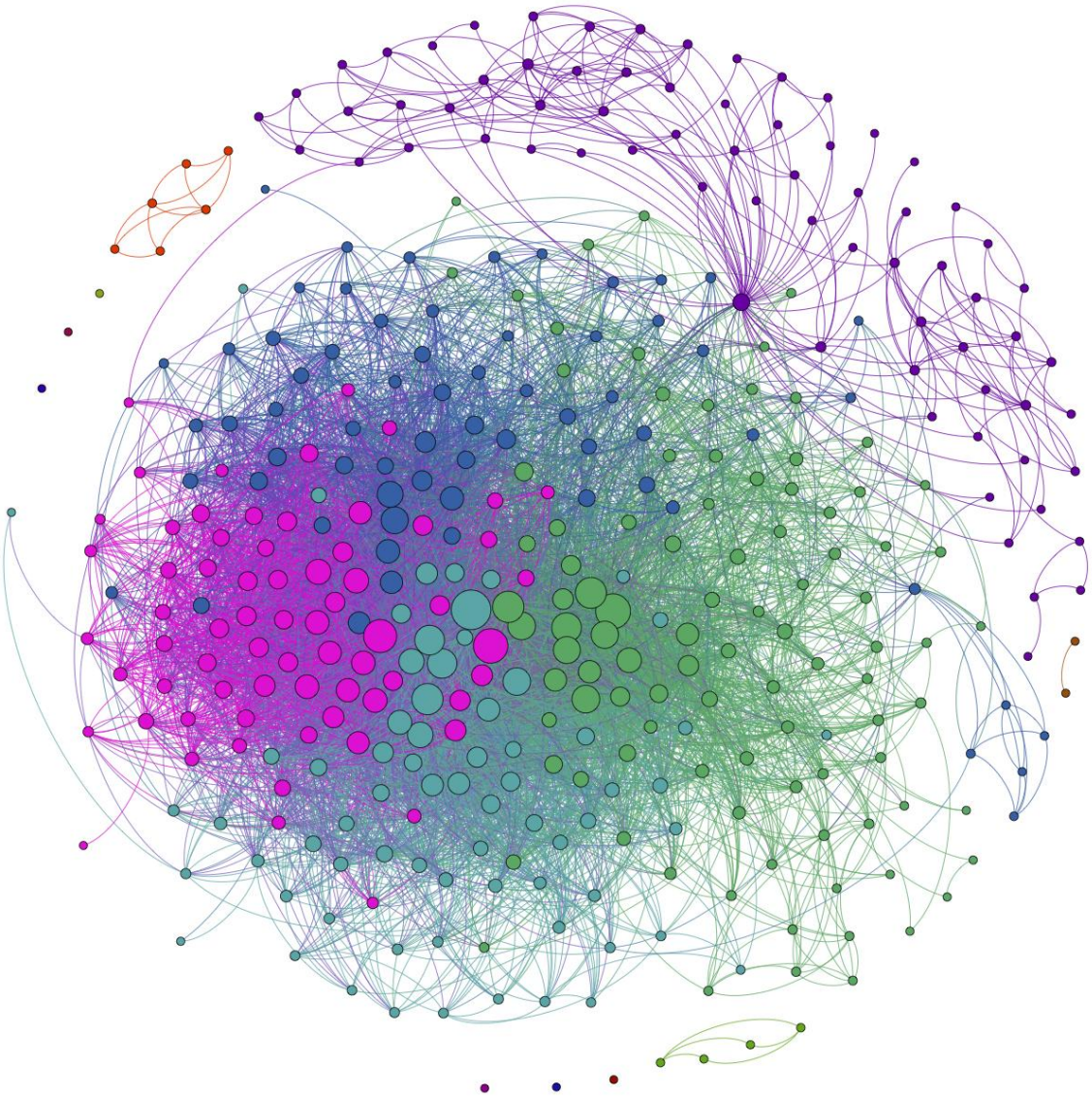
Design and Analysis of Algorithms (CSC-303)
BScCSIT 5th



By Bishnu Rawal

1.1 Algorithm analysis

Algorithm

An *algorithm* is a finite set of computational steps; each step can be executed in finite time, to perform computation or problem solving by transforming **input** (some value(s)) into the **output** (some value, or set of values). Algorithms are not dependent on a particular machine or programming language. We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies the desired input/output relationship.

For example, one might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the **sorting problem**:

> **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
>
> **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

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**.

Algorithm Properties

A. **Input(s)/output(s)**: There must be some inputs from the standard set of inputs and an algorithm's execution must produce outputs(s).
B. **Definiteness**: Each step must be clear and unambiguous.
C. **Finiteness**: Algorithms must terminate after finite time or steps.
D. **Correctness**: Correct set of output values must be produced from the each set of inputs.
E. **Effectiveness**: Each step must be carried out in finite time.

Random-Access Machine (RAM) Model

We want to predict the resources that the algorithm requires, usually, running time. In order to predict resource requirements; a computational model, RAM is used.

- Instructions are executed one after another. No concurrent operations.
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll use instructions commonly found in real computers:
    - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling. Also, shift left/shift right (good for multiplying/dividing by 2^k)
    - Data movement: load, store, copy.
    - Control: conditional/unconditional branch, subroutine call and return.

  Each of these instructions takes a constant amount of time.

The RAM model uses integer and floating point data types.

Best, Worst and Average case

Best case complexity gives lower bound on the running time of the algorithm for any instance of input(s). This indicates that the algorithm can never have lower running time than best case for particular class of problems.
Worst case complexity gives upper bound on the running time of the algorithm for all the instances of the input(s). This insures that no input can overcome the running time limit posed by worst case complexity.
Average case complexity gives average number of steps required on any instance of the input(s).

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

Input size: Depends on the problem being studied.

- Usually, the number of items in the input. Like the size *n* of the array being sorted.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Running time: On a particular input, it is the number of *primitive operations* (steps) executed.

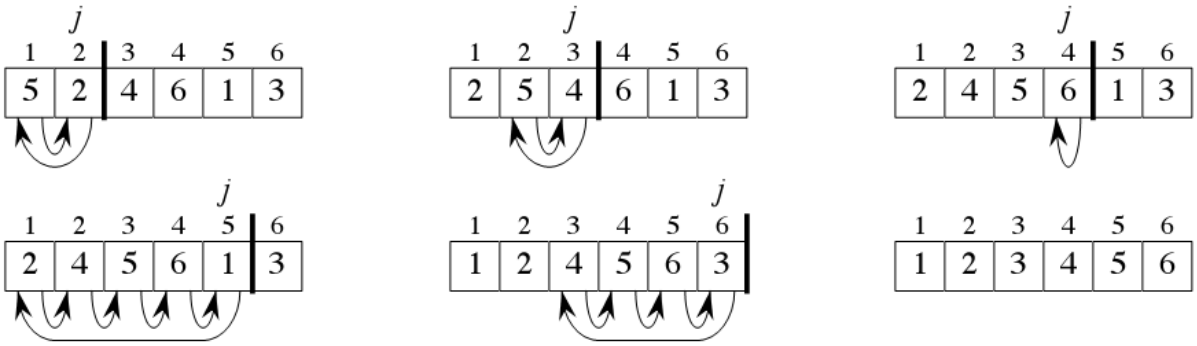
- We want to define steps to be machine-independent.
- Figure that each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line *i* takes the same amount of time c_i.
- We are assuming that the line consists only of primitive operations.
    - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: sort the points by x-coordinate.

Example

Let us look at the sorting problem, **insertion sort** (natural for small input size) having incremental design strategy. (Merge sort for example has a divide-and-conquer design; we'll look into it later)

INSERTION-SORT (A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Illustration:



Analysis:

- Assume that the i^{th} line takes time c_i, which is a constant. (Since the third line i is a comment, it takes no time.)
- For $j = 2, 3...n$, let t_j be the number of times that while loop test is executed for that value of j.

  Note: when a **for/while** loop exits in the usual way due to the test in the loop header –the test is executed one time more than the loop body.

Let T(n) be the Total running time of insertion sort,

$$T(n) = \sum_{all\ statements} (Cost\ of\ statement).(Number\ of\ times\ the\ statement\ is\ executed)$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

The running time depends on the values of t_j, which in turn varies according to input.

Best case: The array is already sorted.

- Always find that $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j are 1.
- Running time is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) .$$

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow T(n)$ is a *linear function* of n.

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.
- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.
- Letting $k = j - 1$, we see that $\sum_{j=2}^{n} (j - 1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Average case: On average, the key in *A[j]* is less than half the elements in *A [1... j -1]* and it's greater than the other half.

- On average, the while loop has to look half way through the sorted subarray A [1... j -1] to decide where to drop *key*.
- $t_j = j/2$.

Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.

We usually concentrate on finding the worst-case running time: the longest running time for any input of size *n* since it gives a guaranteed upper bound on the running time for any input.

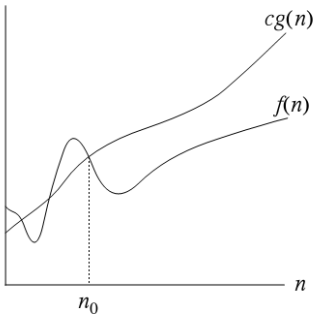
Asymptotic Notation

When we consider an algorithm for some problem, in addition to knowing that it produces a correct solution, we will be especially interested in analyzing its running time. There are several aspects of running time that one could focus on. Our focus will be primarily on the question: "how does the running time scale with the size of the input?" This is called asymptotic analysis, and the idea is that we will ignore low-order terms and constant factors, focusing instead on the shape of the running time curve. We will typically use n to denote the size of the input, and T(n) (or any function name viz. f(n)) to denote the running time of our algorithm on an input of size n. We begin by presenting some convenient definitions for performing this kind of analysis.

[**Hey guys**! In analytic geometry, an **asymptote** (Greek *asumptotos*, which means "not falling together") of a curve is a line such that the distance between the curve and the line approaches zero as they tend to infinity.]

O-Notation (Big-Oh)

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



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Informally, we can say: *f(n)* is proportional to *g(n)*, or better, as *n* gets large.

Example: $2n^2 = O(n^3)$, with c = 1 and $n_0 = 2$.

Examples of functions in $O(n^2)$ are: n^2, $n^2 + n$, $n^2 + 1000n$, $1000n^2 + 1000n$, n, n/1000, $n^{1.99999}$, $n^2/lglglgn$

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

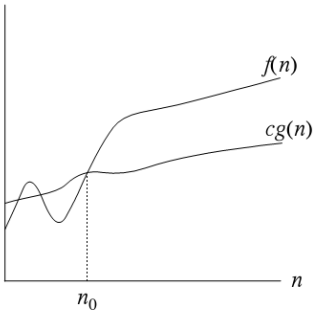
Proof: let's choose c and n_0 values as 14 and 1 respectively then we'll have f(n) <= c*g(n), n >= n_0 as ($3n^2 + 4n + 7$) <= $14*n^2$ for all n >= 1, since the inequality is trivially true, we can show f(n) = *O*(g(n))

Pitfalls of O-Notation:

- Not useful for small input sizes because the constants and smaller terms will matter.
- Omission of the constants can be misleading
  - For example, **2N log N** and **1000 N**
  - Even though its growth rate is larger, the 1st function is probably better. Because the 1000 constant could be memory accesses or disk accesses.
- Assumes an infinite amount of memory
  - Not trivial when using large data sets.
- Accurate analysis relies on clever observations to optimize the algorithm.

Ω-Notation (Big-Omega)

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



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

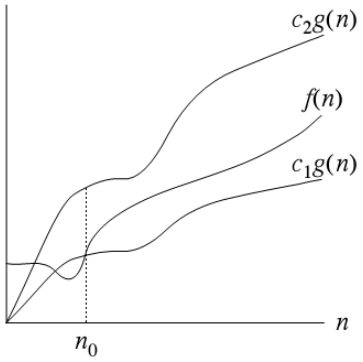
Example: $\sqrt{n} = \Omega(\log n)$, with c = 1 and n_0 = 16.

Examples of functions in $\Omega(n^2)$: n^2, $n^2 + n$, $n^2 - n$, $1000n^2 - 1000n$, $1000n^2 - 1000n$, n^3, $n^{2.00001}$, $n^2 lglglgn$

1. f(n) = $3n^2 + 4n + 7$, g(n) = n^2, prove that $f(n) = \Omega(g(n))$.
   Proof: let us choose c and n_0 values as 1 and 1, respectively then we can have
   f(n) >= c*g(n), n>=n i.e. $3n^2 + 4n + 7 >= 1*n^2$ for all n >= 1
   The above inequality is trivially true.
   Hence, $f(n) = \Omega(g(n))$.

Θ-Notation (Big-Theta)

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Example:

$n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

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

o-Notation (Little Oh)

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

Examples:

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

ω-Notation (Little Omega)

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

Examples:

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

Comparisons of functions

Relational properties:

Transitivity:

f (n) = Θ(g(n)) and *g(n) = Θ(h(n))⇒ f (n) = Θ(h(n))*. Same for *O, Ω, o,* and *ω*.

Reflexivity:

f (n) = Θ(f (n)). Same for *O* and Ω.

Symmetry:

f (n) = Θ(g(n)) if and only if *g(n) = Θ(f (n))*.

Standard Notations and Common Functions

- Monotonicity
  - $f(n)$ is ***monotonically increasing*** if $m \le n \Rightarrow f(m) \le f(n)$.
  - $f(n)$ is ***monotonically decreasing*** if $m \ge n \Rightarrow f(m) \ge f(n)$.
  - $f(n)$ is ***strictly increasing*** if $m < n \Rightarrow f(m) < f(n)$.
  - $f(n)$ is ***strictly decreasing*** if $m > n \Rightarrow f(m) > f(n)$.
- Exponentials

  Useful identities:

$$\begin{aligned} a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ a^m a^n &= a^{m+n} \end{aligned}$$

  We can relate rates of growth of polynomials and exponentials: For all real constants a and b such that a > 1, $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$, which implies that $n^b = O(a^n)$.

- Logarithms

Notations:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} ,\\
\ln n &= \log_e n && \text{(natural logarithm)} ,\\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} ,\\
\lg\lg n &= \lg(\lg n) && \text{(composition)} .
\end{aligned}$$

Useful identities for all real $a > 0$, $b > 0$, $c > 0$, and n, and where logarithm bases are not 1:

$$\begin{aligned}
a &= b^{\log_b a} ,\\
\log_c(ab) &= \log_c a + \log_c b ,\\
\log_b a^n &= n \log_b a ,\\
\log_b a &= \frac{\log_c a}{\log_c b} ,\\
\log_b(1/a) &= -\log_b a ,\\
\log_b a &= \frac{1}{\log_a b} ,\\
a^{\log_b c} &= c^{\log_b a} .
\end{aligned}$$

Changing the base of a logarithm from one constant to another only changes the value by a constant factor, so we usually don't worry about logarithm bases in asymptotic notation. Convention is to use lg within asymptotic notation, unless the base actually matters.

Just as polynomials grow more slowly than exponentials, logarithms grow more slowly than polynomials. In $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$, substitute $\lg n$ for n and 2^a for a:

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 ,$$

implying that $\lg^b n = o(n^a)$.

In the expression $\log_b a$:

- If we hold *b* constant, then expression is strictly increasing as *a* increases.
- If we hold *a* constant, then expression is strictly decreasing as *b* increases.

Recurrences

A **recurrence** is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more base cases and one or more recursive cases. Each of these cases is an equation or inequality, with some function value f(n) on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n The recursive cases relate the function value f (n) to function value f (k) for one or more integers k < n; typically, each recursive case applies to an infinite number of possible values of n.

For example:

- Recursive algorithm for finding factorial
  T(n)=1 when n =1
  T(n)=T(n-1) + O(1) when n>1

- Recursive algorithm for finding Nth Fibonacci number
  T(1)=1 when n=1
  T(2)=1 when n=2
  T(n)=T(n-1) + T(n-2) +O(1) when n>2
- Recursive algorithm for binary search
  T(1)=1 when n=1
  T(n)=T(n/2) + O(1) when n>1

Other examples:

- $T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n-1) + 1 & \text{if } n > 1. \end{cases}$
  Solution: $T(n) = n.$
- $T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{if } n \geq 1. \end{cases}$
  Solution: $T(n) = n \lg n + n.$
- $T(n) = \begin{cases} 0 & \text{if } n = 2, \\ T(\sqrt{n}) + 1 & \text{if } n > 2. \end{cases}$
  Solution: $T(n) = \lg \lg n.$
- $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 \end{cases}$
  Solution: $T(n) = \Theta(n \lg n).$

Solving recurrences

- Eliminating recursion from the function

1. Iteration Method

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

Example:

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

Solution:

T(n) = 2T(n/2) +1
= 2 { 2T(n/4) + 1} +1
= 4T(n/4) + 2 + 1
= 4 { T(n/8) +1 } + 2 + 1
= 8 T(n/8) + 4 + 2 + 1
……..
= 2^k T($n/2^k$) + 2^{k-1}+…+ 4 + 2 + 1
To reach to the base case T(1),
$n/2^k = 1$
or, $n = 2^k$
hence, k = logn [taking log on both sides and using $\log_2 2 = 1$]
T(n)= $2^k + 2^{k-1} + ... + 2^2 + 2^1 + 2^0$
= $(2^{k+1} -1)/ (2-1)$
= $2^{k+1} -1$
= $2.2^k -1$
= 2n-1
= O(n)

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

Solution:
T(n) = T(n/3) + O(n)
T(n) = T($n/3^2$) + O(n/3) + O(n)

$T(n) = T(n/3^3) + O(n/3^2) + O(n/3) + O(n)$
$T(n) = T(n/3^4) + O(n/3^3) + O(n/3^2) + O(n/3) + O(n)$
……
$T(n) = T(n/3^k) + O(n/3^{k-1}) + \cdots\cdots\cdots\cdots + O(n/3) + O(n)$
For Simplicity, assume
$n = 3^k$
$k = \log_3 n$
$T(n) <= T(1) + c\, n/3^{k-1} + \ldots\ldots + c.n/3^2 + c.n/3 + c.n$
$T(n) <= 1 + \{ c.n/3^{k-1} \cdots\cdots\cdots\cdots + c.n/3^2 + c.n/3 + c.n \}$
$T(n) <= 1 + c.n \{ 1/(1-1/3) \}$
$T(n) = 1 + 3/2\ c.n$
$T(n) = O(n)$

2. Substitution Method

Takes two steps:

- Guess the form of the solution, using unknown constants.
- Use induction to find the constants & verify the solution.

Completely dependent on making reasonable guesses

Example:

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

Solution:

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

More specifically:

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

Induction:

Assume, $T(k) <= c.k^3$ for all $k < n$

To show, $T(n) <= c.n^3$ for all $n > n_0$.

Base case: for n = 1, T(n) = 1, by definition

Inductive case: n > 1,

$$T(n) = 4T\left(\frac{n}{2}\right) + n, \qquad By\ Defination$$

$$T(n) \le 4c.\left(\frac{n}{2}\right)^3 + n, \qquad By\ induction$$

$$= \frac{c}{2}.n^3 + n$$

$$= cn^3 - \left(\frac{c}{2}.n^3 - n\right)$$

$$\le cn^3$$ For all n > 0 and c <= 2

Since,

$T(n) \le 2n^3\ \forall n > 0$ Thus, $T(n) = O(n^3)$.

