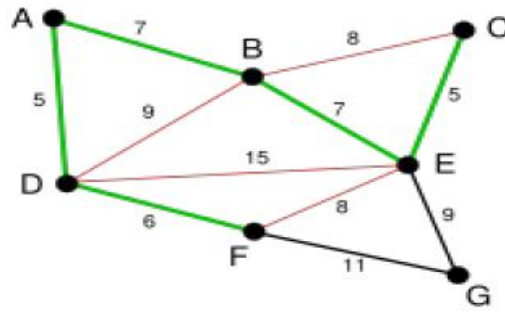
**CE** is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc.



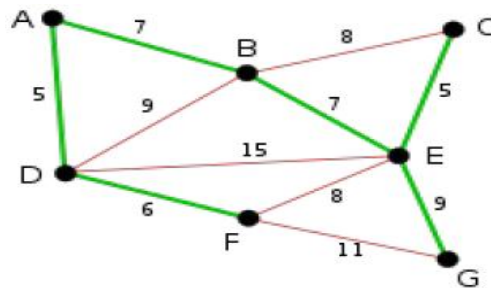
The next arc, **DF** with length 6, is highlighted using much the same method.



The next-shortest arcs are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The arc **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen.



The process continues to highlight the next-smallest arc, **BE** with length 7. Many more arcs are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.



Finally, the process finishes with the arc **EG** of length 9, and the minimum spanning tree is found.

### Algorithm:

KruskalMST(G)

{

$T = \{V\}$  // forest of n nodes

$S$  = set of edges sorted in nondecreasing order of weights

while( $|T| < n-1$  and  $E \neq \Phi$ )

{

Select (u,v) from S in order

Remove (u,v) from E

if((u,v) doesnot create a cycle in T))

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

}

Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU

```
}
```

### Analysis

In the above algorithm creating a tree forest at the beginning takes  $(V)$  time, the creation of set  $S$  takes  $O(\text{Elog}E)$  time and while loop execute  $O(V)$  times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is  $O(\text{Elog}E)$  which is asymptotically equivalently  $O(\text{Elog}V)$ .

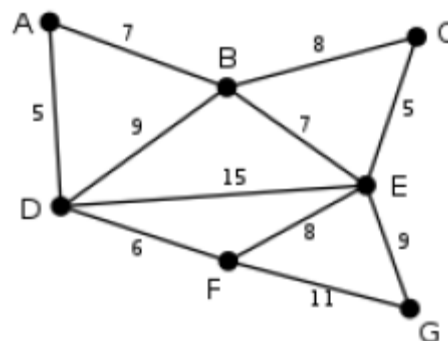
### Prim's algorithm

It is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

### How it works

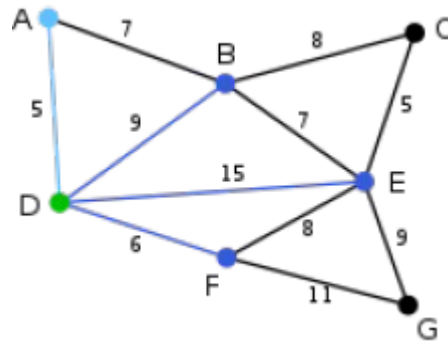
- This algorithm builds the MST one vertex at a time.
- It starts at any vertex in a graph (vertex A, for example), and finds the least cost vertex (vertex B, for example) connected to the start vertex.
- Now, from either 'A' or 'B', it will find the next least costly vertex connection, Without creating a cycle (vertex C, for example).
- Now, from either 'A', 'B', or 'C', it will find the next least costly vertex connection, without creating a cycle, and so on it goes.
- Eventually, all the vertices will be connected, without any cycles, and an MST
- will be the result.

Example,

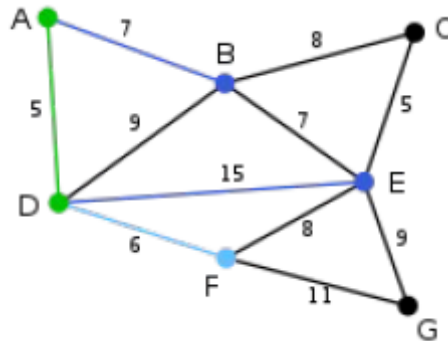


This is our original weighted graph. The numbers near the edges indicate their weight.

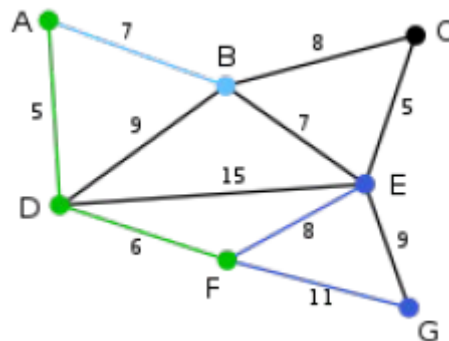
**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**



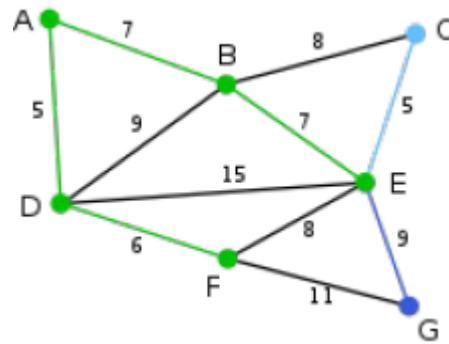
Vertex **D** has been arbitrarily chosen as a starting point. Vertices **A**, **B**, **E** and **F** are connected to **D** through a single edge. **A** is the vertex nearest to **D** and will be chosen as the second vertex along with the edge **AD**.