Ladies and gentlemen!
Ability to guess tighter bound comes with experience, don't worry if you are doing for the first time.

b) $T(n) = 1 \qquad n = 1$
$T(n) = 4T\left(\frac{n}{2}\right) + n \quad n > 1$

Proof: Same recurrence as of (a), but we are trying tight bound now. Let's assume:

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

i.e.

$$T(n) \le cn^2\ \forall n > n_0$$

Assume,

$$T(k) = ck^2\ \forall k < n$$

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

Which is !< cn^2.

Using even tighter guess following the definition of big-O, we can assume:

$$T(n) \le cn^2 - dn\ \forall n > n_0$$

Assume,

$$T(k) = ck^2 - dk\ \forall k < n$$

And we will show: $T(n) = cn^2 - dn$

Base case: n=1, T(n) = 1 and 1 <= c-d

Inductive case: n > 1,

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$
$$\le 4[c.\left(\frac{n}{2}\right)^2 - \left(\frac{d}{2}\right)] + n$$
$$= cn^2 - 2dn + n$$
$$= cn^2 - dn - (dn - n)$$
$$\le cn^2 - dn,$$ Choosing d >=1

$T(n) \le 2n^2 - dn\ \forall n > 0$, thus $T(n) = O(n^2)$.

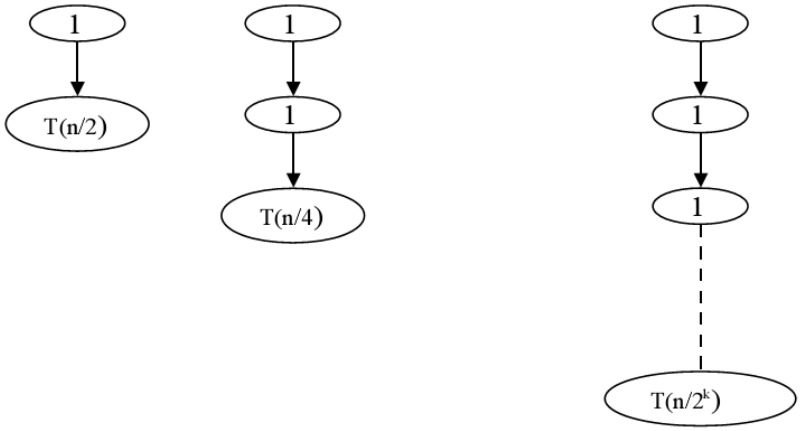
3. Recursion Tree

A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

Example I: consider a recurrence:

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

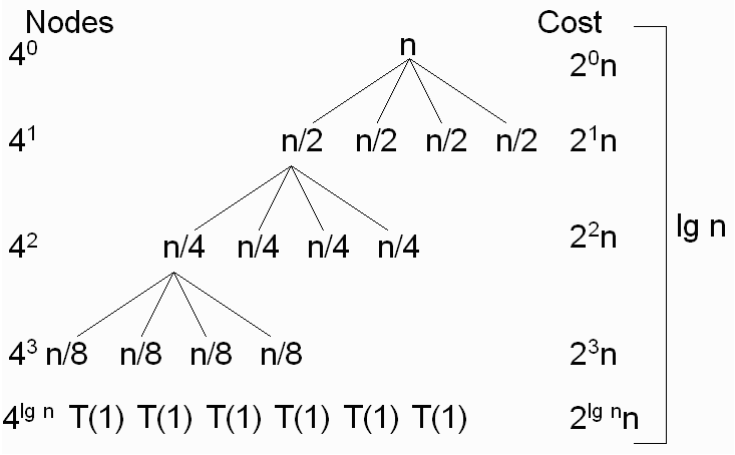
Solution: Drawing recursion tree from given relation and accumulating the cost,
Cost at each level = 1
For simplicity, assume $n = 2^k$
Then k = *log*n
Total cost = 1 + 1 + 1 +…+ upto *log*n terms = O(logn)



Example II:

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

→Bottom level has n=1 size problems or T(1)
→Assuming $n = 2^K$, so k = log n.
→Total Cost: $2^0n+2^1n+2^2n+…+2^{k-1}n+2^kn$
$= n(1 + 2 + 4 + … + 2^{k-1} + 2^k)$
$= n(2^{k+1} -1)/(2-1)$ [Geometric ser.]
$= n (2^{k+1} -1)$
$\leq n2^{k+1}$
$= 2n.2^k$
$= 2n.n$
$= O(n^2)$



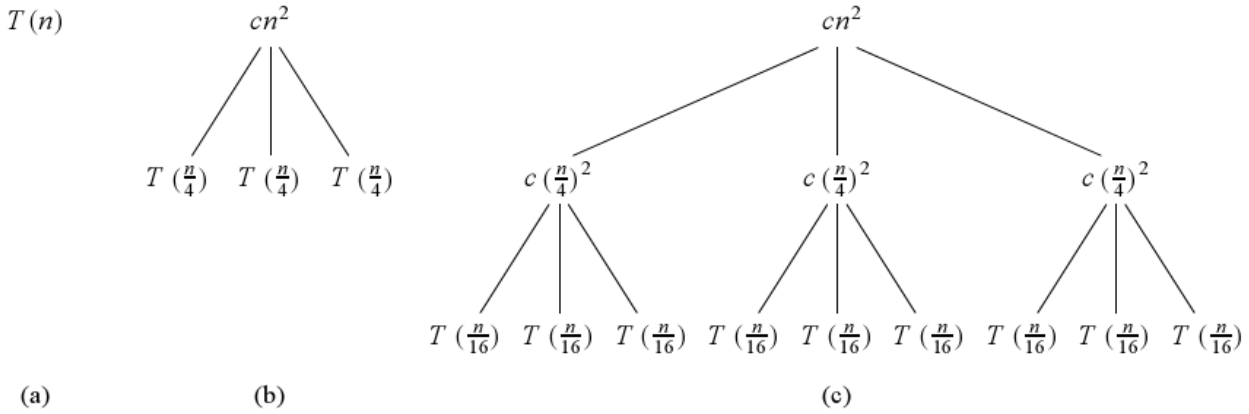
Example III:

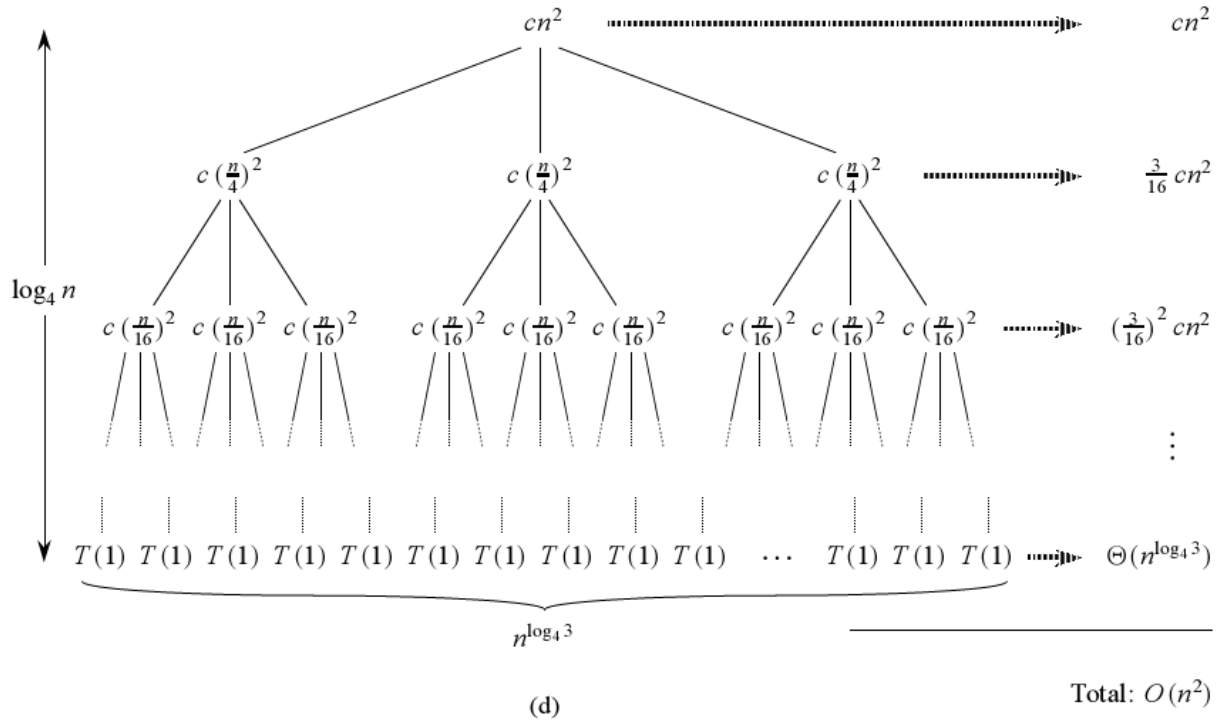
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

Solution: Focusing on upper bound and ignoring ceilings and floors (insubstantial in solving recurrences).
Now drawing recursion tree for

$$T(n) = 3T(n/4) + cn^2, \text{ for } c > 0$$

Because sub problem sizes decrease as we get further from the root, we eventually must reach a boundary condition but how far?
The subproblem size for a node at depth *i* is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has $\log_4 n + 1$ levels (0, 1, 2, . . . $\log_4 n$).





$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}
$$

Case is of decreasing geometric series, so use of

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} .$$

4. Master Method

Master method provides cookbook method for solving recurrences of the form

$T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n) > 0$

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Examples:

To use the master method, we simply determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider

$$T(n) = 9T(n/3) + n .$$

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence

$$T(n) = 3T(n/4) + n \lg n ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

The master method does not apply to the recurrence

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

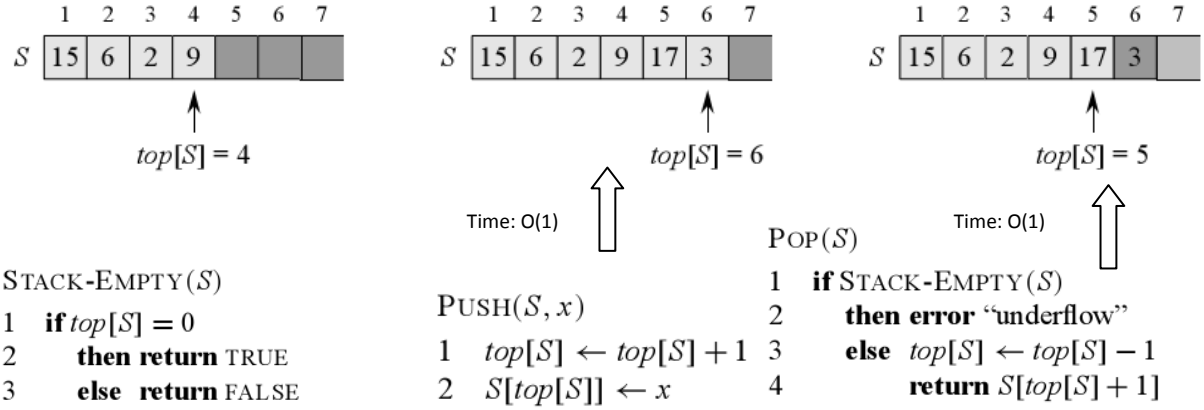
even though it has the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. It might seem that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3.

1.2 Data Structure Review

Linear Data Structures

Stacks (Array implementation)

- The element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy.



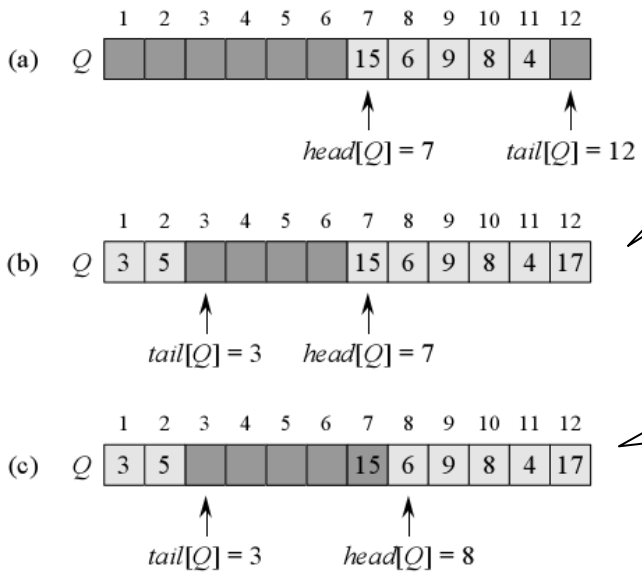

```
STACK-EMPTY(S)
1  if top[S] = 0
2     then return TRUE
3     else  return FALSE
```

```
PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

```
POP(S)
1  if STACK-EMPTY(S)
2     then error "underflow"
3     else  top[S] ← top[S] − 1
4           return S[top[S] + 1]
```

Queues (Simple array and circular implementation)

- The element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy.




```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3     then tail[Q] ← 1
4     else  tail[Q] ← tail[Q] + 1
```

Time: **O(1)**

```
DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3     then head[Q] ← 1
4     else  head[Q] ← head[Q] + 1
5  return x
```

Time: **O(1)**

Priority queue

A **priority queue** is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it so that an element with high priority can be served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue. They 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).

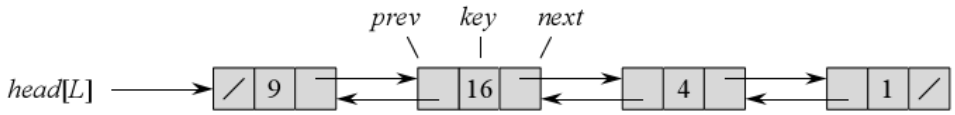
Operations on priority queue:

- 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

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	$O(1)$	$O(n)$	$O(1)$	$O(n)$

Linked List

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, though, in which the linear order is determined by the array indices, the order in a linked list is determined by a **pointer** in each object. Linked lists provide a simple, flexible representation for dynamic sets.



Operations on list

- bool isEmpty(): Return true if and only if this list is empty
- int size(): Return this list's length
- bool get(int i): Return the element with index i in this list
- bool equals(List a, List b): Return true if two lists are same (element wise)
- void clear(): Make this list empty
- void set(int i, int elem): Replace by elem, the i^{th} element 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 element 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

Array representation

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

Singly linked representation (unordered)

- 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

Operation	Array Representation	SLL Representation
get	O(1)	O(n)
Set	O(1)	O(n)
Add(int index, int data)	O(n)	O(n)
Add(int data)	O(1)	O(1)
Remove	O(n)	O(n)
Equals	$O(n^2)$	$O(n^2)$
addAll	$O(n^2)$	$O(n^2)$

Tree (Hierarchical) data structure

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

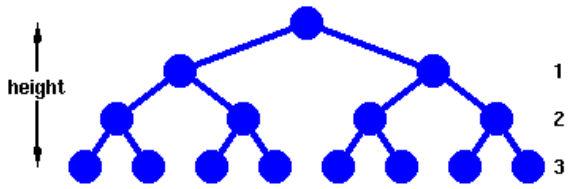
Some of the definitions

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

The first *h* levels of a full tree have $1 + d + d^2 + \ldots + d^{h-1} = (d^h - 1)/(d-1)$

Where: h (height) - number of links from the root to the deepest leaf

d = no. of children (non-leaf node)



A complete (full) tree (h = 1, 2, 3 and d = 2 (binary))

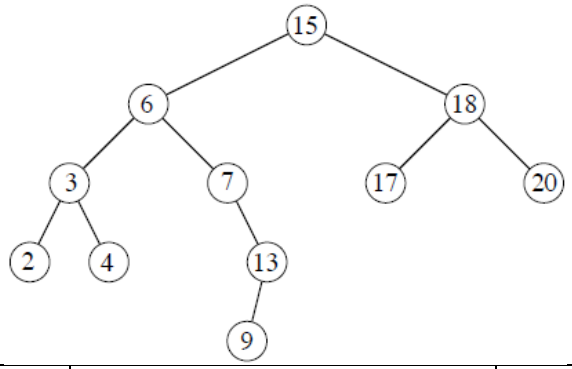
Binary Search Trees

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 h and number of nodes n is ***logn* <= *h* < *n***. for e.g. height of binary tree with 16 nodes may be anywhere between 4 and 15. If tree is height balanced then search and insertion has ***O(logn)*** run time otherwise we will get punished by ***O(n)***.

The keys in a binary search tree are always stored in such a way as to satisfy the

Binary-search-tree property:

Let *x* be a node in a binary search tree. If *y* is a node in the left subtree of *x*, then *key*[*y*] ≤ *key*[*x*]. If *y* is a node in the right subtree of *x*, then *key*[*x*] ≤ *key*[*y*].



Operation	Time Complexity (Average)	Time Complexity (Worst)
BST Search	*O*(log *n*)	*O*(*n*)
BST Insertion	*O*(log *n*)	*O*(*n*)
BST Deletion	*O*(log *n*)	*O*(*n*)
BST Walk (pre, in, post-order)	*O*(*n*)	*O*(*n*)

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	Time Complexity (Worst)
Search	O(log n)
Add	O(log n)
Remove	O(log n)

Heap

A heap can be thought of as a priority queue; the **most important** (e.g. min or max) node will always be at the top, and when removed, its replacement will be the most important. This can be useful when coding algorithms that require certain things to be processed in a complete order. For instance, a well-known algorithm for finding the shortest distance between nodes in a graph, Dijkstra's Algorithm, can be optimized by using a priority queue.

Usually, two kinds used, min and max heaps which satisfies following **heap property**: *A* be heap

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

Heap operations time complexities are listed below:

Operation	Description	Time Complexity
Add	Insert new element to heap	O(log n)
Delete	Deletes a node from heap	O(log n)
GetPrioritized	Access a top element	O(1)

Heaps can also be used to sort data. A **heap sort** is *O(nlogn)* efficiency, though it is not the fastest possible sorting algorithm. [We will cover this later, while studying heap sort]

UNIT 2: Algorithmic Paradigms

2.1 Divide and Conquer

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. The divide-and-conquer paradigm involves 3 steps at each level of the recursion:

1. **Divide** the problem into a number of subproblems.
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

Sorting problem

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 here. Sorting algorithms are often classified by:

- **Computational complexity** (worst, average and best behavior) of element comparisons in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n), with parallel sort in $O(\log^2 n)$, and bad behavior is $O(n^2)$.
- **Memory usage** (and use of other computer resources). In particular, some sorting algorithms are **in place**. Strictly, an in place sort needs only O(1) memory beyond the items being sorted; sometimes O(log n) additional memory is considered "in place".
- **Recursion**. Some algorithms are either recursive or non-recursive, while others may be both (e.g. merge sort).
- **Stability**: stable sorting algorithms maintain the relative order of records with equal keys (i.e. values).
- Whether or not they are a **comparison sort**. A comparison sort examines the data only by comparing two elements with a comparison operator.
- **General method**: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heap sort.
- **Adaptability**: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Merge Sort

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Here we deal with the subproblems, we state each sub problem as sorting a subarray **A[p..r]**. Initially, p = 1 and r = n but these values change as we recurse through subproblems.

To sort **A[*p* ..*r*]:**

Divide by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$, where q is the halfway point of $A[p..r]$.

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

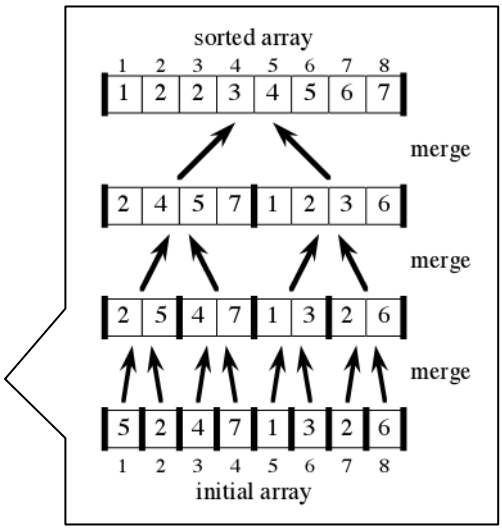
Combine by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To accomplish this step, we'll define a procedure MERGE(A, p, q, r).

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