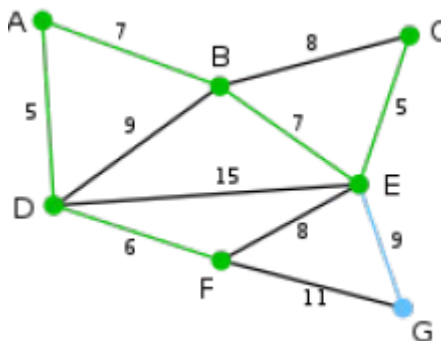
The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15, and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.



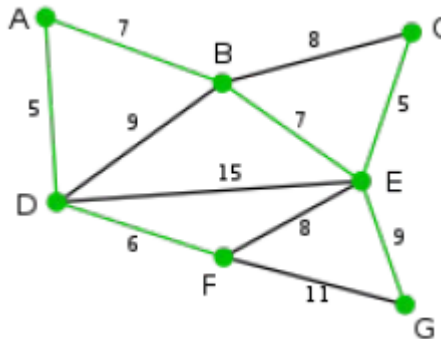
The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.



In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and **G** is 11 away from **F**. **E** is nearest, so we highlight the vertex **E** and the arc **BE**.



Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the arc **EC**.



Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight it and the arc **EG**.

### Algorithm:

PrimMST(G)

{

$T = \Phi$ ; //  $T$  is a set of edges of MST

Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU



```

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};
}
}

```

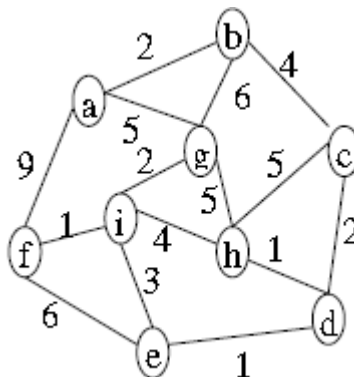
### Analysis

In the above algorithm while loop execute for  $O(V)$  time. 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. If we use heap data structure edge of minimum weight can be selected in  $O(\log E)$  time and the running time of prim's algorithm becomes  $O(E \log E)$  which is equivalent to  $O(E \log V)$ .

## Shortest Path Algorithms

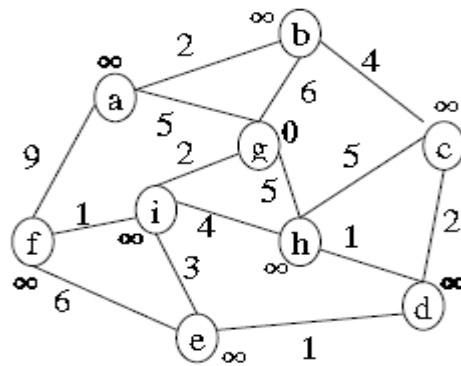
### Dijkstra Algorithm

This is an 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 below:

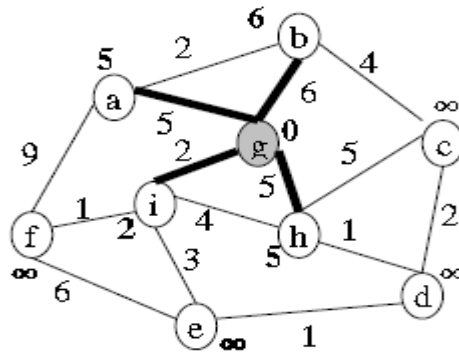


Initially, shortest path to all vertices from some given source is infinity and, suppose g is the source vertex.

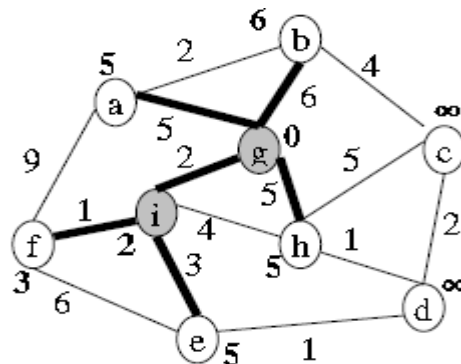
**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**



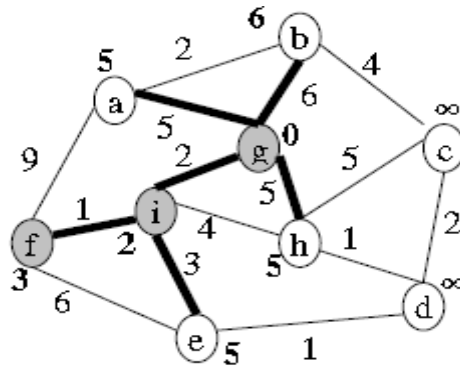
Take a vertex with shortest path (i.e vertex g), and relax all neighboring vertex. Vertex a, b, i and h gets relaxed.



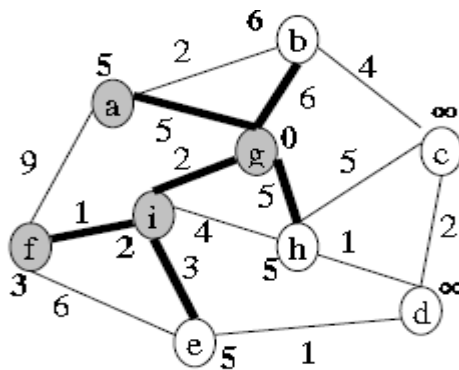
Again, Take a vertex with shortest path (i.e vertex i), and relax all neighboring vertex. Vertices f and e gets relaxed.



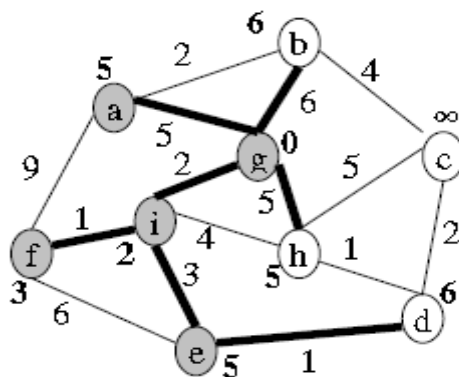
Take a vertex with shortest path (i.e vertex f), and relax all neighboring vertex. None of the vertices gets relaxed.



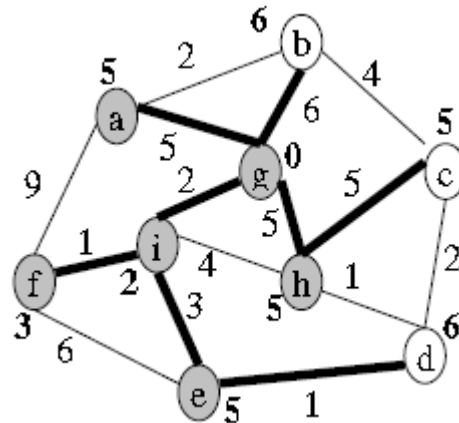
Take a vertex with shortest path (i.e vertex a or vertex e or vertex h), and relax all neighboring vertex. None of the vertices gets relaxed.



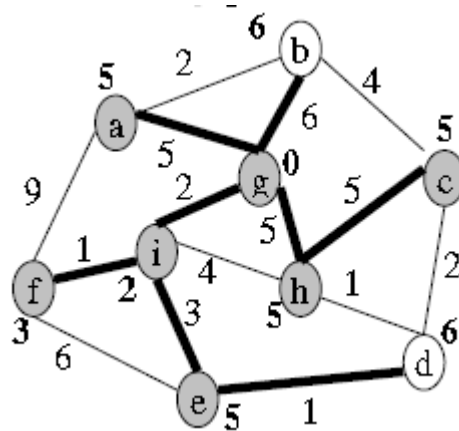
Take a vertex with shortest path (i.e vertex e or vertex h), and relax all neighboring vertex. Vertex d gets relaxed.



Take a vertex with shortest path (i.e vertex h), and relax all neighboring vertex. Vertex c gets relaxed



Take a vertex with shortest path (i.e vertex c), and relax all neighboring vertex. None of the vertices gets relaxed.



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

**Algorithm:**

```

Dijkstra(G,w,s)
{
    for each vertex  $v \in V$ 
        do  $d[v] = \infty$ 
     $p[v] = \text{Nil}$ 
     $d[s] = 0$ 
     $S = \Phi$ 

```

```

Q = V
While(Q !=  $\Phi$ )
{
    u = Take minimum from Q and delete.
    S = S  $\cup$  {u}
    for each vertex v adjacent to u
        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  $O(V^2)$ .

### 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  $n \times n$  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  $1, 2, \dots, k$  as intermediate vertices in the path. If we consider shortest path with intermediate vertices as above then computing the path contains two cases.  $D^k(i, j)$  does not contain  $k$  as intermediate vertex and  $D^k(i, j)$  contains  $k$  as intermediate vertex. Then we have the following relations  $D^k(i, j) = D^{k-1}(i, j)$ , when  $k$  is not an intermediate vertex, and  $D^k(i, j) = D^{k-1}(i, k) + D^{k-1}(k, j)$ , when  $k$  is an intermediate vertex. So from the above relations we obtain:

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

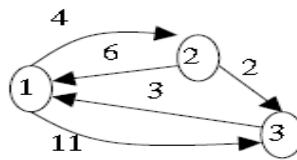
### Algorithm:

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

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 D0.
    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]};
}
```

**Example:**



Adjacency Matrix

W	1	2	3
1	0	4	11
2	6	0	2
3	3	$\infty$	0

D <sup>1</sup>	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

D <sup>2</sup>	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

D <sup>3</sup>	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.

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

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

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

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

$$\begin{aligned} D^1(3,1) &= \min\{D^0(3,1), D^0(3,1) + D^0(1,1)\} \\ &= \min\{3, 3 + 0\} = 3 \end{aligned}$$

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

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

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

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

$$\begin{aligned} D^2(2,3) &= \min\{D^1(2,3), D^1(2,2) + D^1(2,3)\} \\ &= \min\{2, 0 + 2\} = 2 \end{aligned}$$

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

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

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

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

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

$$\begin{aligned} D^3(2,3) &= \min\{D^2(2,3), D^2(2,3) + D^2(3,3)\} \\ &= \min\{2, 2 + 0\} = 2 \end{aligned}$$

$$\begin{aligned} D^3(3,1) &= \min\{D^2(3,1), D^2(3,3) + D^2(3,1)\} \\ &= \min\{3, 0 + 3\} = 3 \end{aligned}$$

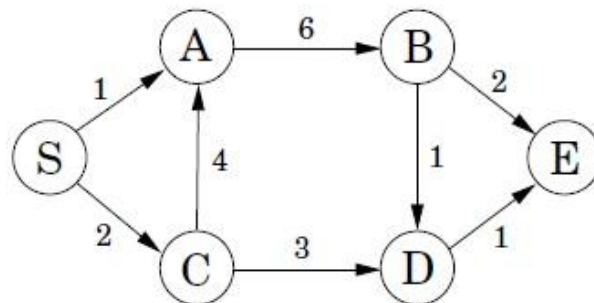
$$\begin{aligned} D^3(3,2) &= \min\{D^2(3,2), D^2(3,3) + D^2(3,2)\} \\ &= \min\{7, 0 + 7\} = 7 \end{aligned}$$

### Analysis:

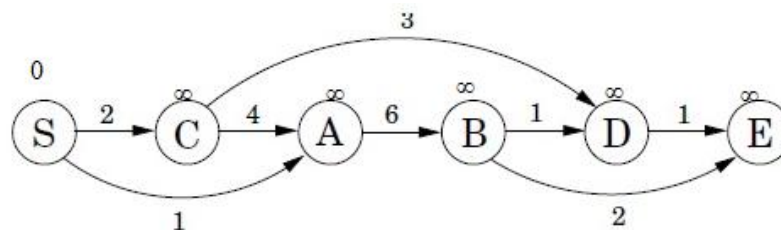
Clearly the above algorithm's running time is  $O(n^3)$ , where  $n$  is cardinality of set  $V$  of vertices.