```
MERGE-SORT(A, p, r)
if p < r                          ▷ Check for base case
  then q ← ⌊(p + r)/2⌋            ▷ Divide
       MERGE-SORT(A, p, q)        ▷ Conquer
       MERGE-SORT(A, q + 1, r)    ▷ Conquer
       MERGE(A, p, q, r)          ▷ Combine
```

Initial call: MERGE-SORT(A, 1, n)

Example: Bottom-up view for n = 8
(Heavy lines shows subarrays used in subproblems)



Now ***Merge*** Procedure:

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```

INPUT: Array *A* and indices *p*, *q*, *r* such that $p \le q \le r$ and subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] are sorted
OUTPUT: The two subarrays are merged into a single sorted subarray in *A*[*p* .. *r*].
We implement it so that it takes Θ(*n*) time, where $n = r - p + 1$, which is the number of elements being merged.

Running time: The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time. The last **for** loop makes *n* iterations, each taking constant time, for $\Theta(n)$ time.

Total time: ***Θ(n)***.

Idea behind Linear Time Merging

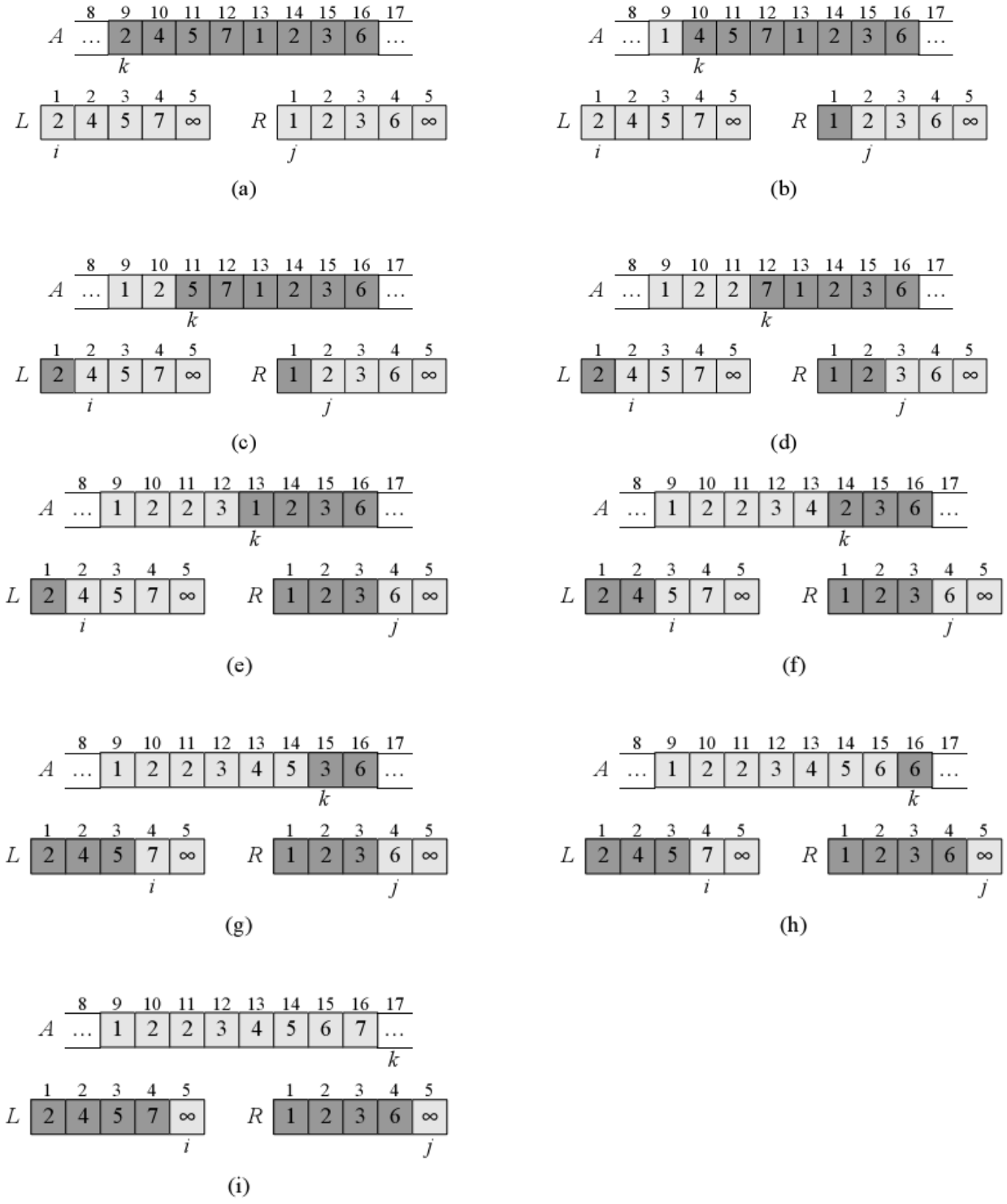
Think of two piles of cards, each pile is sorted and placed face-up on a table with the smallest cards on top. We will merge these into a single sorted pile, face-down on the table. A basic step:

- Choose the smaller of the two top cards.
- Remove it from its pile, thereby exposing a new top card.
- Place the chosen card face-down onto the output pile.
- Repeatedly perform basic steps until one input pile is empty.

- Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.

Each basic step should take constant time, since we check just the two top cards. There are at most *n* basic steps, since each basic step removes one card from the input piles, and we started with *n* cards in the input piles. Therefore, this procedure should take ***Θ(n)*** time.

Example: Call to **MERGE**(9, 12, 16)



Analyzing merge sort

- For simplicity, assume that n is a power of 2 ⇒ each divide step yields two subproblems, both of size exactly $n/2$.
- The base case occurs when $n = 1$.
- When $n \geq 2$, time for merge sort steps:
  - **Divide:** Just compute q as the average of p and r ⇒ $\Theta(1)$.
  - **Conquer:** Recursively solve 2 subproblems, each of size $n/2$ ⇒ $2T(n/2)$.
  - **Combine:** MERGE on an n-element subarray ⇒ $\Theta(n)$ time.

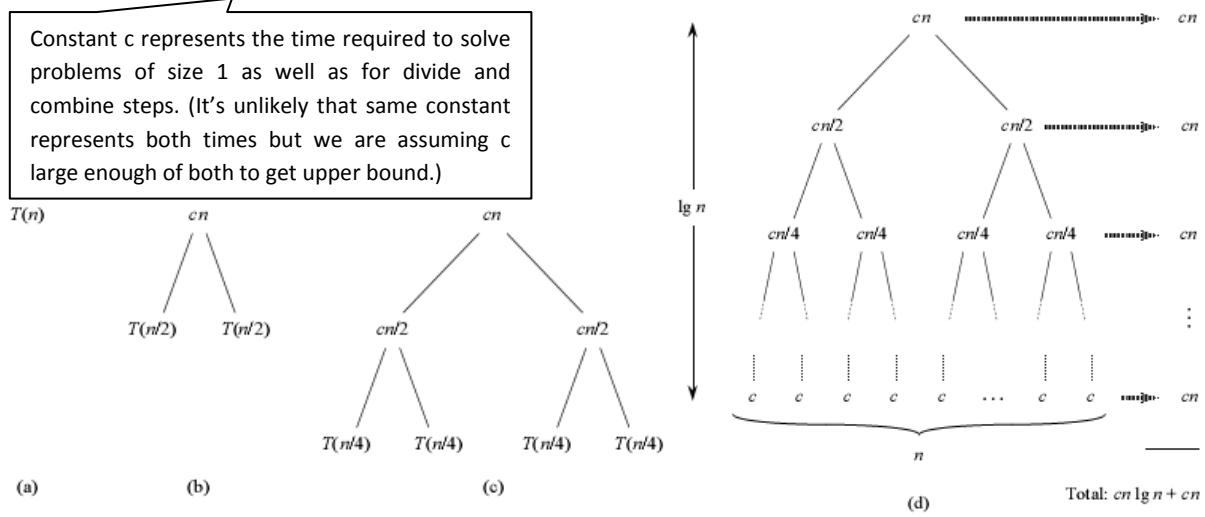
Since $\Theta(1)$ and $\Theta(n)$, summed together they give a function that is linear in n: $\Theta(n)$ ⇒ recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

Solving this recurrence:
Rewriting recurrence as:

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases}$$

Constant c represents the time required to solve problems of size 1 as well as for divide and combine steps. (It's unlikely that same constant represents both times but we are assuming c large enough of both to get upper bound.)



Total cost is sum of costs at each level. Have lg n + 1 levels, each costing cn
⇒ Total cost is T(n) = cn lg n + cn.
Ignoring low-order term of cn and constant coefficient c
⇒ $T(n) = \Theta(n \lg n)$.

Quick Sort

- Worst-case running time: $\Theta(n^2)$.
- Expected running time: $\Theta(n \lg n)$.
- Constants hidden in $\Theta(n \lg n)$ are small.
- Sorts in place.

Quicksort, like merge sort, is based on the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \,..\, r]$.
Divide: Partition the array $A[p \,..\, r]$ into two (possibly empty) subarrays $A[p \,..\, q - 1]$ and $A[q + 1 \,..\, r]$ such that each element of $A[p \,..\, q - 1]$ is $\leq A[q]$, and $A[q] \leq$ to each element of $A[q + 1 \,..\, r]$. Compute the index q as part of this partitioning procedure.
Conquer: Sort the two subarrays $A[p \,..\, q - 1]$ and $A[q + 1 \,..\, r]$ by recursive calls to quicksort.

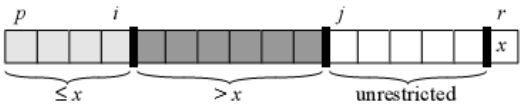
Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p \,..\, r]$ is now sorted.

```
QUICKSORT(A, p, r)
if p < r
  then q ← PARTITION(A, p, r)
       QUICKSORT(A, p, q − 1)
       QUICKSORT(A, q + 1, r)
```

Initial call: QUICKSORT*(A, 1, n)* assuming A be of length n and 1-index based.

Partitioning: Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays. Partition subarray $A[p \,..\, r]$ by the following procedure:

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
    do if A[j] ≤ x
          then i ← i + 1
               exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

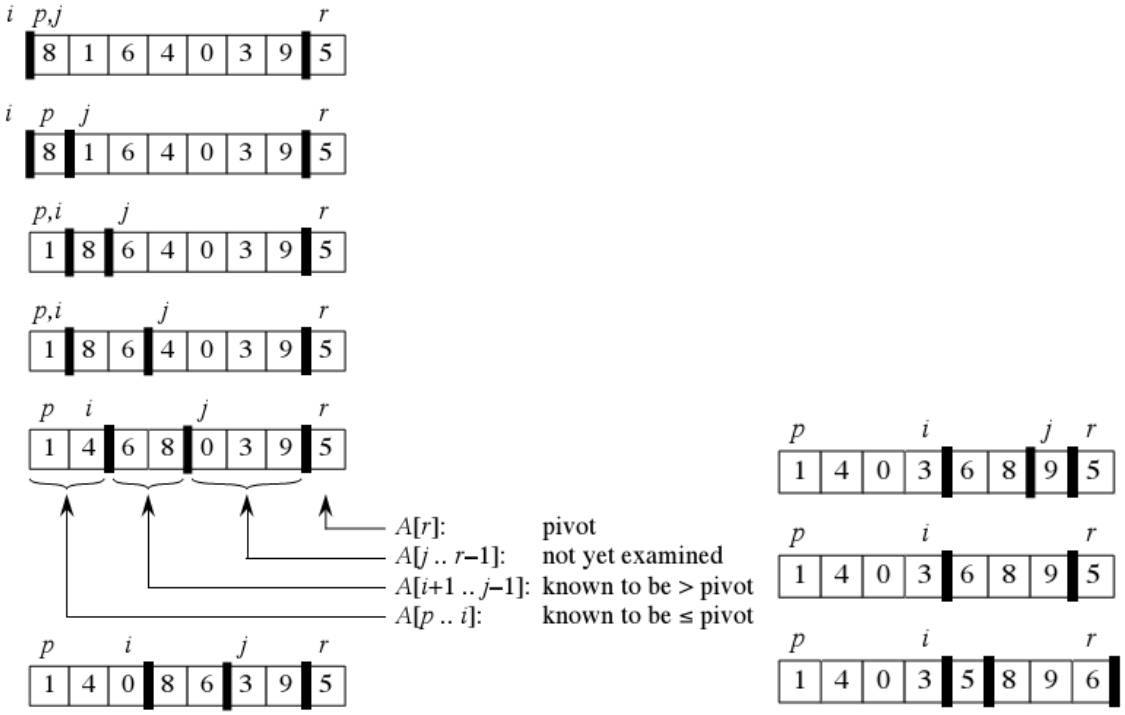


PARTITION always selects the last element $A[r]$ in the subarray $A[p \,..\, r]$ as the ***pivot*** - the element around which to partition.
As the procedure executes, the array is partitioned into four regions, some of which may be empty:
Loop invariant:

1. All entries in $A[p \,..\, i]$ are ≤ pivot.
2. All entries in $A[i + 1 \,..\, j - 1]$ are > pivot.
3. $A[r]$ = pivot.

It's not needed as part of the loop invariant, but the fourth region is $A[j \,..\, r - 1]$, whose entries have not yet been examined, and so we don't know how they compare to the pivot.

Example: On an 8-element subarray



Quick sort complexity analysis

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as merge sort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and *n* - 1 elements in the other subarray.
- Getting the recurrence:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \ . \end{aligned}$$

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

Best case

- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\leq n/2$ elements.
- Get the recurrence

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \ . \end{aligned}$$

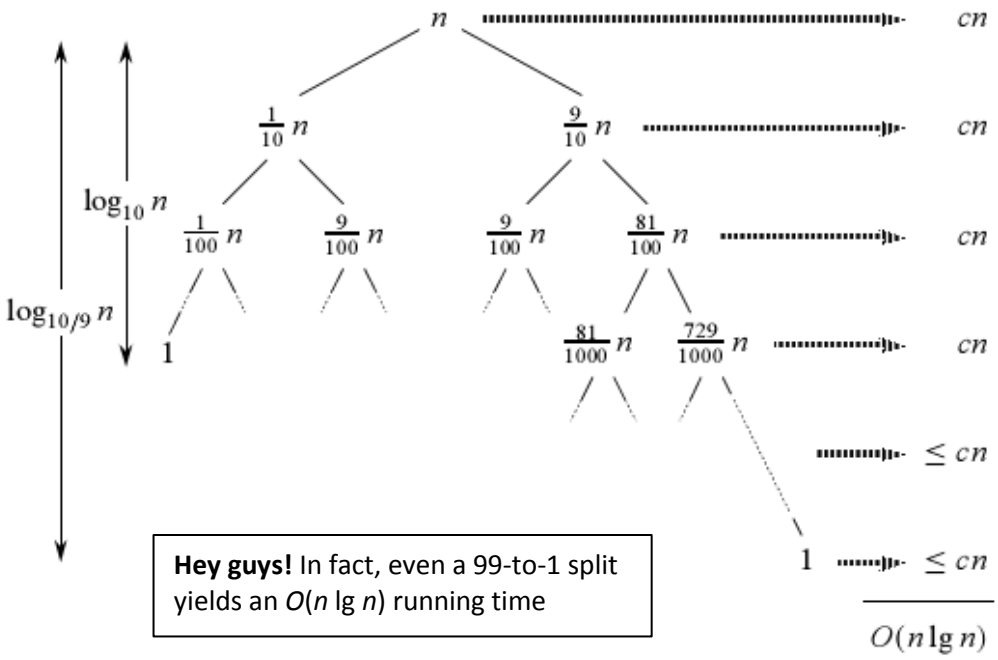
Balanced partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
- Imagine that **PARTITION** always produces a 9-to-1 split.
- Get the recurrence

$$\begin{aligned} T(n) &\leq T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n) \ . \end{aligned}$$

(See recursion tree for proof)

$T(n) \leq T(9n/10) + T(n/10) + cn$ c is constant hidden in *Θ(n).*



Hey guys! In fact, even a 99-to-1 split yields an *O*(*n* lg *n*) running time

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. Randomization is added to quick sort by **not always using *A*[*r*] as the *pivot***, but instead randomly picking (***random sampling***) an element from the subarray that is being sorted.

Minor modifications to **QUICKSORT** and **PARTITION** algorithm:

```
RANDOMIZED-PARTITION(A, p, r)
i ← RANDOM(p, r)
exchange A[r] ↔ A[i]
return PARTITION(A, p, r)
```

Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
  then q ← RANDOMIZED-PARTITION(A, p, r)
       RANDOMIZED-QUICKSORT(A, p, q − 1)
       RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Randomization of quicksort stops any specific type of array from causing worst case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized QUICKSORT, but not in RANDOMIZED-QUICKSORT.

Complexity Analysis

Worst-case

We will prove that a worst-case split at every level produces a worst-case running time of $O(n^2)$. Recurrence for the worst-case running time of QUICKSORT:

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n) .$$

Because PARTITION produces two subproblems, totaling size n - 1, q ranges from 0 to n - 1. Using substitution method to solve this recurrence:

Guess: $T(n) \le cn^2$, for some c.

Substituting our guess into the above recurrence:

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) + \Theta(n) \end{aligned}$$

The maximum value of $(q^2 + (n - q - 1)^2)$ occurs when q is either 0 or n - 1. (Second derivative with respect to q is positive.) This means that

$$\begin{aligned} \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) &\le (n - 1)^2 \\ &= n^2 - 2n + 1 \end{aligned}$$

Therefore,

$$\begin{aligned} T(n) &\le cn^2 - c(2n - 1) + \Theta(n) \\ &\le cn^2 \qquad \text{if } c(2n - 1) \ge \Theta(n) . \end{aligned}$$

Pick c so that $c(2n - 1)$ dominates $\Theta(n)$.

Therefore, the worst-case running time of quicksort is $\boldsymbol{O(n^2)}$.

Average case

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

Now we give recurrence relation for the algorithm as

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

$T(k)$ and $T(n-k)$ are repeated two times i.e. there are 2 T(1)'s, 2 T(2)'s … 2 T(n-1)'s while running through $1 \le k \le n-1$.

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

$nT(n) = 2\sum_{k=1}^{n-1} T(k) + O(n^2)$ $[Since, n * O(n) = O(n^2)]$
Similarly,
$(n-1)T(n-1) = 2\sum_{k=1}^{n-1} T(k) + O(n-1)^2$
Now,
$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$
$T(n) - (n+1)T(n-1) = 2n - 1$

Dividing both sides by $n(n+1)$,

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

Let $A_n = \frac{T(n)}{n+1}$
Then, above equation can be written as:

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

$A_n = \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} \le \sum_{i=1}^{n} \frac{2i}{i(i+1)} = 2\sum_{i=1}^{n} \frac{1}{(i+1)} \le 2\log n$ [Since, Harmonic series sum]
Substituting the value of A_n,

$\frac{T(n)}{n+1} \le 2\log n$
$T(n) = 2(n+1)\log n = 2n\log n + 2\log n = c.n\log n - (c-2)\log n + 2\log n$
Taking value of constant *c* such that term $(c-2)\log n$ dominates $2\log n$, then we can write,
$T(n) \le c.\text{nlog}\, n$
$\boldsymbol{T(n) = O(\text{nlog}\, n)}$

Heap Sort

- *O(n* lg *n)* worst case - like merge sort.
- Sorts in place - like insertion sort.
- Combines the best of both algorithms.

To understand heapsort, we'll cover heaps and heap operations, and then we'll take a look at priority queues.

Heaps

Heap *A* is a nearly complete binary tree.

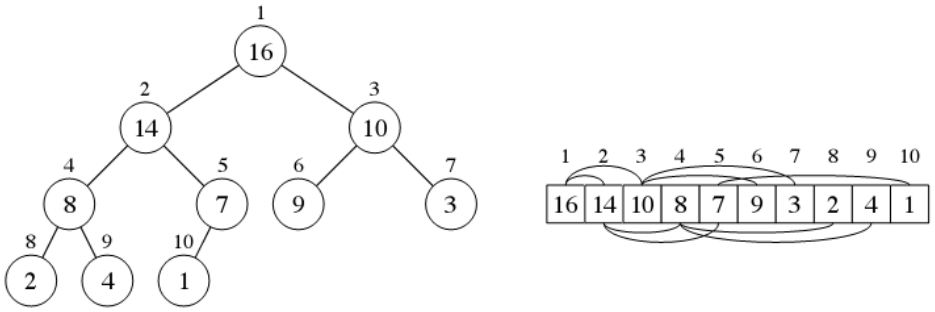
- ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.

- ***Height*** of heap = height of root = $\theta(\log n)$
- A heap can be stored as an array *A*:
  - Root of tree is *A*[1].
  - Parent of $A[i] = A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$.
  - Computing is fast with binary representation implementation.

Heap property:

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

Example: of a max-heap.



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 (For heapsort, max-heaps are normally used): MAX-HEAPIFY
- Create a max-heap from an unordered array : BUILD-MAX-HEAP
- Sort an array in place : HEAPSORT
- Priority queues

MAX-HEAPIFY (Maintaining heap-property)

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, *A*[*i*] may be smaller than its children.
- Assume left and right subtrees of *i* are max-heaps.
- After MAX-HEAPIFY, subtree rooted at *i* is a max-heap.

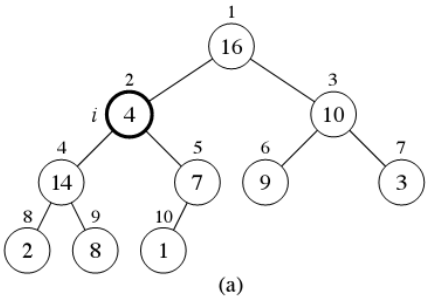
```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
  then largest ← l
  else largest ← i
if r ≤ n and A[r] > A[largest]
  then largest ← r
if largest ≠ i
  then exchange A[i] ↔ A[largest]
       MAX-HEAPIFY(A, largest, n)
```

- n = heap-size[A]
- **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 = $O(h) = O(\log n)$

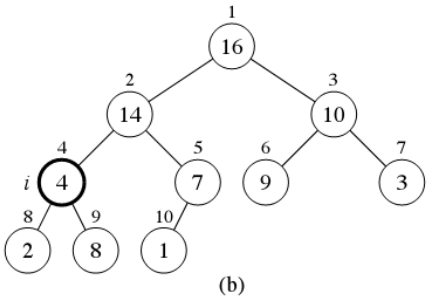
The way MAX-HEAPIFY works:

- Compare *A*[*i*], *A*[LEFT*(i)*], and *A*[RIGHT*(i)*].
- If necessary, swap *A*[*i*] with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at *i* is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

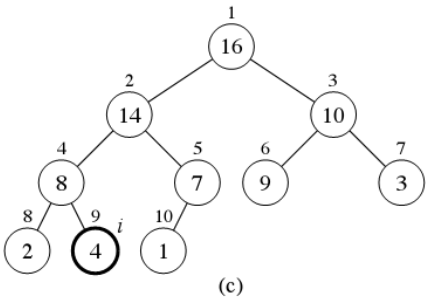
Run MAX-HEAPIFY on the following heap example:



(a)



(b)



(c)

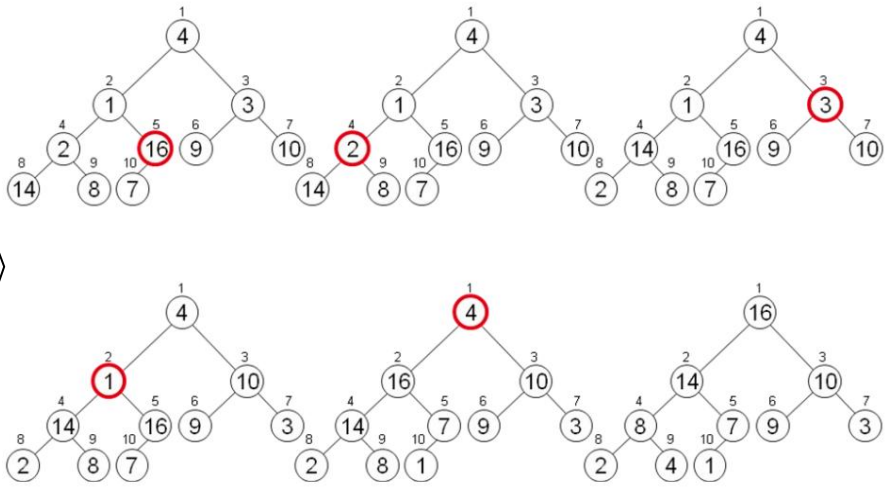
- Node 2 violates the max-heap property.
- Compare node 2 with its children, and then swap it with the larger of the two children.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Building a heap

```
BUILD-MAX-HEAP(A, n)
for i ← ⌊n/2⌋ downto 1
    do MAX-HEAPIFY(A, i, n)
```

Example: Building a max-heap from the following unsorted array.

- *i* starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.



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

Analysis: Build-Max-Heap

Simple bound: Loop executes *O(n)* times, each call to heapify takes $O(log\ n)$. Thus time complexity $\Rightarrow O(n\ log\ n)$.

Tighter analysis: Above bound is not asymptotically tight. Let's try to tighten it with the observation: *Time to run heapify is linear in the height of the node it's run on, and most nodes have small heights*.

Heapify takes $O(h) \Rightarrow$ Cost of heapify on node *i* is proportional to the height of node *i* in the tree.
Thus,

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

Where

$h_i = h - i$: Height of the heap rooted at level *i*.

$n_i = 2^i$: number of nodes at level *i*.

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

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

Let, $k = h - i$

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

$\Rightarrow T(n) \leq n \sum_{k=0}^{\infty} \frac{k}{2^k} = n \sum_{k=0}^{\infty} k(\frac{1}{2})^k\ [Since, n = 2^h\ and\ changing\ bound\ to\ \infty]$

We know that, for infinite geometric sum with $|x| < 1$, we have

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$Differentiating\ both\ sides\ w.r.t.x, we\ get$$

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

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

Now using this summation for $T(n)$ where $x = 1/2$, we get

$$T(n) = n.\left(\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right) = 2n$$

$$\boldsymbol{T(n) = O(n)}$$

Heap Sort Algorithm

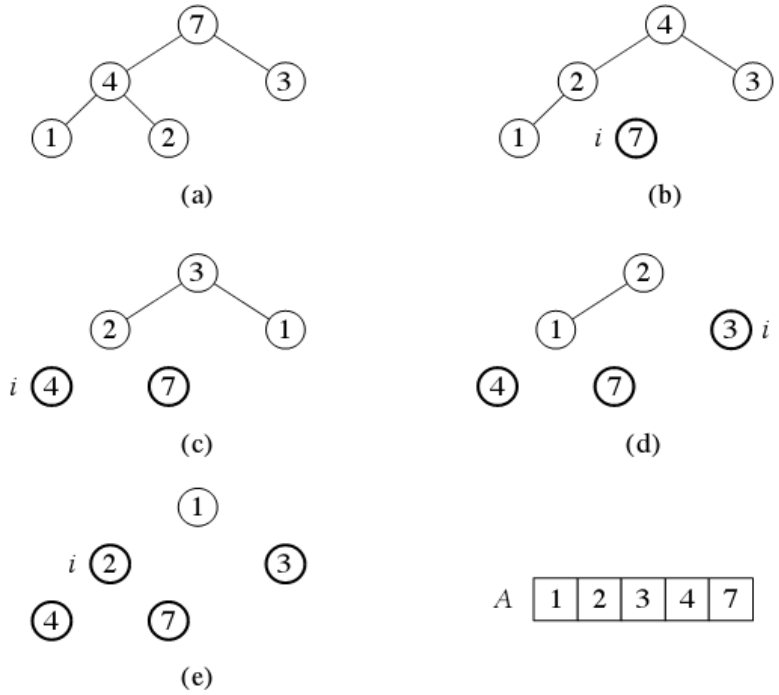
Given an input array, the heapsort algorithm acts as follows:

- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i ← n downto 2
    do exchange A[1] ↔ A[i]
       MAX-HEAPIFY(A, 1, i − 1)
```

Though heapsort is a great algorithm, a well-implemented quicksort usually beats it in practice.

Example: Sort an example heap on the board. [Nodes with heavy outline are no longer in the heap.]



Analysis

· BUILD-MAX-HEAP: *O(n)*

· **for** loop: *n* - 1 times

· exchange elements: *O*(1)

· MAX-HEAPIFY: *O*(log *n)*
Total time: ***O(n* log *n)***

Summary: Sorting algorithms

	Time					
Sort	Average	Best	Worst	Space	Stability	Remarks
Bubble sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Always use a modified bubble sort
Modified Bubble sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	Stops after reaching a sorted array
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quicksort	O(n*log(n))	O(n*log(n))	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

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" index.

Algorithm

```
Linear-Search(A, n, key)
  For i ←1 to n
    if A[i] == key
          return i
  return -1 // Unsuccessful search
```

Time Complexity: ***O*(*n*)**

Binary Search

Binary search or **half-interval search** algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order.

Algorithm:

```
int binary_search(int A[], int key, int min, int max)
{
  if (max < min)
    // set is empty, so return value showing not found
    return -1; //KEY_NOT_FOUND
  else
    {
      // calculate midpoint to cut set in half
      int mid = (min + max) / 2;

      // three-way comparison
      if (A[mid] == key)
        // key has been found
       return mid;
      else if (A[mid] < key)
        // key is in upper subset
        return binary_search(A, key, mid+1, max);
```

```
        else
          // key is in lower subset
          return binary_search(A, key, min, mid-1);
      }
}
```

Example: If A (array) be:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Call: `binary_search(A, 73, 0, 14)`
Key = 73

Found: return 9

Analysis:

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

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$
$$= O(\log n)$$

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

Selection

- *ith order statistic is the i th smallest element of a set of n elements.*
- *The minimum is the first order statistic (i = 1).*
- *The maximum is the nth order statistic (i = n).*
- *A median is the "halfway point" of the set.*
- *When n is odd, the median is unique, at i = (n + 1)/2.*
- *When n is even, there are two medians:*
  - *The lower median, at i = n/2, and*
  - *The upper median, at i = n/2 + 1.*
  - *We mean lower median when we use the phrase "the media"*

Minimum and Maximum (Min-Max)

Finding minimum (or maximum or both simultaneously) from the set *A* of size *n* can be done with upper bound of $O(n)$ using:

1. **Iterative Procedure**

```
MinMax(A, n)
        min ← A[1]
        max ← A[1]
        for i←2 to n
                if(min > A[i])
                        min ← A[i]
                if(max < A[i])
                        max ← A[i]
        return {min, max}
```

Examine each element of the set in turn and keep track of the smallest (or largest) element seen so far.
Time Complexity: Total of $2 * (n - 1)$ comparisons
: $\boldsymbol{O(n)}$

2. Divide And Conquer Procedure
   Idea: 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(A, l, r) // A: Input array, l: left (lower) index, r: right (upper) index
{
    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 problem
            mid = (l + r)/2; //integer division

            //solve the subproblems
            {min,max} = MinMax(A, l, mid);
            {min1,max1}= MinMax(A, mid +1, r);

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

Analysis:
Recurrence relation for **MinMax** algorithm in terms of number of comparisons is:
$T(n) = 2T(n/2) + 1, \quad if\ n > 2$
$T(n) = 1, \quad if\ n \leq 2$
Solving the recurrence by using master method (case 1), $T(n) = O(n)$.

Selection problem

Input: A set *A* of *n* distinct numbers and a number , with $1 \leq i \leq n$.
Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in *A*. In other words, the *i*th smallest element of *A*.
The selection problem can be solved in $O(n \lg n)$ time.

- Sort the numbers using an $O(n \lg n)$-time algorithm, such as heapsort or merge sort.
- Then return the *i* th element in the sorted array.

There are faster algorithms, however.

- First, we'll look at general quadratic general selection algorithm.
- Then, we'll look at a simple general selection algorithm with a time bound of *O(n)* in the average case.

- Finally, we'll look at a more complicated general selection algorithm with a time bound of *O(n)* in the worst case.

1. General Selection Algorithm using partial sorting (Non-linear)

```
SELECT(A, k, n)
     for i ← 1 to k
         minIndex = i
         minValue = A[i]
         for j ← i+1 to n
             if A[j] < minValue
                 minIndex = j
                 minValue = A[j]
         swap A[i] and A[minIndex]
     return A[k]
```

We simply find the minimum value and move it to the beginning, repeating on the remaining list until we have accumulated *k* elements, and then return the *k*th element.

Analysis: A partial selection sort yields a simple selection algorithm which takes $O(kn)$ time. This is asymptotically inefficient, but can be sufficiently efficient if *k* is small, and is easy to implement.

2. Selection in Expexted Linear Time

Selection of the i th smallest element of the array A can be done in $\theta(n)$ time. The function RANDOMIZED-SELECT utilizes divide and conquer approach and uses RANDOMIZED-PARTITION from the quicksort algorithm. RANDOMIZED-SELECT differs from quicksort because it recurses on one side of the partition only.

```
RANDOMIZED-SELECT(A, p, r, i)
if p = r
  then return A[p]
q ← RANDOMIZED-PARTITION(A, p, r)
k ← q − p + 1
if i = k          ▷ pivot value is the answer
  then return A[q]
elseif i < k
  then return RANDOMIZED-SELECT(A, p, q − 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

After the call to RANDOMIZED-PARTITION, the array is partitioned into two subarrays

A[*p* . . *q* - 1] and *A*[*q* + 1 . . *r*], along with a ***pivot*** element *A*[*q*].

- The elements of subarray *A*[*p* . . *q* − 1] are all ≤ *A*[*q*].
- The elements of subarray *A*[*q* + 1 . . *r*] are all > *A*[*q*].
- The pivot element is the *k*th element of the subarray *A*[*p* . . *r*], where *k* = *q* − *p* + 1.
- If the pivot element is the *i* th smallest element (i.e., *i* = *k*), return *A*[*q*].
- Otherwise, recurse on the subarray containing the *i* th smallest element.
    - If *i* < *k*, this subarray is *A*[*p* . . *q* −1], and we want the *i* th smallest element.
    - If *i* > *k*, this subarray is *A*[*q* + 1 . . *r*] and, since there are *k* elements in *A*[*p* . . *r*] that precede *A*[*q* + 1 . . *r*], we want the *(i* − *k)*th smallest element of this subarray.

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:

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

Where, $\max(k, n-k) = k, if\ k \geq \lceil n/2 \rceil$

And $\max(k, n-k) = n-k, otherwise.$

Observe that every $T(k)$ or $T(n\ -\ k)$ will repeat twice for both odd and even value of n. One time from 1 to $\lceil n/2 \rceil$ and second time for $\lceil n/2 \rceil$ to $n-1$, so we can write

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

Using substitution method to solve this recurrence,
Guess $T(n)\ =\ O(n)$
To show: $T(n) \leq cn$
Assume: $T(k) \leq ck, \forall k > 1$
Substituting on the relation:

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

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

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

$$T(n) = cn - c - \frac{cn}{4} + \frac{c}{2} + O(n) = cn - \frac{cn}{4} - \frac{c}{2} + O(n) \leq cn, chossing\ c: -\frac{cn}{4} - \frac{c}{2} + O(n) \leq 0$$

$$\boldsymbol{T(n) = O(n)}$$

3. Selection is worst-case linear time

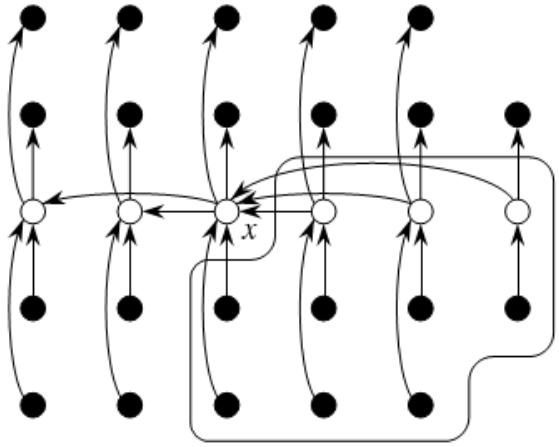
We can find the *i* th smallest element in *O(n)* time *in the worst case*.

Idea: Guarantee a good split when the array is partitioned.

Will use the deterministic procedure PARTITION, but with a small modification. Instead of assuming that the last element of the subarray is the pivot, the modified PARTITION procedure is told which element to use as the pivot.

SELECT works on an array of *n* > 1 elements. It executes the following steps:

1. Divide the *n* elements into groups of 5. Get $\lceil n/5 \rceil$ groups: $\lfloor n/5 \rfloor$ groups with exactly 5 elements and, if 5 does not divide *n*, one group with the remaining *n* mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups:
    a. Run insertion sort on each group. Takes *O(1)* time per group since each group has ≤5 elements.
    b. Then just pick the median from each group, in *O(1)* time.
3. Find the median *x* of the $\lceil n/5 \rceil$ medians by a recursive call to SELECT.
4. Let *x* be the *k*th element of the array after partitioning, so that there are *k* -1 elements on the low side of the partition and *n* - *k* elements on the high side.
5. Now there are three possibilities:
    a. If *i* = *k*, just return *x*.
    b. If *i* < *k*, return the *i* th smallest element on the low side by making a recursive call to SELECT.
    c. If *i* > *k*, return the *(i - k)*th smallest element on the high side by calling SELECT recursively



Each white circle is the median of a group, as found in step 2. Arrows go from larger elements to smaller elements, based on what we know after step 4. Elements in the region on the lower right are known to be greater than *x*.

Analysis:

We first determine a lower bound on the number of elements that are greater than the partitioning element *x*. At least half of the medians found in step 2 are greater than1 the median-of-medians *x*. Thus, at least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are greater than *x*, except for the one group that has fewer than 5 elements if 5 does not divide *n* exactly, and the one group containing *x* itself. Discounting these two groups, it follows that the number of elements greater than *x* are at least:

$$3\left(\left\lceil\left(\frac{1}{2}\right)\left\lceil\frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Similarly, the number of elements that are less than *x* is at least $\frac{3n}{10} - 6$. Thus, in the worst case, SELECT is called recursively on at most $\frac{7n}{10} + 6$ elements in step 5.

Now,

STEP 1, 2, 4: $O(n)$ each.

STEP 3: $T(\lceil n/5 \rceil)$

STEP 5: $T\left(\frac{7n}{10} + 6\right)$

We can therefore obtain the recurrence:

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140\,, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 140\,. \end{cases}$$

I'll clarify later why this magic constant 140 is used.

We show that the running time is linear by substitution. More specifically, we will show that $T(n) \leq cn$ for some suitably large constant *c* and all $n > 0$.

Assume $T(n) \leq cn$ for some suitably large constant *c* and all $n \leq 140$; Substituting this inductive hypothesis into the right-hand side of the recurrence yields:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an)\,, \end{aligned}$$

which is at most cn if

$$-cn/10 + 7c + an \leq 0\,.$$

Solving this inequality for c,

$$\begin{aligned} -cn + 70c + 10an &\leq 0 \\ c(n - 70) - 10an &\geq 0 \\ c \geq 10a\left(\frac{n}{n - 70}\right) &\forall n > 70 \end{aligned}$$

Assuming $n \geq 140$ gives us $c \geq 20a$. We could use any constant >70 but that will give us fractional multiplier for *a* (now 20).

Hence, $\boldsymbol{T(n) = O(n) \quad \forall c \geq 20a}$

Matrix Multiplication

If $A = (a_{ij})$ and $B = (b_{ij})$ are square nxn matrices, then in the product matrix $C = A.B$, we define a entry c_{ij} by:

$$cij = \sum_{k=1}^{n} a_{ik}.b_{kj}$$

Above operation can be done in three ways:

1. **Iterative approach**