**Directed Acyclic Graph (DAG)**

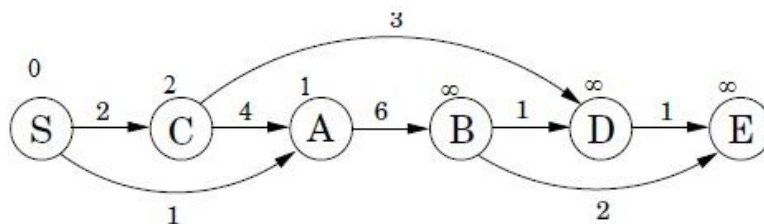
DAG, here directed means that each edge has an arrow denoting that the edge can be traversed in only that particular direction. Acyclic means that the graph has no cycles, i.e., starting at one node, you can never end up at the same node. DAG can be used to find shortest path from a given source node to all other nodes. To find shortest path by using DAG, first of all sort the vertices of graph topologically and then relax the vertices in topological order.

**Example:**

**Step1:** Sort the vertices of graph topologically

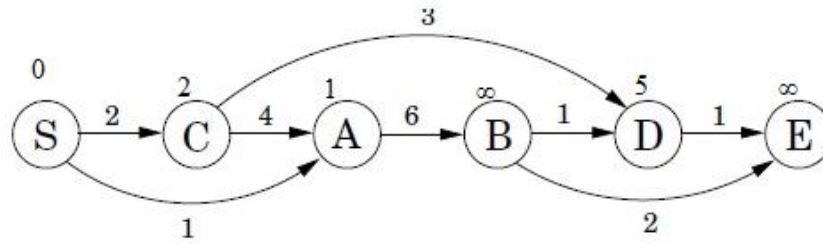


**Step2:** Relax from S  $\infty$

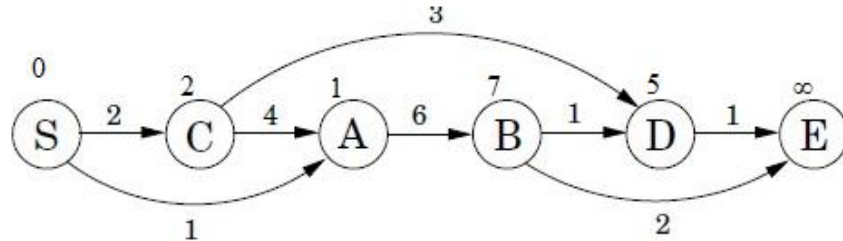


**Step3:** Relax from C

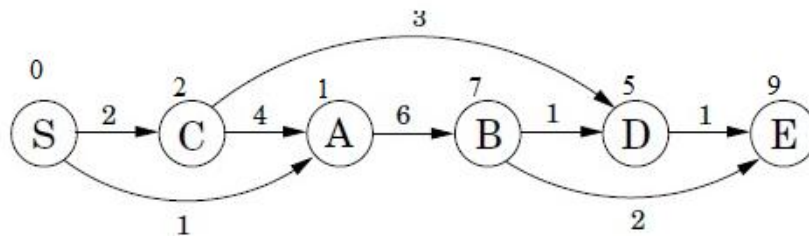




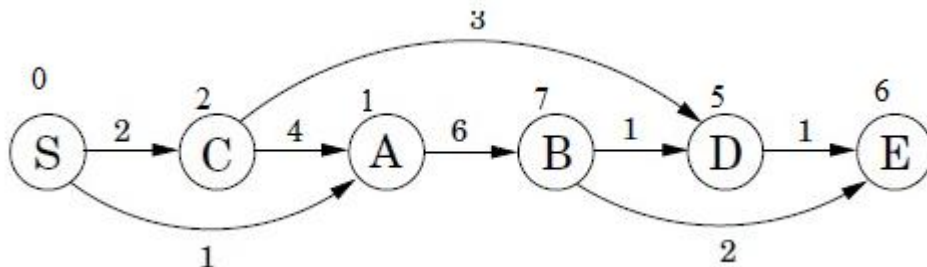
**Step4:** Relax from A



**Step5:** Relax from B



**Step6:** Relax from D



### Algorithm

DagSP( $G, w, s$ )

{

Topologically Sort the vertices of  $G$

for each vertex  $v \in V$

$d[v] = \infty$

Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU

```
d[s] = 0
for each vertex u, taken in topologically sorted order
    for each vertex v adjacent to u
        if d[v] > d[u] + w(u,v)
            d[v] = d[u] + w(u,v)
}
```

**Analysis:**

In the above algorithm, the topological sort can be done in  $O(V+E)$  time (Since this is similar to DFS! see book.). The first for loop block takes  $O(V)$  time. In case of second for loop it executes in  $O(V^2)$  Time so the total running time is  $O(V^2)$ . Aggregate analysis gives us the running time  $O(E+V)$ .

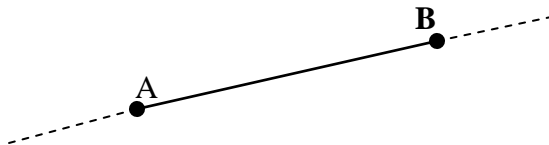
## Chapter 7

### Geometric Algorithms

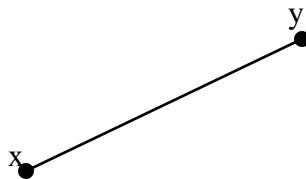
The easy explanation is that geometry algorithms are what a software developer programs to solve geometry problems. And we all know what geometry problems are, right? The simplest of such problems might be to find the intersection of two lines, the area of a given region, or the inscribed circle of a triangle. Methods and formulas have been around for a long time to solve such simple problems. But, when it comes to solving even these simple problems as accurate, robust, and efficient software programs, the easy formulas are sometimes inappropriate and difficult to implement. This is the starting point for geometry algorithms as methods for representing elementary geometric objects and performing the basic constructions of classical geometry.