```
SQUARE-MATRIX-MULTIPLY(A, B)
    n = A.rows
    let C be a new nxn matrix
    for i = 1 to n
        for j = 1 to n
            c_ij = 0
            for k = 1 to n
                c_ij = c_ij + a_ik.b_kj
    return C
```

Time Complexity: $O(n^3)$

2. **Divide-and-Conquer approach**

Divide square (nxn) matrices A, B and C into four n/2 x n/2 matrices as:

$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

Then, C = A.B

$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} . \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

$i.e$

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

SQUARE-MATRIX-MULTIPLY-RECURSIVE (A, B)

```
    n = A.rows
    let C be a new nxn matrix
    if n == 1
        c_11 = a_11.b_11
    else
        Partition A, B, C
        C_11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_11, B_11)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_12, B_12)
        C_12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_11, B_12)
```

+ SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_{12}, B_{22})

C_{21} = SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_{21}, B_{11})
+ SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_{22}, B_{21})

C_{22} = SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_{21}, B_{12})
+ SQUARE-MATRIX-MULTIPLY-RECURSIVE (A_{22}, B_{22})

return C

Analysis:

$$T(n) = \begin{cases} \theta(1) & n = 1 \\ 8T\left(\frac{n}{2}\right) + \theta(n^2) & n > 1 \end{cases} [\theta(n^2)\ for\ matrix\ additions]$$

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

3. Strassen's Method

Instead of performing 8 recursive multiplications of n/2 x n/2 matrices, it performs only 7. Cost of eliminating one matrix multiplication will be constant number of several new additions.

Algorithm:

STEP 1: Divide input matrices A, B and output matrix C into n/2 x n/2 sub matrices. $[\theta(1)]$

STEP 2: Create 10 matrices S_1, S_2 … S_{10} each of which is n/2 x n/2 and is sum or difference of two matrices created in STEP 1. $[\theta(n^2)$: $adding\ or\ subtracting\ \frac{n}{2}\ x\ \frac{n}{2}\ matrices\ 10\ times]$

$$S_1 = B_{12} - B_{22}$$
$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}$$

STEP 3: Recursively compute 7 matrix products P_1, P_2… P_7 using sub matrices created STEP1 and STEP2. Each P_i is n/2 x n/2. $\left[7T\left(\frac{n}{2}\right)\text{: } 7\ Recursive\ calls\ with\ input\ dimension\ halved\left(\frac{n}{2}\right)\right]$

$$P_1 = A_{11}.S_1$$
$$P_2 = S_2.B_{22}$$
$$P_3 = S_3.B_{11}$$
$$P_4 = A_{22}.S_4$$
$$P_5 = S_5.S_6$$
$$P_6 = S_7.S_8$$
$$P_7 = S_9.S_{10}$$

STEP 4: Compute result matrix C by adding/subtracting various P_i's. $[\theta(n^2)]$

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

Analysis:

Running time $T(n)$ of Strassen's algorithm:

$$T(n) = \begin{cases} \theta(1) & n = 1 \\ 7T\left(\frac{n}{2}\right) + \theta(n^2) & n > 1 \end{cases}$$

$$Solving,$$

$$T(n) = \theta\left(n^{\log_2 7}\right) = n^{2.81}$$

2.2 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. Dynamic programming is applicable when the subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time the subsubproblem is encountered.

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming) Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find ***a*** solution with ***the*** optimal value.
  - Minimization or maximization. (We'll see both)

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. (There is also top-down alternative, called memoization)

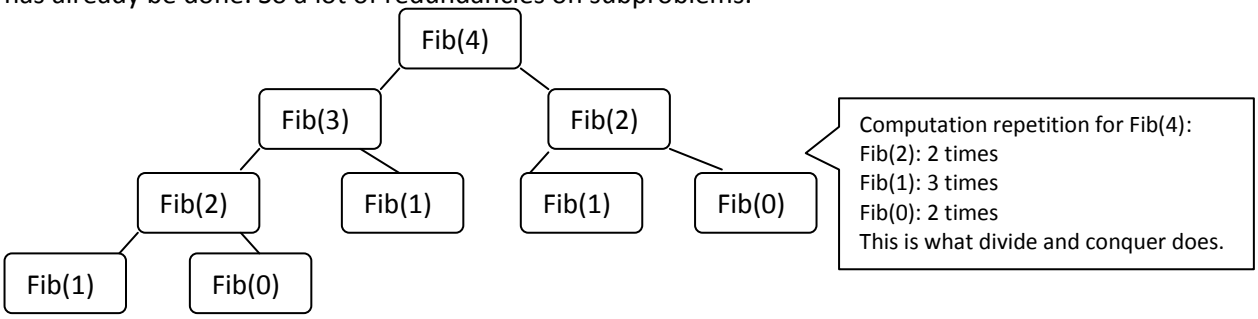
The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the formulation of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to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

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

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. (Like n^2, n^3… not log*n*, e^n)

1. Fibonacci Numbers

In **recursive version** of an algorithm for finding Fibonacci number, we can notice that for each calculation of larger number we have to calculate two previous smaller numbers regardless of the computation that has already be done. So a lot of redundancies on subproblems.



DP algorithm for Fibonacci:

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

This is the iterative approach you have already seen and unknowingly using DP ☺ , funny, isn't it?
In this case table structure is simply a 1-D array, which indexed to reuse its subproblem elements.

Analysis

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 sub-problem that is already solved to solve the problem of higher size.

2. 0/1 Knapsack Problem

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is w_i and it worth v_i. We can either pick item or leave it 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 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 and returns a table c as output.
Let us assume that the table-cell c[i,w] is the value of solution for items 1 to i and maximum weight w (small w). Then we can define recurrence relation for 0/1 knapsack problem as:

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

Algorithm:

```
DP_01Knapsack(W, n , v[], w[])
{
    Create new 2-d array (table) c[n,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

Overall running time of the algorithm, $T(n) = \boldsymbol{O(nW)}$.

Example:

Let, number of items n = 7, there weights be w[] = {2, 3, 6, 5, 4, 1} and corresponding values be v[] = {13, 14, 24, 25, 18, 10}. If knapsack can hold maximum weight W = 8, we have to find items that gives us maximum benefit.

Given: w[] = {2, 3, 6, 5, 4, 1}
v[] = {13, 14, 24, 25, 18, 10}

To fill up table, we are simply following the algorithm. I have already told you guys, make up your own rules while tracing. I have told mine in class, if you remember☺.

i \ w	0	1	2	3	4	5	6	7	8←W
0	0	0	0	0	0	0	0	0	0
1	0	0	13	13	13	13	13	13	13
2	0	0	13	14	14	27	27	27	27
3	0	0	13	14	14	27	27	27	37
4	0	0	13	14	14	27	27	38	39
5	0	0	13	14	18	27	31	38	39
n→6	0	10	13	23	24	28	37	41	48

What does each cell (c[i, w]) mean?
ANS: They are the optimal solutions for subproblems keeping in mind the optimal substructure viz. c[3, 5]: if there are 3 items and bag can contain only 5, then maximum benefit we can achieve is 27.

To find **items to keep**, use this algorithm:

```
Start with i = n and j = W
    If c[i, j] != c[i-1, j]
            Take i^th item
            i = i – 1
            j = j – w[i]
    else
            i = i – 1
```

c[6, 8] = 48, which is the maximum benefit that we can get.

3. Matrix-Chain-Multiplication

Statement: Given a chain $(A_1, A_2, \ldots, A_n)$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Hey! Algorithm does not perform multiplications, it just determine the best order.

Since Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five distinct ways:

$$(A_1(A_2(A_3A_4))),$$
$$(A_1((A_2A_3)A_4)),$$
$$((A_1A_2)(A_3A_4)),$$
$$((A_1(A_2A_3))A_4),$$
$$(((A_1A_2)A_3)A_4).$$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

Illustration:

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10 × 100, 100 × 5, and 5 × 50, respectively. If we multiply according to the parenthesization$((A_1A_2)A_3)$, we perform 10 · 100 · 5 = 5000 (A_1A_2), plus another 10 · 5 · 50 = 2500 scalar multiplications to multiply this matrix by *A*3, for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization$(A_1(A_2A_3))$, we perform 100 · 5 · 50 = 25,000 scalar multiplications to compute the 100 × 50 matrix product *A*2 *A*3, plus another 10 · 100 · 50 = 50,000 scalar multiplications to multiply *A*1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is **10 times faster**.

Algorithm:

Let $A_{i\ldots j}$ denote result of multiplying matrices through i to$j$. $A_{i\ldots j}$ is $p_{i-1} \times p_j$matrix. Now we split the product between $A_{i\ldots k}$ and $A_{k+1\ldots j}$for some integer$k$: $i \le k < j$. So total cost is sum of computing $A_{i\ldots k}$, cost of computing $A_{k+1\ldots j}$ and cost of multiplying $A_{i\ldots k}$ and $A_{k+1\ldots j}$.

Recursive definition: Let $m[i, j]$ denotes minimum number of scalar multiplications needed to compute$A_{i\ldots j}$.

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

MATRIX-CHAIN-ORDER (p) // p: array of matrix dimensions, {p_0, p_1… p_n}

```
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n                ▷ l is the chain length.
    do for i ← 1 to n − l + 1
            do j ← i + l − 1
               m[i, j] ← ∞
               for k ← i to j − 1
                   do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                      if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

The algorithm first computes *m*[*i*, *i*] ← 0 for *i* = 1, 2, . . . , *n* (the minimum costs for chains of length 1). It then uses recurrence above to compute *m*[*i*, *i* + 1] for *i* = 1, 2, . . . , *n* – 1.

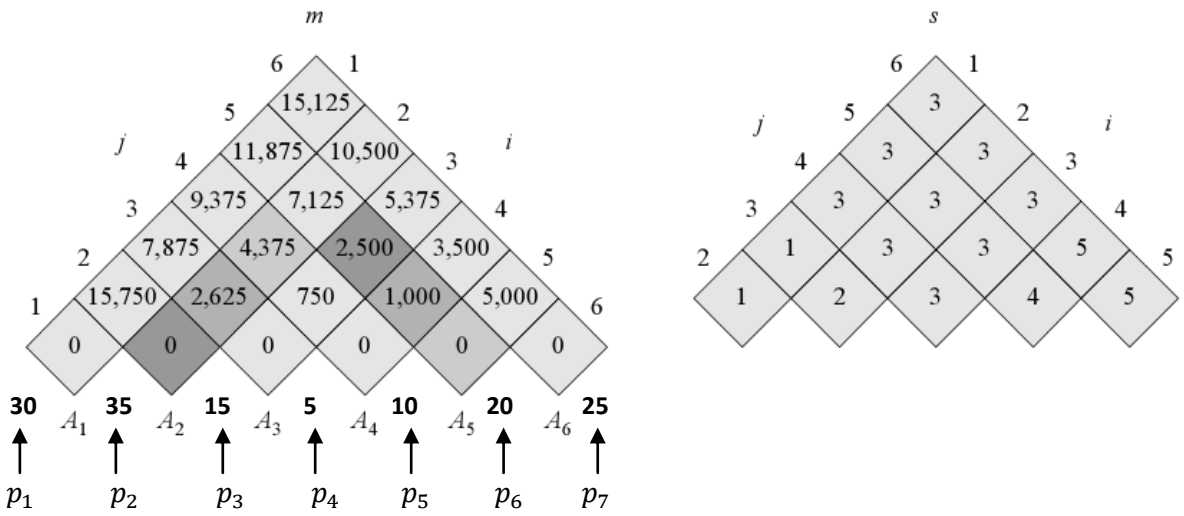
And to fully parenthesize matrix chain we use following recursive function:

PRINT-OPTIMAL-PARENS (s, i, j)

```
if i = j
   then print "A"_i
   else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
        print ")"
```

Example: let matrices and their dimensions be A_1 A_2 A_3 A_4 A_5 A_6

30X35 35X15 15X5 5X10 10X20 20X25



Tables are rotated so that a diagonal lies horizontally. Each table cells are calculated as:

$m[1,2] = \sum_{1<k\leq 2} \min\{m[1,k] + m[k+1,2] + p_{1-1}p_k p_2\}$

$= \min\{m[1,1] + m[2,2] + p_0 p_1 p_2\}$

$= \min\{0 + 0 + 30 \times 35 \times 15\}$

$= 15750$ (also s[1, 2] = 1 (k for which we get minimum value))

…

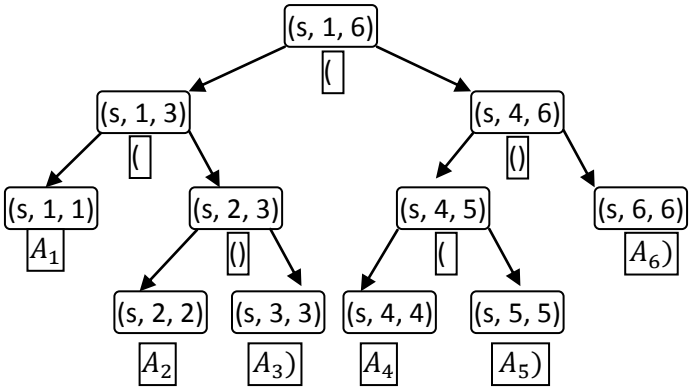
Others are calculated similarly, viz.

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= 7125\,.$$

….

Now, we use *s* table to fully parenthesize matrix chain:



Recursion tree is for **PRINT-OPTIMAL-PARENS (s, i, j)** where nodes has only argument set and two nodes for two recursive calls.

Tracing Rule:

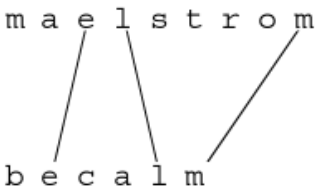
- Each left node (including root) where $i \neq j$prints "(" and prints a Matrix A_i if $i = j$.
- Each right node where $i \neq j$prints "()" and if $i = j$, prints "A_i)".

OUTPUT: $((\boldsymbol{A_1}(\boldsymbol{A_2}\boldsymbol{A_3}))((\boldsymbol{A_4}\boldsymbol{A_5})\boldsymbol{A_6}))$

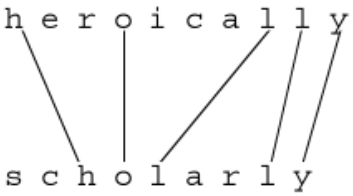
4. Longest Common Subsequence (LCS)

Problem: Given 2 sequences, $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

Examples:



LCS: elm



LCS: holly

Dynamic solution:

Optimal substructure

Suppose, $X_i = prefix\ \langle x_1, \ldots, x_i \rangle$ and $Y_i = prefix\ \langle y_1, \ldots, y_i \rangle$

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}.

Recursive formulation

Define $c[i, j]$ = length of LCS of X_i and Y_j. We want $c[m, n]$.

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

Algorithm
LCS-LENGTH(X, Y)

```
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i − 1, j − 1] + 1
                 b[i, j] ← "↖"
            else if c[i − 1, j] ≥ c[i, j − 1]
                 then c[i, j] ← c[i − 1, j]
                      b[i, j] ← "↑"
                 else c[i, j] ← c[i, j − 1]
                      b[i, j] ← "←"
return c and b
```

- Two tables ***c*** and ***b*** are merged into a single table for ease i.e. each cell contains two information length (c) and direction (b).
- Use b table to print LCS with following recursive algorithm:

```
PRINT-LCS(b, X, i, j)
if i = 0 or j = 0
   then return
if b[i, j] = "↖"
   then PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
elseif b[i, j] = "↑"
   then PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Running Time: ***O(m+n)***

Initial call: PRINT-LCS(b, X, m, n)

Running time: ***O(mn)*** since each table entry takes ***O(1)*** time to compute.

Demonstration:
If Two sequences be: X = ABCBDAB and Y = BDCABA, we use above algorithms to compute tables ***c*** and ***b***.

		j	0	1	2	3	4	5	6
i			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B		0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	C		0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS length: **4**
LCS: **BCBA**

HEY! We will discuss in class, how actually cells are computed.

Greedy Paradigm

This is straight forward way of algorithm design and generally solves optimization problems. In greedy programming: Input elements are exposed to some constraints to get the **feasible solutions** and the feasible solution that meets the **objective function** best among all solutions is called **optimal solution**. Greedy algorithms always make **local optimal choice** in the hope that, it will generate **global optimal solution**. However it is not guaranteed that all greedy algorithms yield optimal solutions.

Most of the problems that can be solved using greedy approach have two parts:

a) Greedy choice property
We can assemble a globally optimal solution by making locally optimal (greedy) choices.

b) Optimal substructure
A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to the sub-problems.

To prove that a greedy algorithm is optimal we must show the above two parts are exhibited by the problem.

1. Fractional-Knapsack problem

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of ith item is w_i and it worth v_i. Any amount of item can be put into the bag i.e. x_i fraction of item can be collected, where $0 \leq x_i \leq 1$. 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 value per weight, continue this until the knapsack is full.

Input: $v[1 \ldots n]$ and $w[1 \ldots n]$ contain the values and weights respectively of the n objects sorted in decreasing order of (v_i/w_i) . W is the capacity of the knapsack.

Output: $x[1 \ldots n]$ is the solution vector that contains fractional amount of n items.

```
GreedyFracKnapsack(W, n)
{
    for(i=1; i<=n; i++)
            x[i] = 0.0;
    tempW = W;
    for(i=1; i<=n; i++)
    {
            if(w[i] <= tempW)
            {
                    x[i] = 1.0;
                    tempW = tempW - w[i];
            }
            else
                    x[i] = tempW/w[i];
    }
    return x;
}
```

Analysis: To sort items into decreasing order of $\left(\frac{v_i}{w_i}\right) = O(nlogn)$ and for loop takes $O(n)$.

$$T(n) = O(nlogn) + O(n) = O(nlogn)$$

Example: Given weight-value pair of items, first sort items on decreasing order of (v_i/w_i) and then just put first item into the bag, when it is finished move to next and next and next.

2. Huffman codes

It compresses data very efficiently: saves up to 20-90%. Assuming data to be sequence of characters, Huffman greedy algorithm uses character frequency table to build up an optimal way of representing each character as a binary string.