### Geometric Primitives

A line is a group of points on a straight path that extends to infinity. Any two points on the line can be used to name it.



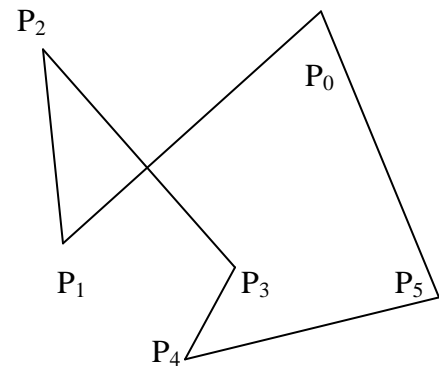
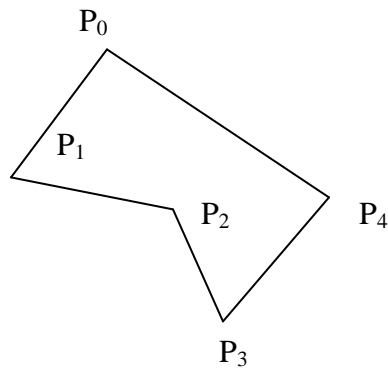
A line segment is a part of a line that has two end points. The two end points of the line segment are used to name the line segment.



### Polygon

Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU

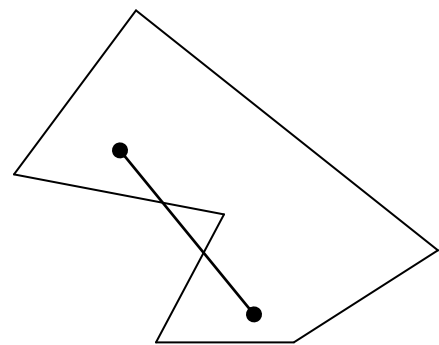
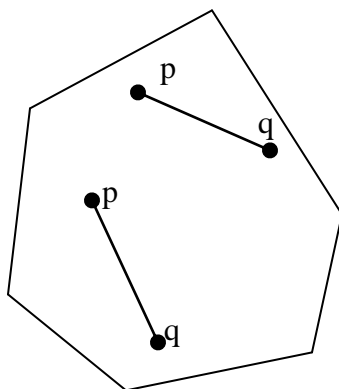
A closed figure of  $n$  line segments, where  $n \geq 3$ . The polygon  $P$  is represented by its vertices, usually in counterclockwise order of traversal of its boundary,  $P = (p_0, p_1, \dots, p_{n-1})$  or the line segments that are ordered as  $P = (S_0, S_1, \dots, S_{n-1})$  such that the end point of preceding line segment becomes starting point of the next line segment.



A Simple polygon is a polygon  $P$  with no two non-consecutive edges intersecting. There is a well-defined bounded interior and unbounded exterior for a simple polygon, where the interior is surrounded by edges.

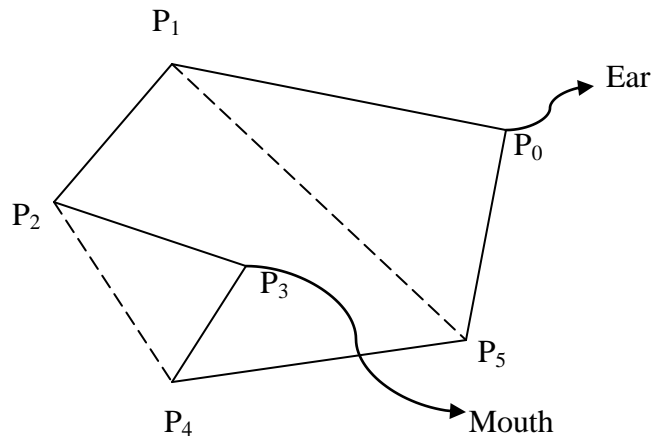
### Convex Polygon

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



### Ear and Mouth

A vertex  $p_i$  of a simple polygon  $P$  is called an ear if for the consecutive vertices  $p_{i-1}, p_i, p_{i+1}$  ( $p_{i-1}, p_{i+1}$ ) is a diagonal. A vertex  $p_i$  of a simple polygon  $P$  is called a mouth if the diagonal  $(p_{i-1}, p_{i+1})$  is an external diagonal.



### Convex Hull

The convex hull of a polygon  $P$  is the smallest convex polygon that contains  $P$ . Similarly, we can define the convex hull of a set of points  $R$  as the smallest convex polygon containing  $R$ .



### Computing point of intersection between two line segments

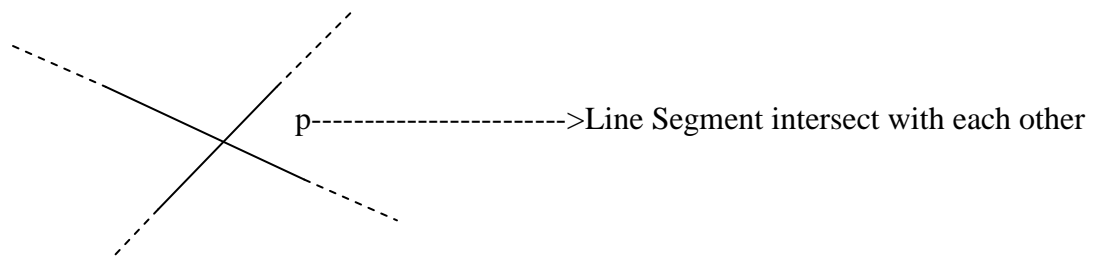
We can apply our coordinate geometry method for finding the point of intersection between two line segments. Let  $S1$  and  $S2$  be any two line segments. The following steps are used to calculate point of intersection between two line segments. We are not considering parallel line segments here in this discussion.

- Determine the equations of line through the line segment  $S1$  and  $S2$ . Say the equations are  $L1 = (y = m1x + c1)$  and  $L2 = (y = m2x + c2)$  respectively. We can find the equation

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

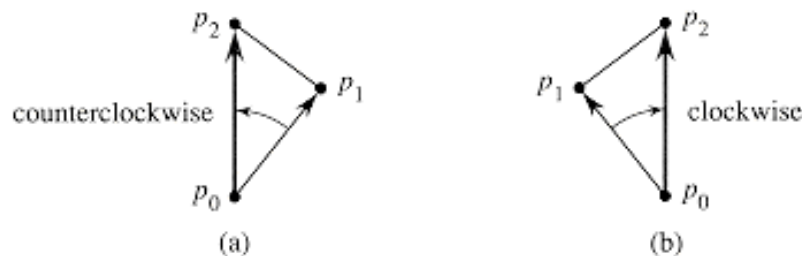
of line L1 using the formula of slope  $(m_1) = (y_2 - y_1) / (x_2 - x_1)$ , where  $(x_1, y_1)$  and  $(x_2, y_2)$  are two given end points of the line segment S1. Similarly we can find the  $m_2$  for L2 also. The values of  $c_i$ 's can be obtained by using the point of the line segment on the obtained equation after getting slope of the respective lines.

- Solve two equations of lines L1 and L2, 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 S2. 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 S2.



### Determining Intersection between two Line Segments

Whether two consecutive line segments turn left or right at point  $p_1$ . Cross products allow us to answer this question without computing the angle. We simply check whether directed segment is clockwise or counterclockwise relative to directed segment. To do this, we compute the cross product  $(p_2 - p_0) \times (p_1 - p_0)$ . If the sign of this cross product is negative, then is counterclockwise with respect to  $P_1$ , and thus we make a left turn at  $P_1$ . A positive cross product indicates a clockwise orientation and a right turn. A cross product of 0 means that points  $p_0$ ,  $p_1$ , and  $p_2$  are collinear.



We compute the cross product of the vectors given by two line segments as:

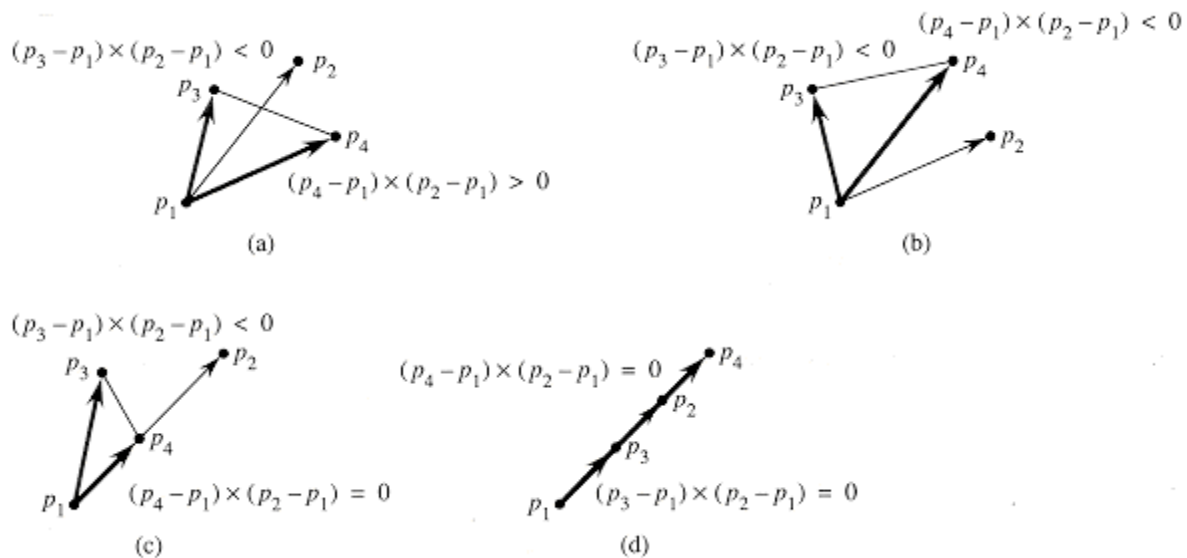
$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0, y_1 - y_0) \wedge (x_2 - x_0, y_2 - y_0) = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0)$ , this can be represented as

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

Here we have,

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

Using the concept of left and right turn we can detect the intersection between the two line segments in very efficient manner. Two segments  $S_1 = (P, Q)$  and  $S_2 = (R, S)$  do not intersect if PQR and PQS are of same turn type or RSP and RSQ are of same turn type.



## Graham Scan Algorithm

The convex hull of a set  $Q$  of points is the smallest convex polygon  $P$  for which each point in  $Q$  is either on the boundary of  $P$  or in its interior. Graham's scan solves the convex-hull problem by maintaining a stack  $S$  of candidate points. Each point of the input set  $Q$  is pushed once onto the stack, and the points that are not vertices of convex hull are eventually popped from the stack. When the algorithm terminates, stack  $S$  contains exactly the vertices of convex hull, in counterclockwise order of their appearance on the boundary.