Let's start with example: Considering 100000-character file with 6 repeating characters with frequencies:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

Many options to represent such file of information with binary character code:

- **Fixed-Length codeword**: we need 3-bits to represent 6 characters; a=000, b=001,…, f=101, requiring 300000 bits to code entire file. Can it be better? Of course with following option.
- **Variable-Length codeword**: Strategy is to use short codeword for frequent-characters and longer for infrequent. Suppose, we encode like one given in following table. We'll notice the difference.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Requires 300000 bits to store file.

Requires 224000 bits, saves 25%.

We'll see later how this variable-length code is designed. (This is what Huffman greedy algorithm does)

Constructing Huffman codes:

It uses the notion of ***prefix codes***; codes in which no codeword is also a prefix of some other codeword. For example: *001011101* is parsed uniquely as *aabe* using above variable-length scheme. The thing is, if we use variable-codeword like; a=0, b=01, c=101… we can't parse 001011101 uniquely on decoding (ambiguity).

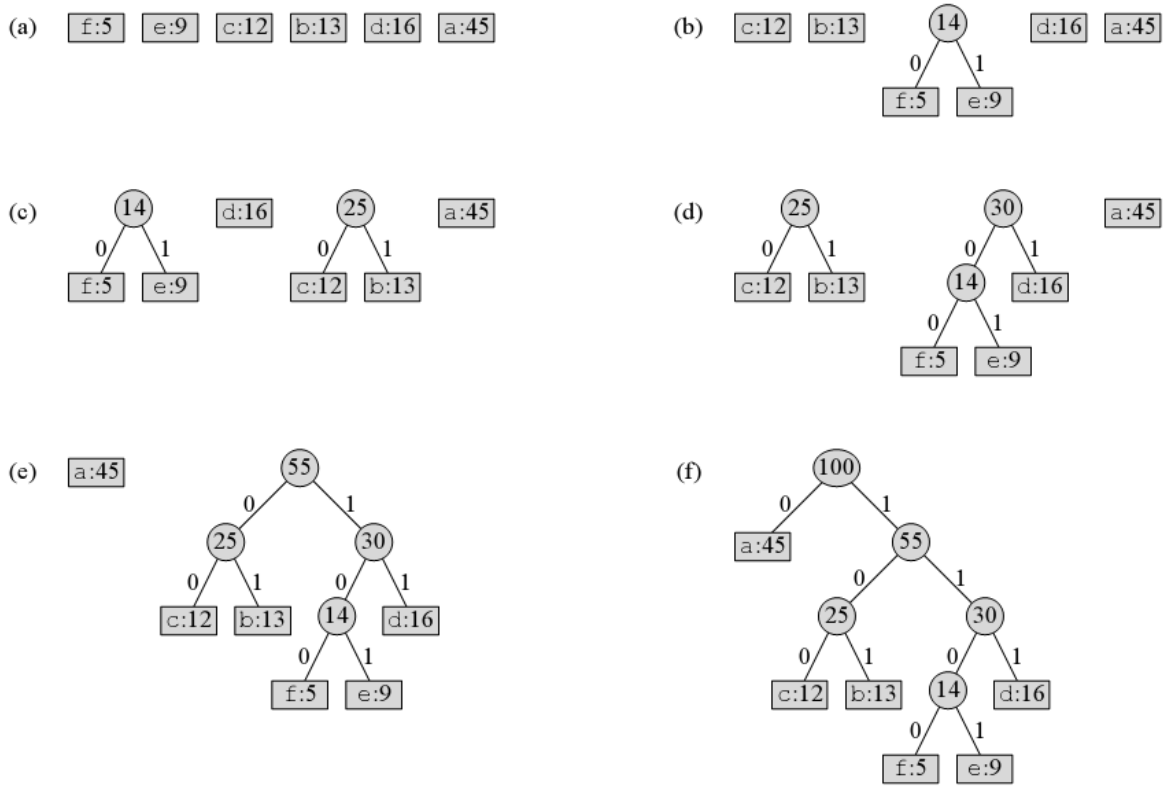
Huffman(C)

$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ **to** $n - 1$
  do allocate a new node z
    $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN(Q)
    $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN(Q)
    $f[z] \leftarrow f[x] + f[y]$
    INSERT(Q, z)
return EXTRACT-MIN(Q) ▷ Return the root of the tree.

A min-priority queue *Q*, keyed on frequency *f*, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

Analysis: For a set *C* of *n* characters, the initialization $Q \leftarrow C$: $O(n)$ time using the BUILD-MINHEAP to maintain priority queue. The **for** loop is executed exactly *n*-1 times, and since each heap operation requires time $O(\log n)$, the loop contributes $O(n \log n)$ to the running time. Thus, the total running time of HUFFMAN on a set of *n* characters is ***O*(*n* log *n*).**

Creating Huffman-tree for same example:



Now start from root to get codeword for particular character.

3. Job-Sequencing with deadlines

Arrangement of jobs on a single processor with deadline constraints is named as *job sequencing with deadlines*. We arrange n jobs in sequence to obtain maximum benefit.

Statement:

- Each i[th] job is associated with a deadline $d_i \geq 0$ and profit $p_i > 0$.
- For any i[th] job, the profit p_i is earned, if job is completed by its deadline.
- Each job takes one-unit of time to complete.
- Only one processor is available for processing all jobs

Feasible solution to this problem is a subset list of jobs such that each job in this subset can be completed by its deadline. A value of feasible solution is the sum of the profits of the jobs in the list($\sum p_i$).

Example: Consider a sequencing problem where 4 jobs have a profit of (10, 30, 60, 40) and corresponding deadlines (2, 3, 1, 3).

Brute-Force approach: List of feasible solutions are:

Sno	*Feasible solution list*	*Processing sequence*	*Total profit* $\sum_{i \in J} p_i$
1	{1, 2}	1 → 2 (Process 2nd job after 1st job)	40
2	{1, 3}	3 → 1 (Process 1st job after 3rd job)	70
3	{1, 4}	1 → 4	50
4	{2, 3}	3 → 2	90
5	{2, 4}	2 → 4 or 4 → 2	70
6	{3, 4}	3 → 4	100
7	{1, 2, 3}	3 → 1 → 2	100
8	{1, 2, 4}	1 → 2 → 4 or 2 → 1 → 4 or 4 → 1 → 2	80
9	{1, 3, 4}	3 → 1 → 4	110
10	{2, 3, 4}	3 → 2 → 4 or 3 → 4 → 2	130
11	{1}	1	10
12	{2}	2	30
13	{3}	3	60
14	{4}	4	40

Solution 10 is optimal yielding a total profit of 130. Optimal job sequence is either (3, 2, 4) or (3, 4, 2).

Greedy approach:
It formulates an optimization measure (objective function($\sum p_i$)) to determine how next job is selected. What it says is: next job to include is that increases total profit the most, which requires considering jobs in decreasing order of p_i's.

```
JOBSEQUENCE(job[1..n], p[1..n], d[1..n])
    Create list[1..n] and initialize all items to 0
    profit = 0
    for i = 1 to n
            k = d[i]
            while(k > 0)
                    if(list[k] = 0)
                            list[k] = job[i]
                            profit += p[i]
                    k--
    return list
```

Time complexity: Sorting (p_i into decreasing order) takes O(n log n) at worst. Selection of jobs takes constant time (O(1)). Consider n jobs in turn. For each job, inserting the job into the partial solution using its deadline takes O(n) time. Hence, total running time: **$O(n^2)$**.

For same example, ordering jobs in decreasing order of profit;
Profits = (60, 40, 30, 10)
Job # = (3, 4, 2, 1)
Deadlines = (1, 3, 3, 2)

job (i)	p_i	d_i	profit	operation	*list*[1..4] initialized to 0
3	60	1	60	Assign job 3 to slot 1.	3 0 0 0
4	40	3	100	Assign job 4 to slot 3.	3 0 4 0
2	30	3	130	Assign job 2 to slot 2, because slot 3 is not empty.	3 2 4 0
1	10	2	130	Reject job 1, because it violates its deadline.	3 2 4 0

Optimal job sequence is 3 → 2 → 4 with maximum profit of 130.

Unit 3
Graph Algorithms, Computational Geometry and NP-Completeness

3.1 Graph Algorithms

Graphs

Graph is a collection of vertices or nodes, connected by edges. Graphs are 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.

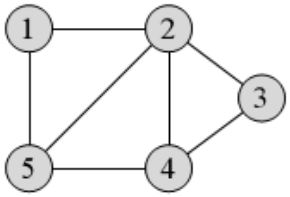
Applications:

- Communication and transportation networks
- VLSI and other sorts of logic circuits
- Surface meshes used for shape description in computer-aided design
- Geographic information systems
- Precedence constraints in scheduling systems etc.

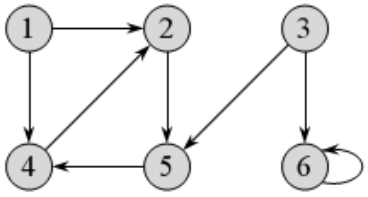
Some definitions:

A **directed graph** (or **digraph**) G = (V, E) consists of a finite set V of **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.



Graph



Digraph

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) on 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. 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 $G = (V, E)$ is said to be **sparse** if $E = \theta(V)$, and **dense** if $|E|$ $is\ close\ to$ $|V|^2$. When giving the running times of algorithms, we will usually express it as a function of both V and E, so that the performance on sparse and dense graphs will be apparent.

Representation of Graph

Graph $G = (V, E)$ can be represented in two standard ways: **adjacency list** and **adjacency matrix**.

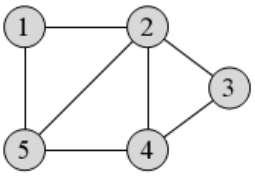
Adjacency List:

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in *V*. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to *u* in *G*. (Alternatively, it may contain pointers to these vertices.) The vertices in each adjacency list are typically stored in an arbitrary order. This type of representation is suitable for the undirected graphs without multiple edges and directed graphs.

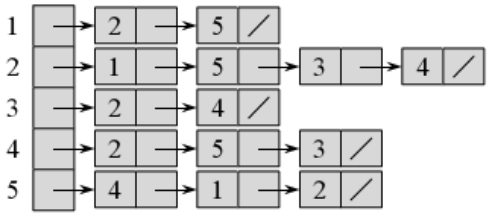
Adjacency Matrix:

For the ***adjacency-matrix representation*** of a graph *G* = (*V*, *E*), we assume that the vertices are numbered 1, 2. . . |*V*| in some arbitrary manner. Then the adjacency-matrix representation of a graph *G* consists of a |*V*| × |*V*| matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 \ if \ (i,j) \in E \\ 0 \ \ otherwise \end{cases}$$



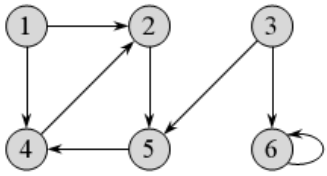
Undirected graph



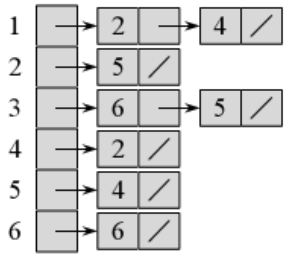
Adjacency-list representation

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

Adjacency-matrix representation



Directed graph



Adjacency-list representation

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency-matrix representation

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.

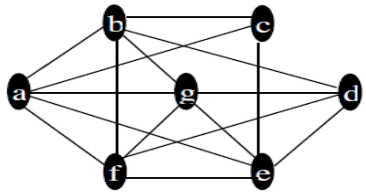
Breadth-First-Search (BFS)

Given a graph *G* = (*V*, *E*) and a distinguished ***source*** vertex *s*, breadth-first search systematically explores the edges of *G* to "discover" every vertex that is reachable from *s*. It computes the distance (smallest number of edges) from *s* to each reachable vertex. It also produces a "breadth-first tree" with root *s* that contains all reachable vertices. The algorithm works on both directed and undirected graphs.

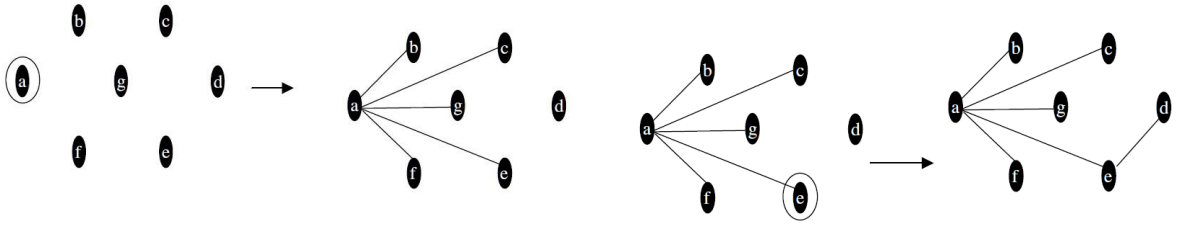
```
BFS(G, s)     //s: start vertex (root)
{
        T = {s};
        L = ∅; //L: queue (Empty)
        Enqueue(L, s);
        while (L != ∅ )
        {
                v = Dequeue(L);
                for each neighbor u to v
                if ( u not in L and u not in T )
                {
                        Enqueue( L, u);
                        T = T U {u}; //put edge {v, u}
                }
        }
}
```

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
Output: $d[v]$ = distance (smallest # of edges) from s to v, for all $v \in V$.

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



Solution:



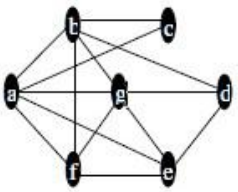
Analysis
From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are searched and this can be done in O(|V|) time (|V|: Number of vertices). This computation is for all the possible vertices that may be in the queue i.e. n, produce complexity of an algorithm as $O(|V|^2)$. Alternatively, 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

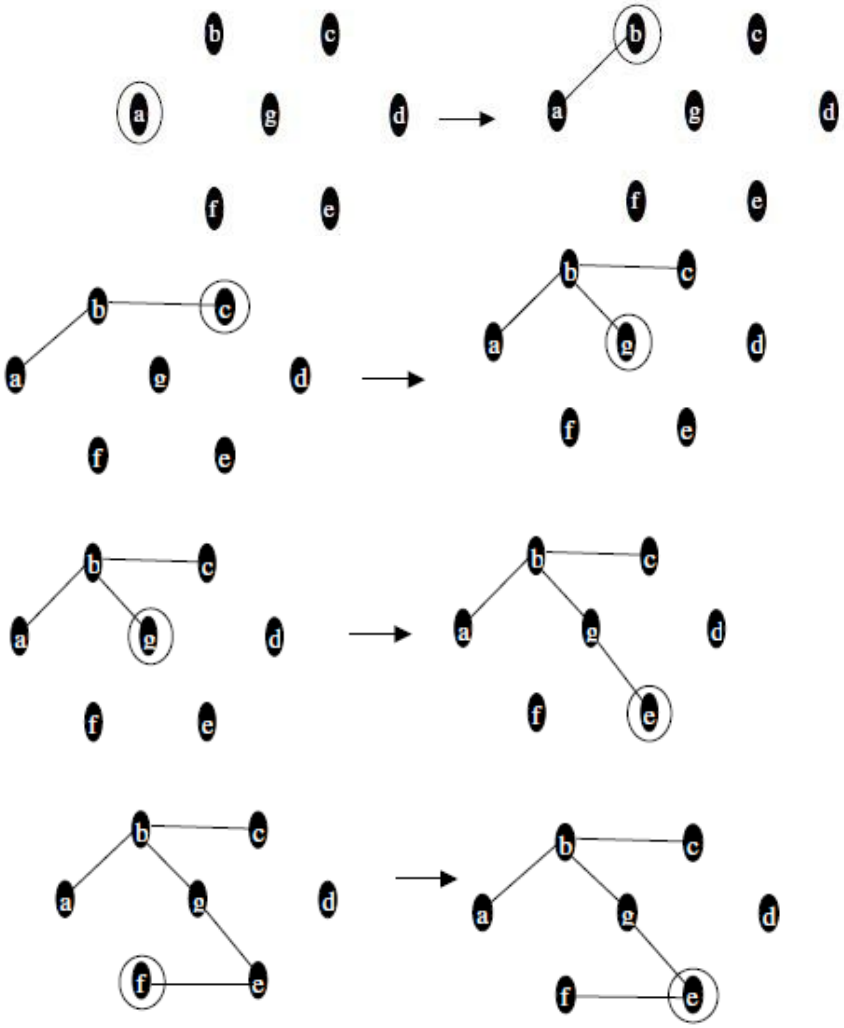
path is **DFS tree**. Otherwise, we must add other edges and vertices. For this move back from the last vertex that is met in the previous path in recursion. This method of search is also called **backtracking**.

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



Solution:

Choose a as initial vertex then we have



Algorithm:

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

Analysis:

The complexity of the algorithm is greatly affected by **Traverse** function, if V be vertex-set and E be edge-set, we can write its running time in terms of the relation $T(|V|) = T(|V|-1) + O(|V|)$, here $O(|V|)$ is for each vertex and 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 $O(|V|^2)$.
Also from aggregate analysis we can write the complexity as $O(|E| + |V|)$ because traverse function is invoked |V| times maximum and for loop executes $O(|E|)$ times in total.

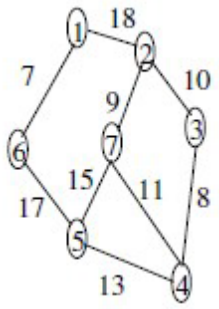
Minimum Spanning Tree

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

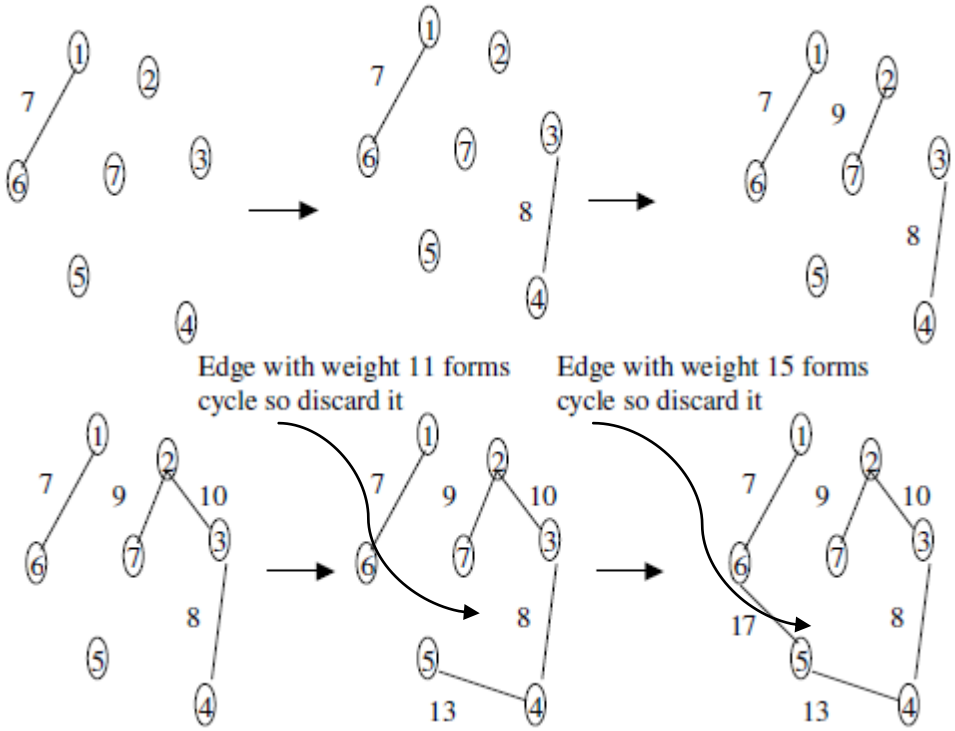
Kruskal's Algorithm:

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

Example: Find MST and its weight of the given graph



Solution:



Total cost of MST: **64**

Algorithm:

```
KruskalMST(G)
{
        T = {V}          // forest of |V| nodes
        S = set of edges sorted in nondecreasing order of weight
        while(|T| < |V| − 1 and E ! = ∅)
        {
                Select (u, v) from S in order
                Remove (u, v) from E
                if((u, v) does not create a cycle in T))
                T = T U {(u, v)}
        }
}
```

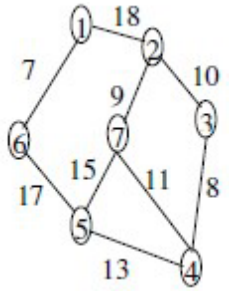
Analysis:

In the above algorithm the n tree forest at the beginning takes |V| time, the creation of set S takes $O(|E|log|E|)$ 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 $O(|E|log|E|)$ or asymptotically equivalent to $O(|E|log|V|)$.