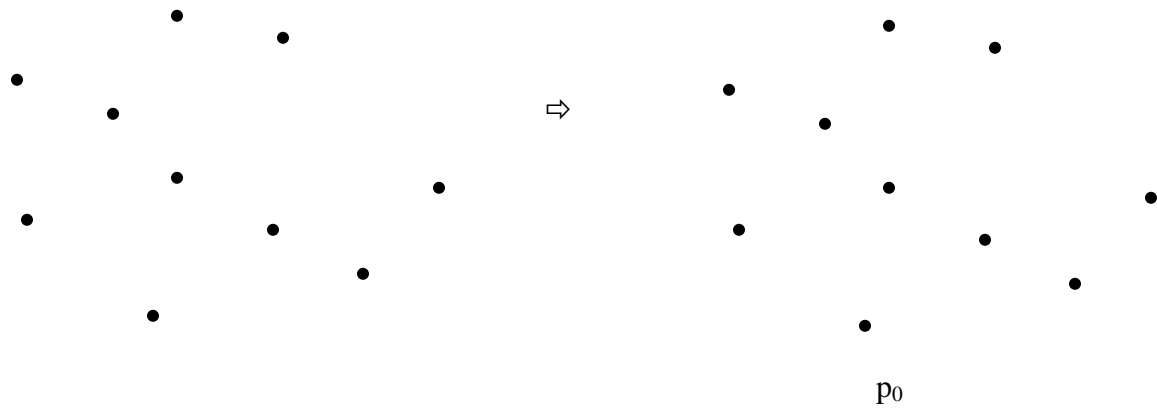
The algorithm starts by picking a point in  $Q$  known to be a vertex of the convex hull. This can be done in  $O(n)$  time by selecting the rightmost lowest point in the set; that is, a point with first a minimum (lowest)  $y$  coordinate. Having selected this base point, call it  $p_0$ , the algorithm then sorts the other points  $p$  in  $Q$  by the increasing counter-clockwise angle the line segment  $p_0p$  makes with the  $x$ -axis. If there is a tie and two points have the same angle, discard the one that is closest to  $p_0$ .

Given Points:

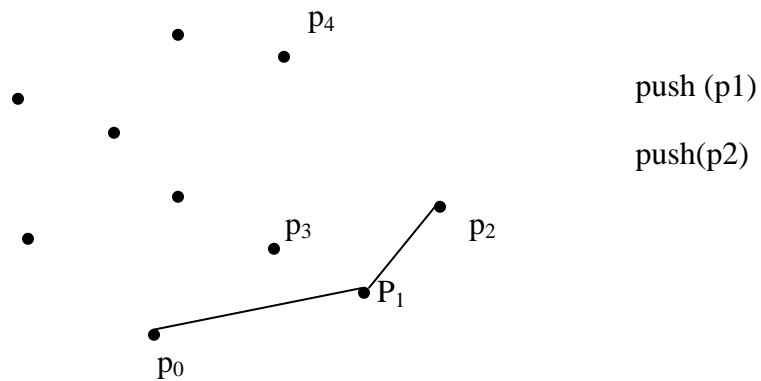
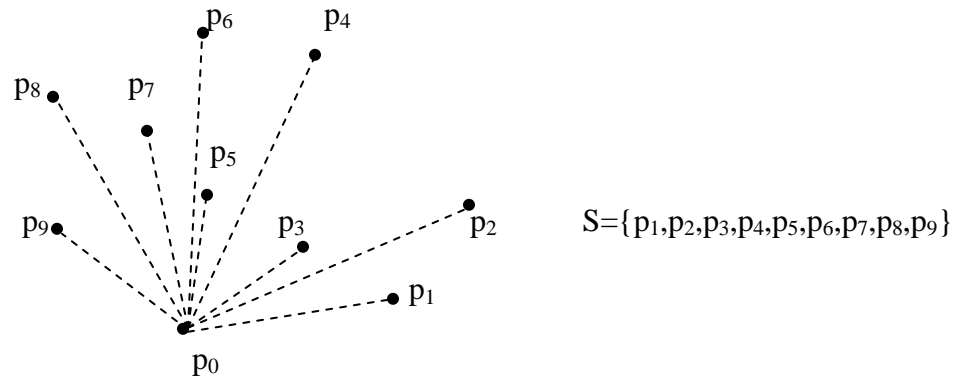
Select point with lowest  $y$ -coordinate

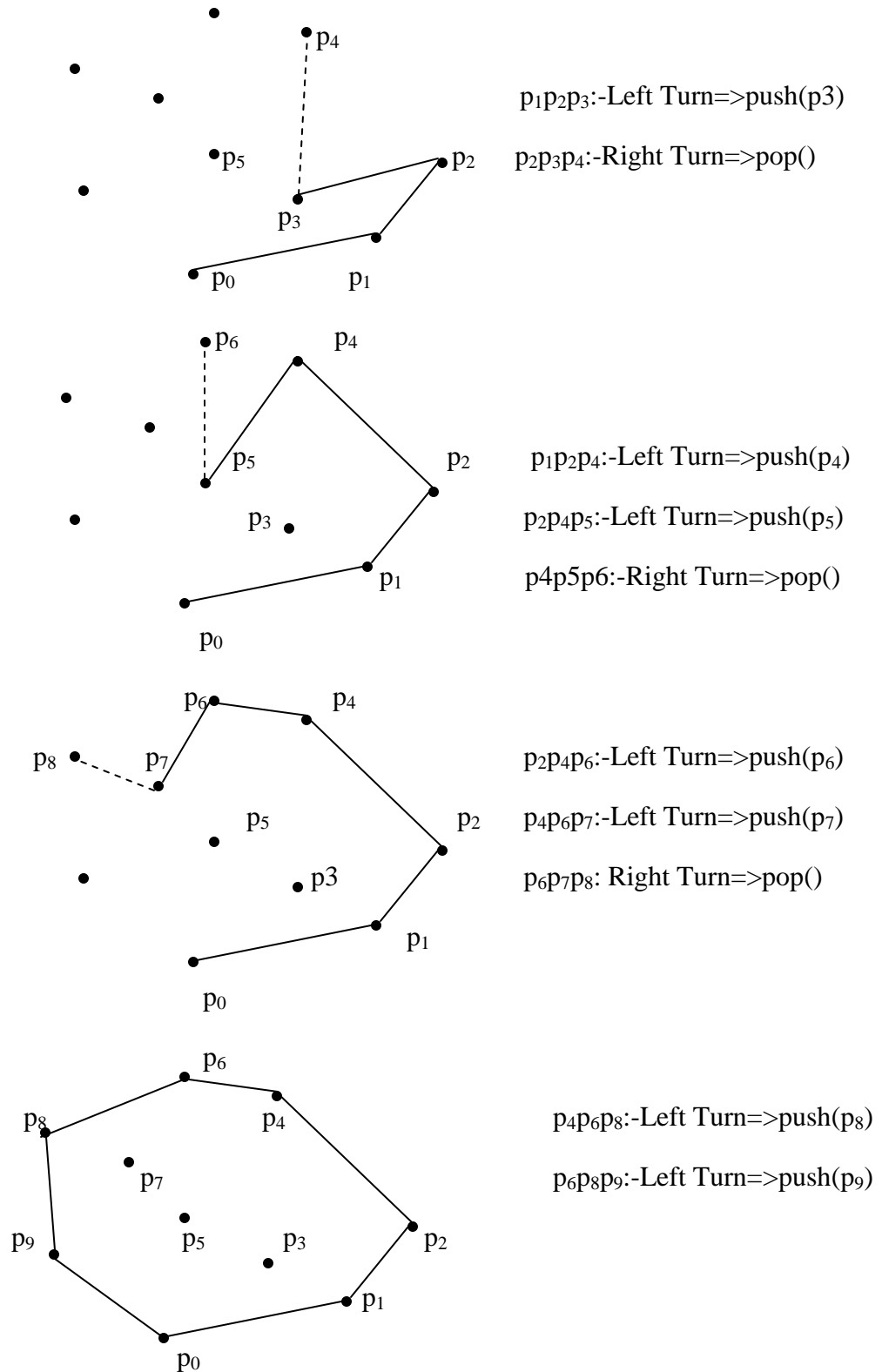
**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**