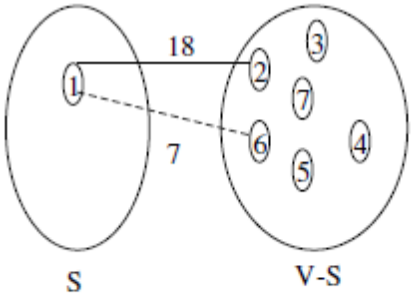
Prim's Algorithm

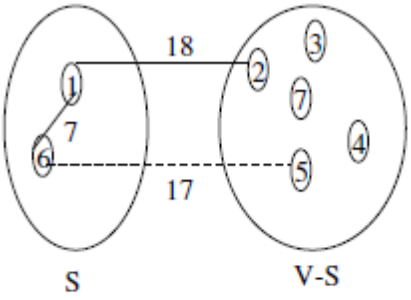
This is another algorithm for finding MST. The idea behind this algorithm is just take any arbitrary vertex and choose the edge with minimum weight incident on the chosen vertex. Add the vertex and continue the above process taking all the vertices added. Remember the cycle must be avoided.

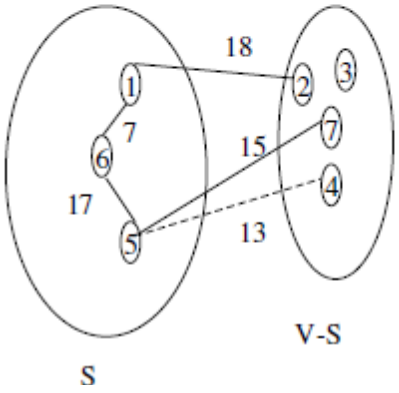
Example: MST of the same graph as Kruskal's:

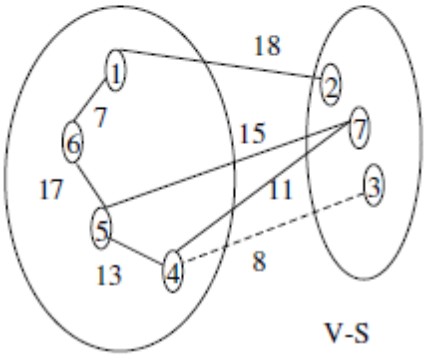


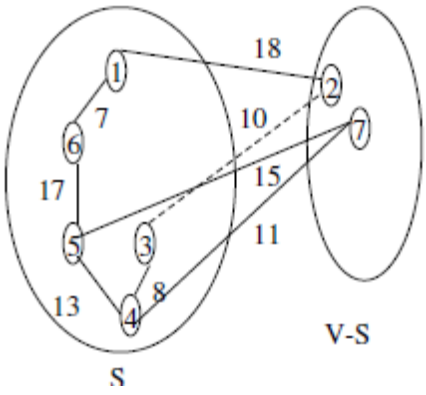
Solution: Each time dotted-edge is chosen

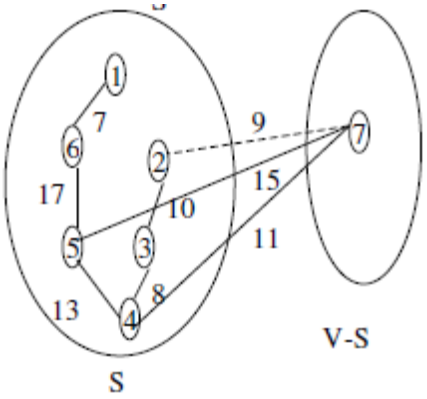


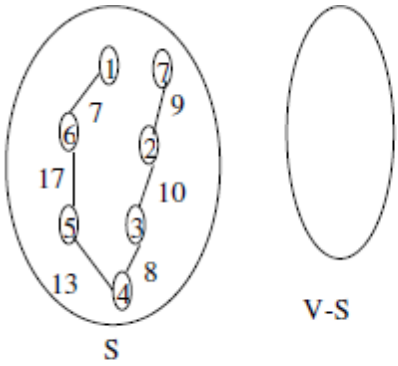












The total weight of MST is 64.

Algorithm:

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

Analysis:

In the above algorithm while loop execute O(|V|). The edge of minimum weight incident on a vertex can be found in O(|E|), so the total time is $O(|E||V|)$. We can improve the performance of the above algorithm by choosing better data structure like priority queue and running time of prim's algorithm will be $O(|E|log|V|)$.

Shortest Path Problem

In a ***shortest-paths problem***, we are given a weighted, directed graph *G* = (*V*, *E*), with weight function $w : E \rightarrow \boldsymbol{R}$ mapping edges to real-valued weights. The ***weight*** of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

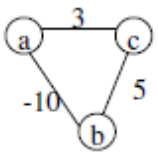
$$w(p) = \sum_{i=1}^{k} w(v_{i-1}v_i)$$

We define the ***shortest-path weight*** from *u* to v by:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise .} \end{cases}$$

A ***shortest path*** from vertex *u* to vertex v is then defined as any path *p* with weight $w(p) = \delta(u, v)$.

It is important to remember that the shortest path may or may not exist in a graph i.e. if there is negative weight cycle then there is no shortest path. For example: the below graph has no shortest path from **a** to **c** .You can notice the negative weight cycle for path a to b.



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

- **Single Source:** This type of problem asks us to find the shortest path from the given vertex (source) to all other vertices in a connected graph.
- **Single Destination:** This type of problem asks us to find the shortest path to the given vertex (destination) from all other vertices in a connected graph.
- **Single Pair:** This type of problem asks us to find the shortest path from the given vertex (source) to another given vertex (destination).
- **All Pairs:** This type of problem asks us to find the shortest path from the all vertices to all other vertices in a connected graph

Single Source Problem

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

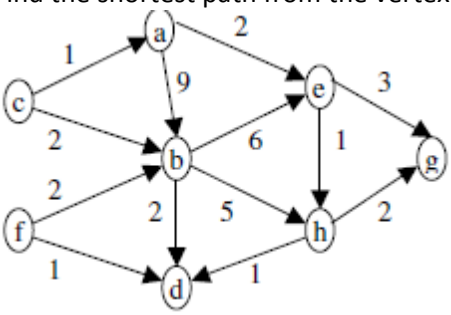
Directed Acyclic Graphs (Single Source Shortest paths)

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

Hey! A ***topological sort*** of a dag *G* = (*V*, *E*) is a linear ordering of all its vertices such that if *G* contains an edge (*u*, v), then *u* appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.) A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

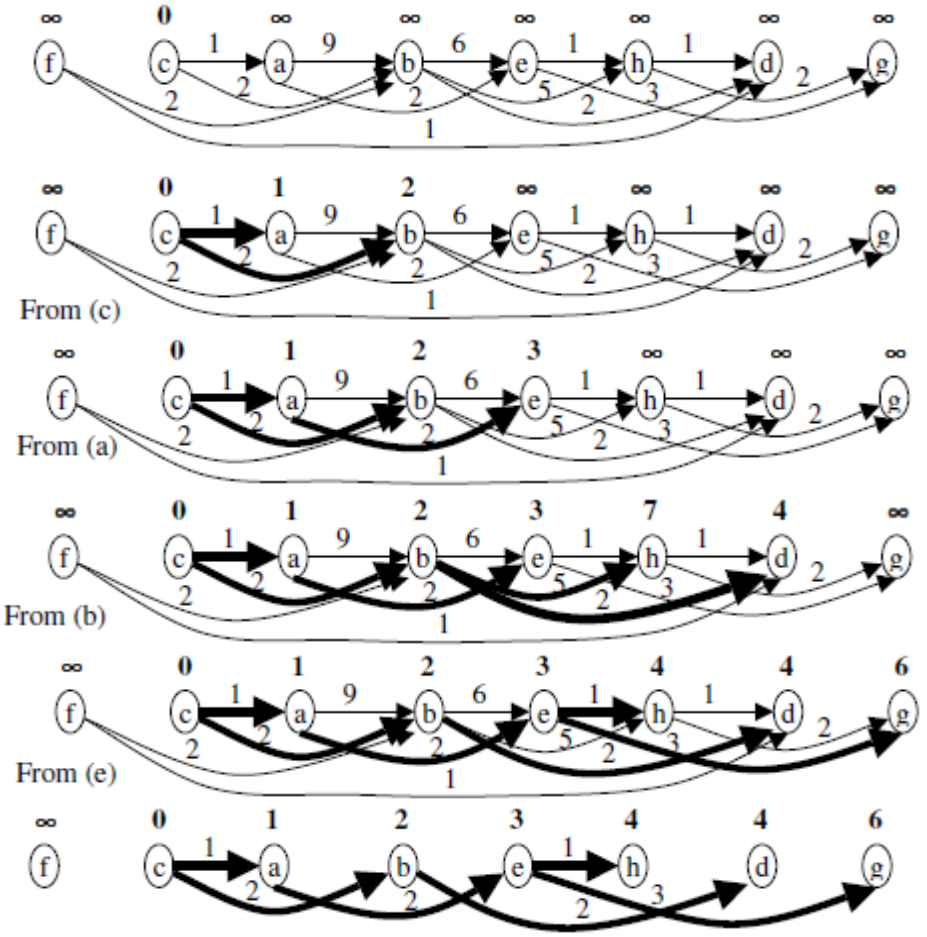
Example:

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



Solution:

Topologically sorted and initialized.



From vertices (h), (d) and (g), no change in d[v], so above is the shortest path tree.

Algorithm:

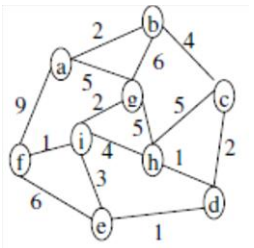
DagSP(G, w, s)
{
    Topologically Sort the vertices of G
    For each vertex v ∈ V
        $d[v] = \infty$
    $d[s] = 0$
    For each vertex u, taken in topologically sorted order
        for each vertex v adjacent to u
            if $d[v] > d[u] + w(u,v)$
                $d[v] = d[u] + w(u,v)$
}

Dijkstra's Algorithm

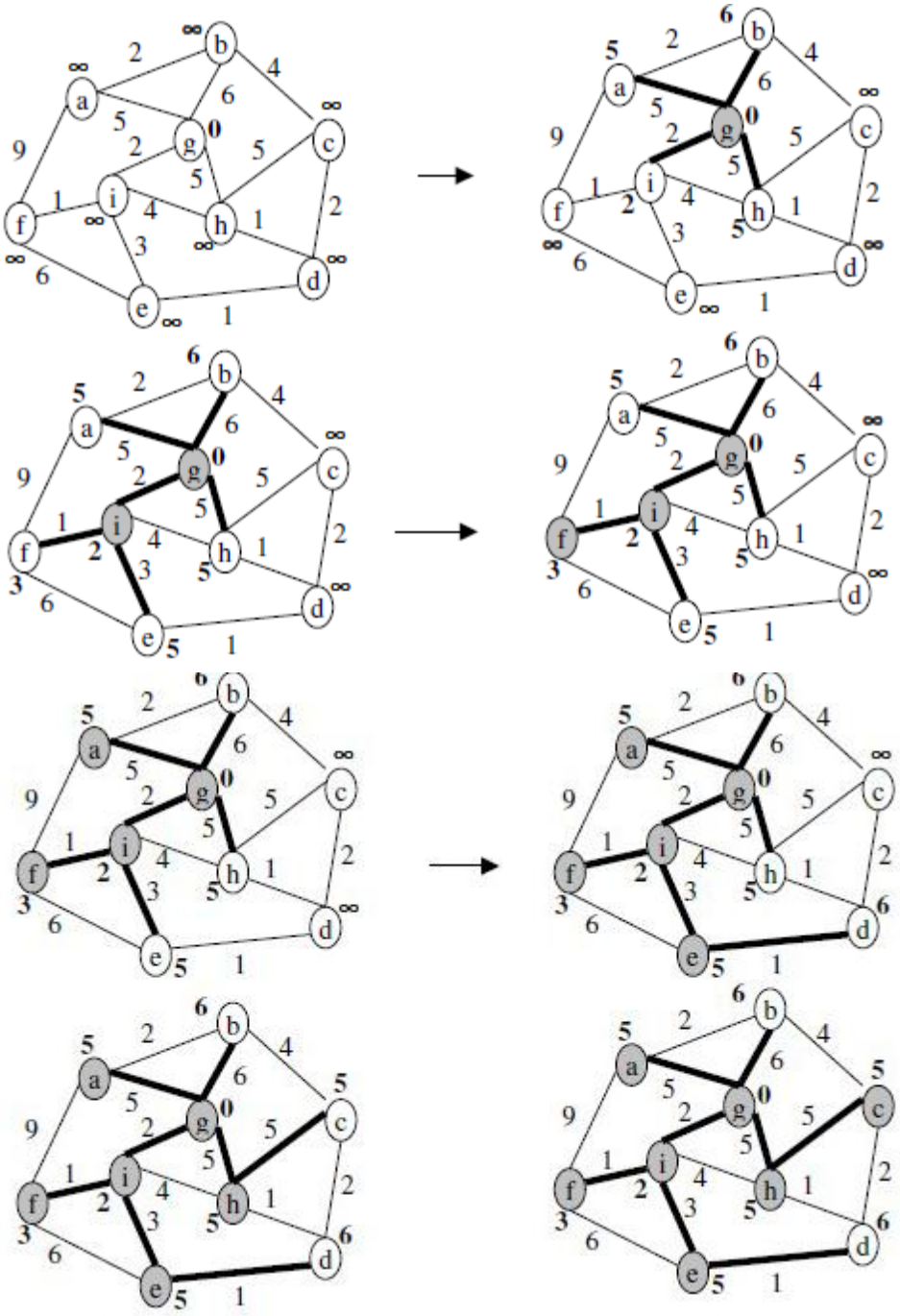
This is another approach of getting single source shortest paths. In this algorithm it is assumed that there is no negative weight edge. Dijkstra's algorithm works using greedy approach, as we will see later.

Example:

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



Solution:



There will be no changes for vertices b and d. Dark connection gives us a shortest-path tree.

Algorithm:

```
DIJKSTRA(G, w, s)
{
        for each vertex v ∈ V
                d[v] = ∞
        d[s] = 0
        S = ∅
        Q = V
        While(Q!= ∅)
        {
                u = Take minimum from Q and delete.
                S = S U {u}
                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 first for loop takes O(V) time. Initialization of priority queue Q takes O(V) time. The while loop executes for O(V), where for each execution the block inside the loop takes O(V) times . Hence the total running time is **$O(V^2)$**.

All Pairs Problem

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

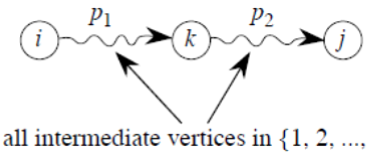
Flyod-Warshall Algorithm

It uses a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph *G* = (*V*, *E*). Negative-weight edges may be present, but we assume that there are no negative-weight cycles.

For path $p = \langle v_1, v_2, \ldots, v_l \rangle$, an ***intermediate vertex*** is any vertex of *p* other than v_1 or v_l.

Let $d_{ij}^{(k)} = shortest\ path\ weight\ of\ any\ path\ i \rightsquigarrow j\ with\ all\ intermediate\ vertices\ in \{1, 2, \ldots, k\}$

- If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, 2, \ldots, k-1\}$.
- If k is an intermediate vertex:



Recursive formulation:

When *k* = 0, a path from vertex *i* to vertex *j* has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$.

Because for any path, all intermediate vertices are in the set {1, 2 . . . *n*}, the matrix $D^{(n)} = d_{ij}^{(n)}$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)\ for\ all\ i, j \in V$.

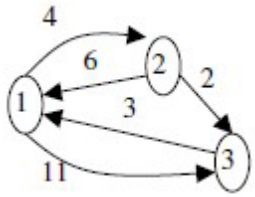
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases}$$

Algorithm:

```
FLOYD-WARSHALL(W, n)
D^(0) ← W
for k ← 1 to n
    do for i ← 1 to n
          do for j ← 1 to n
                do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

Analysis: $O(n^3)$, where n = |V|, cardinality of vertex set V.

Example:



Solution:

Adjacency Matrix

W	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

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

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

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

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

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

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

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

$\mathbf{= \min\{4, 6 + 7\} = 4}$

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

$= \min\{6, 6+ 0\} = 6$

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

$\mathbf{= \min\{6, 2+ 3\} = 5}$

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

$= \min\{2, 2+ 0\} = 2$

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

$= \min\{3, 0+ 3\} = 3$

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

$= \min\{7, 0+ 7\} = 7$

Geometric Algorithms

Computational Geometry

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

Application Domains

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

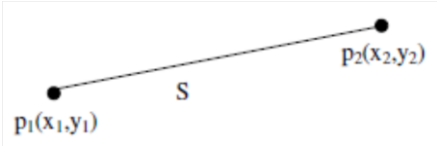
Some Definitions

Point:

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

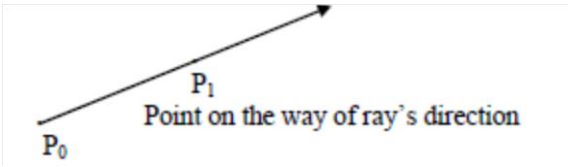
Line segment:

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



Ray:

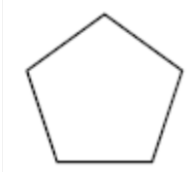
A ray is an infinite one dimensional subset of a line determined by two points: say P_0, P_1, where one point is denoted as the endpoint. Thus, a ray consists of a bounded point & is extended to infinitely along a line segment.



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

Polygon:

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

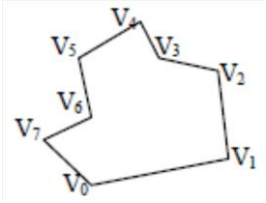


Simple Polygon:

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

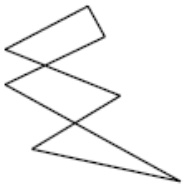
- The intersection of each pair of segments adjacent in cyclic ordering is a simple single point shared by them: $e_i \cap e_{i+1} = V_{i+1}$.
- And non-adjacent segments do not intersect: $e_i \cap e_j = \emptyset$.