Sort points in increasing order of angle in counter clockwise order, say sorted set of points is  $S$





**Algorithm**

GrahamScan(P)

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

**Analysis**

It requires  $O(n)$  time to find  $p_0$ . Sorting of points require  $O(n \log n)$  time,. Push operation takes constant time i.e.,  $O(1)$ . We can understand that the while loop is executed  $O(m-2)$  times in total. Thus we can say that while loop takes  $O(1)$  time. So the worst case running time of the algorithm is  $T(n) = O(n) + O(n \log n) + O(n) = O(n \log n)$ , where  $n = |P|$ .

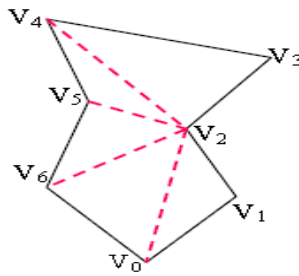
**Polygon Triangulation**

Triangulation of a simple polygon  $P$  is decomposition of  $P$  into triangles by a maximal set of non-intersecting diagonals. Diagonal is an open line segment that connects two vertices of  $P$  and

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

lies in the interior of P. Triangulations are usually not unique. Any triangulation of a simple polygon with  $n$  vertices consists of exactly  $n-2$  triangles.

Triangulation is done by adding the diagonals. For polygon with  $n$ -vertices, the candidate diagonals can be  $O(n^2)$ . Check intersection of  $O(n^2)$  segments with  $O(n)$  edges it costs  $O(n^3)$ . There can be total of  $n-3$  diagonals on triangulation. So total cost in triangulation is  $O(n^4)$ .



Chapter 8**NP-Completeness**

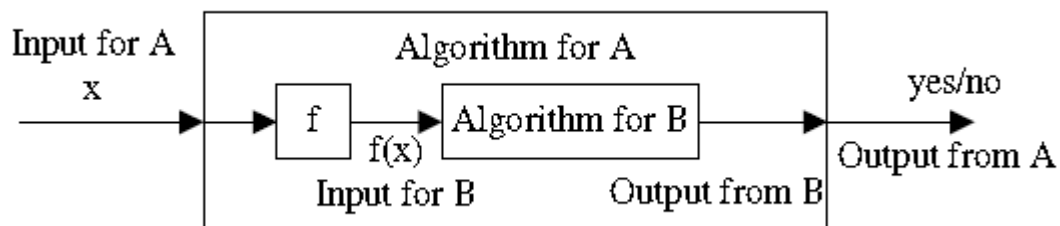
Most of the problems considered up to now can be solved by algorithms in worst-case polynomial time. There are many problems and it is not necessary that all the problems have the apparent solution. This concept, somehow, can be applied in solving the problem using the computers. The computer can solve: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

**Tractable and Intractable Problems**

We call problems as tractable or easy, if the problem can be solved using polynomial time algorithms. The problems that cannot be solved in polynomial time but requires 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.

**Polynomial time reduction**

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



### **P and NP classes and NP completeness**

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

#### **Complexity Class P**

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

#### **Complexity Class NP**

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

#### **Complexity Class NP-Complete**

**Prepared By: Arjun Singh Saud, Faculty CDCISIT, TU**

NP complete problems are those problems that are hardest problems in class NP. We define some problem say A, is NP-complete if

- a.  $A \in NP$ , and
- b.  $B \leq_p A$ , for every  $B \in NP$ .

The problem satisfying property b is called NP-hard.

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

## **Cook's Theorem**

SAT is NP-complete

### **Proof**

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

- $SAT \in NP$
- SAT is NP-Hard

### **SAT $\in$ NP**

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

### **SAT is NP-hard**

Take a problem  $V \in NP$ , let A be the algorithm that verifies V in polynomial time (this must be true since  $V \in NP$ ). 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.

## Approximation Algorithms

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

### Vertex Cover Problem

A **vertex cover** of an undirected graph  $G=(V,E)$  is a subset  $V' \subseteq V$  such that for all edges  $(u,v) \in E$  either  $u \in V'$  or  $v \in V'$  or  $u$  and  $v \in V'$ . 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.

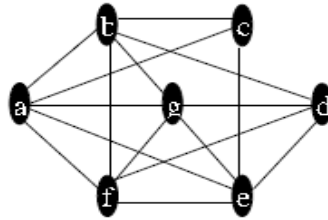
### Algorithm

ApproxVertexCover ( $G$ )

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

**Example:** (vertex cover running example for graph below)





**Solution:**