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



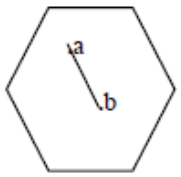
Non-Simple polygon (Self Intersecting)

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

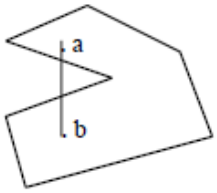


Convex Polygon:

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



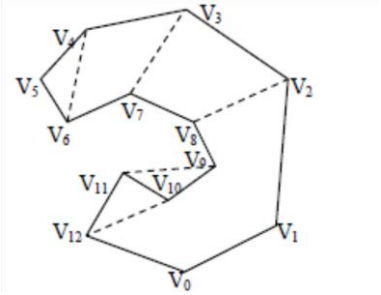
Convex Polygon



Non Convex (Concave)

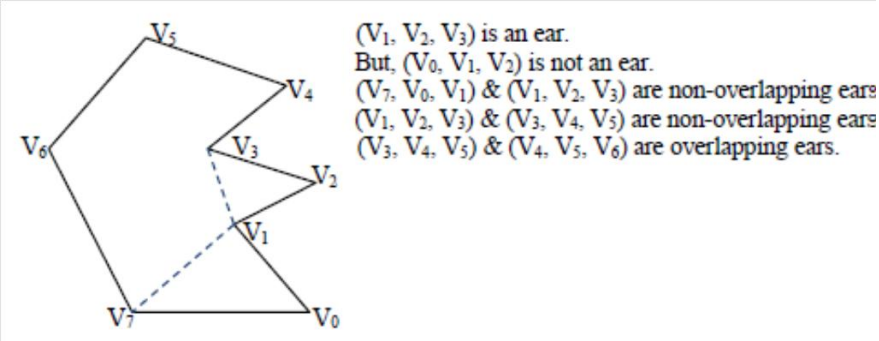
Diagonal of a simple polygon:

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



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

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

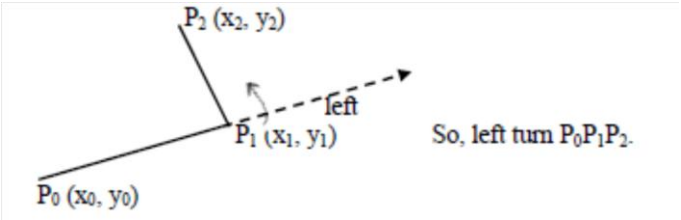


Mouth:
Three consecutive vertices V_i, V_{i+1}, V_{i+2} of a polygon form a mouth if (V_i, V_{i+2}) is an external diagonal. In above figure, (V_0, V_1, V_2) & (V_2, V_3, V_4) are mouths of the polygon.

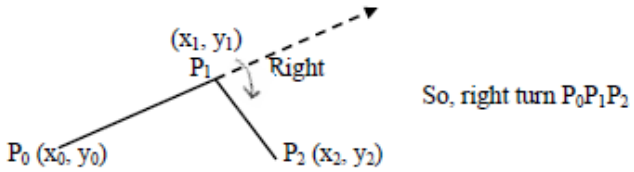
One-Mouth Theorem:
Except for convex polygons, every simple polygon has at least one mouth.
Two-Ear Theorem:
Every polygon of $n \geq 4$ vertices has at least two non-overlapping ears.

Notion of Left Turn & Right Turns

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



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

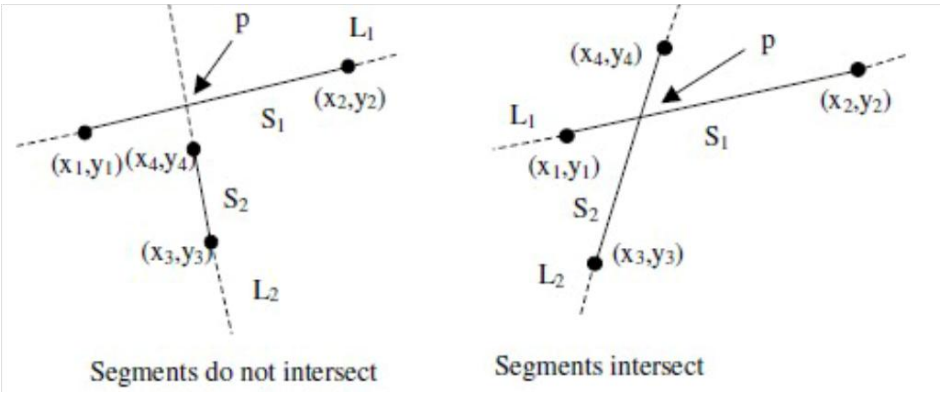


Computing point of intersection between two line segments
We can apply our coordinate geometry method for finding the point of intersection between two line segments. Let S_1 and S_2 be any two line segments. The following steps are used to calculate point of intersection between two line segments. We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

- Determine the equations of line through the line segment S_1 and S_2. Say the equations are L_1 = ($y = m_1x + c_1$) and L_2 = ($y = m_2x + c_2$) respectively. We can find the equation of line L_1 using slope m_1

= $(y_2-y_1)/(x_2-x_1)$, where (x_1,y_1) and (x_2,y_2) are two given end points of the line segment S_1. Similarly we can find L_2 using m_2. The values of c_i's can be obtained by using a point on a line segment on the obtained equation after getting slope of the respective lines.

- Solve two equations of lines L_1 and L_2, let the value obtained by solving be $p = (x_i, y_i)$. Here we confront with two cases. The first case is, if p is the intersection of two line segments then p lies on both S_1 and S_2. The second case is if p is not an intersection point then p does not lie on at least one of the line segments S_1 and S_2.
  The figure below shows both the cases.



Detecting point of intersection

In straightforward manner we can compute the point of intersection (p) between the lines passing through S_1 and S_2 and see whether the line segments intersect or not as done in above discussion. However, the above method uses the division in the computation and we know that division is costly. Here we try to detect the intersection using the concept of turns.

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

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0,\ y_1 - y_0) \times (x_2 - x_0,\ y_2 - y_0)$$
$$= (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0)$$

This can be represented as the determinant:

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

Here we have,

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

See figure below to have idea on left and right turn as well as cross product's geometric interpretation.

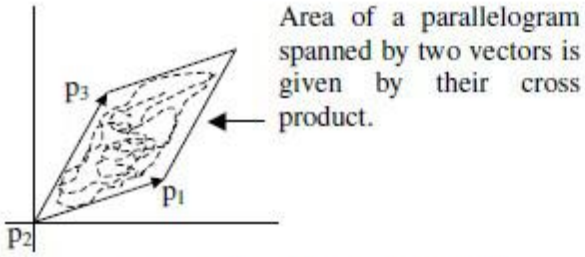


Fig: Cross product geometrical interpretation.

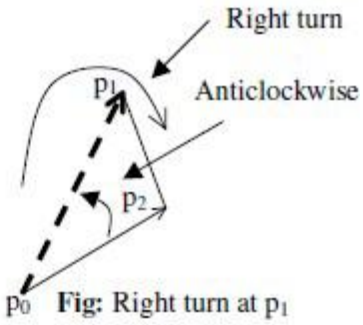


Fig: Right turn at p_1

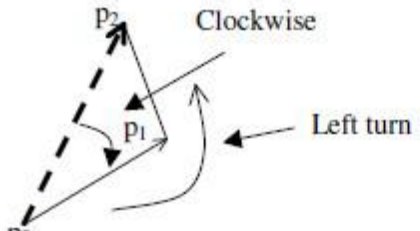


Fig: Left turn at p_1

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

Convex hull

Definition:

- The convex hull of a finite set of points, S in plane is the smallest convex polygon P, that encloses S (Smallest area).
- The convex hull of a set of points, S in the plane is the union of all the triangles determined by points in S.
- The convex hull of a finite set of points, S, is the intersection of all the convex polygons (sets) that contain S.
- There are wide ranges of application areas of convex hulls such as:
    - In pattern recognition, an unknown shape may be represented by its convex hull, which is then matched to a database of known shapes.
    - In motion planning, if the robot is approximated by its convex hull, then it is easier to plan collision free path on the landscape of obstacles.
    - Smallest box, fitting ranges & so on.

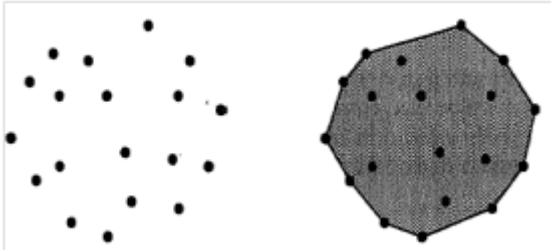


Fig: A point set and its convex hull

Graham-Scan Algorithm

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

Input: P = {p_0, p_1,…, p_{n-1}} of n-points.

Output: Convex hull of P

GRAHAM-SCAN (Q)

1. Find a point q_i with lowest y-coordinate and left most if tied, let it be q_0.
2. Sort the input points angularly about q_0, let the sorted list is now {q_1, q_2, … , q_m}
3. Let S be stack, PUSH(q_0, S), PUSH(q_1, S).
4. For i = 3 to m
   If (Left-Turn (NEXT-TO-TOP(S), TOP(S), q_i) == true)
   PUSH (q_i, S)
   else
   POP (S)
5. Return S

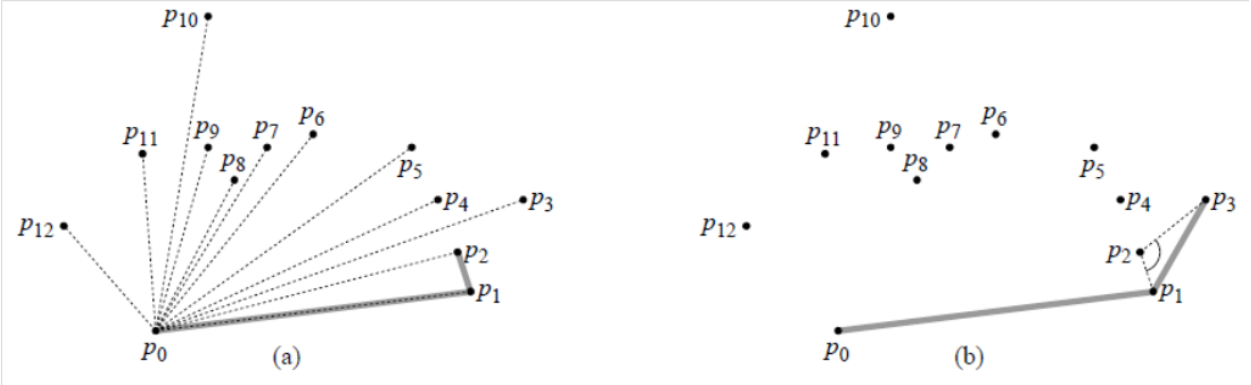
- Each points popped from stack are not vertex of convex hull.
- Finally, when all elements are processed, the points that remain on stack are the vertices of the convex hull.

Complexity Analysis:

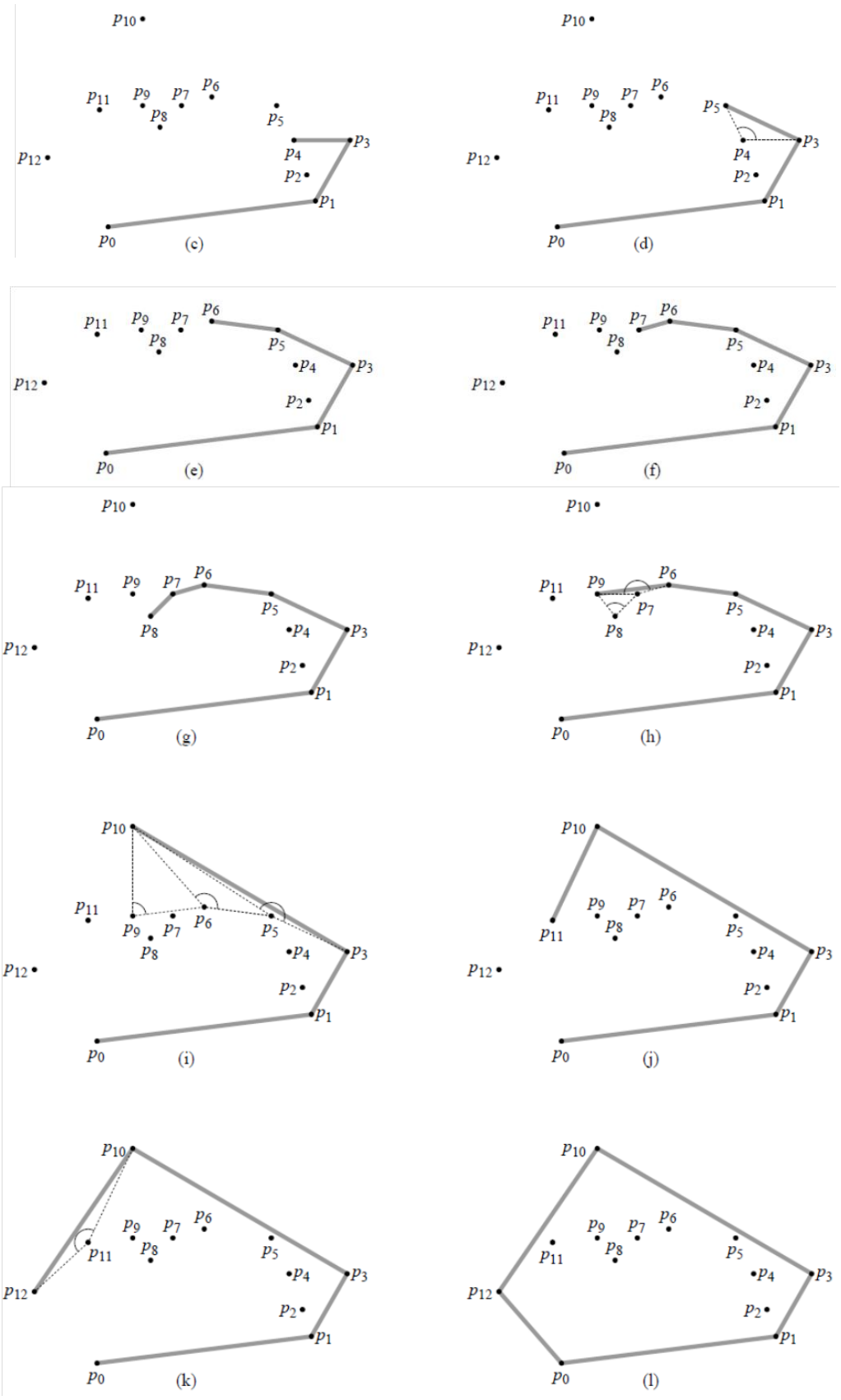
- Finding minimum y-coordinate point it takes $O(n)$ time.
- Sorting angularly about the point takes $O(nlogn)$ time.
- Pushing & popping takes constant time.
- The while loop runs for $O(n)$ times

Hence, the complexity = $O(n) + O(nlogn) + O(1) + O(n) = \boldsymbol{O(nlogn)}$.

Example: The execution of GRAHAM-SCAN on the set *Q* :



(a) The sequence $\langle p_1 p_2 \ldots p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack *S* containing p_0, p_1, and p_2. **(b)–(k)** Stack *S* after each iteration of the for loop. Dashed lines show non-left turns, which cause points to be popped from the stack


p10
p11 p9 p7 p6
p8
p5
p12
p4 p3
p2
p1
p0
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)

NP Complete Problems &Approximation Algorithms

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

Tractable and Intractable Problems

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

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.

Problems

Abstract Problems:

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

Decision Problems:

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

Optimization Problems:

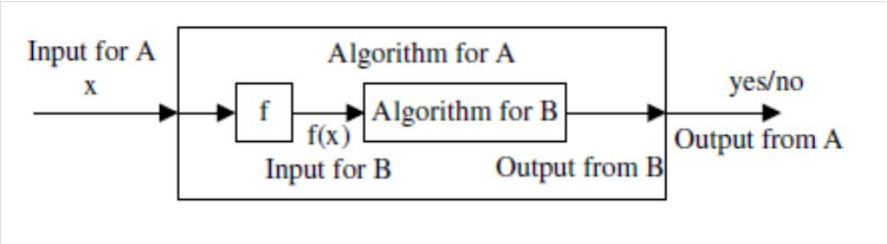
We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with the best value. This kind of problem is called optimization problem. For e.g. given the graph G, and the vertices u and v find the shortest path from u to v with minimum number of edges.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

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

Complexity Class P

Complexity class P is the set of concrete decision problems that are polynomial time (O(log n), O(n^3) etc not O(n!), O(3^n) etc) solvable by deterministic algorithm. If we have an abstract decision problem A with instance set I mapping the set $\{0,1\}$, an encoding e: $I \rightarrow \{0,1\}^*$ is used to denote the concrete decision problem e(A). We have the solutions to both the abstract problem instance $i \in I$ and concrete problem instance $e(i) \in \{0,1\}^*$ as $A(i) \in \{0,1\}$.
We define **polynomial time computable** function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ with respect to some polynomial time algorithm PA such that given any input $x \rightarrow \{0,1\}^*$, results in output f(x). For some set I of problem instances two encoding e_1 and e_2 are **polynomially related** if there are two polynomial time computable functions **f** and **g** such that for any $i \in I$, both $f(e_1(i)) = e_2(i)$ and $g(e_2(i)) = e_1(i)$ are true i.e. both the encoding should computed from one encoding to another encoding in polynomial time by some algorithm.

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



Complexity Class NP

NP is the set of decision problems solvable by nondeterministic algorithms in polynomial time. When we have a problem, it is generally much easier to verify that a given value is solution to the problem rather than calculating the solution of the problem. Using the above idea we say the problem is in class NP (nondeterministic polynomial time) if there is an algorithm for the problem that verifies the problem in polynomial time. V is the verification algorithm to the decision problem D if V takes input string x as an instance of the problem D and another binary string y, certificate, whose size is no more than the polynomial in the size of x. the algorithm V verifies an input x if there is a certificate y such that answer of D to the input x with certificate y is yes.
E.g. Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable? I.e. does the circuit have assignment sequence of truth values that produces the output of the circuit as 1?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in

polynomial time. So we can say that circuit satisfiability problem is NP. We can always say P I NP, since if we have the problem for which the polynomial time algorithm exists to solve (decide: notice the difference between decide and accept) the problem, then we can always get the verification algorithm that neglects the certificate and accepts the output of the polynomial time algorithm. From the above fact we are clear that P I NP but the question, whether P = NP remains unsolved and is still the big question in theoretical computer science. Most of the computer scientists, however, believes that P 1 NP.

NP-Completeness

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

1. $A \in NP$, and
2. $A \leq_p B$, for every $B \in NP$.

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

Cook's Theorem

Theorem 1: "*SAT is NP-hard"*

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

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. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for A_x. Conversely, if A_x outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

Theorem 2: "*SAT is NP-complete"*

Proof: To show that SAT is NP-complete we have to show two properties as given by the definition of NP-complete problems. The first property i.e. SAT is in NP we showed above, so it is sufficient to show the second property holds for SAT. The proof for the second property i.e. SAT is NP-hard is from lemma 3. This completes the proof.

Approximation Algorithms

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

An algorithm has an approximate ratio of $\rho(n)$ if, for any problem of input size n, the cost C of solution by an algorithm and the cost C* of optimal solution has the relation as $max(C/C^*, C^*, C) \leq \rho(n)$. Such an algorithm is called $\rho(n) - approximation\ algorithm$. The relation applies for both maximization $(0 < C \leq C^*)$ and minimization $(0 < C^* \leq C)$ problems. $\rho(n)$ is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have $\rho(n) = 1$.

Vertex Cover Problem

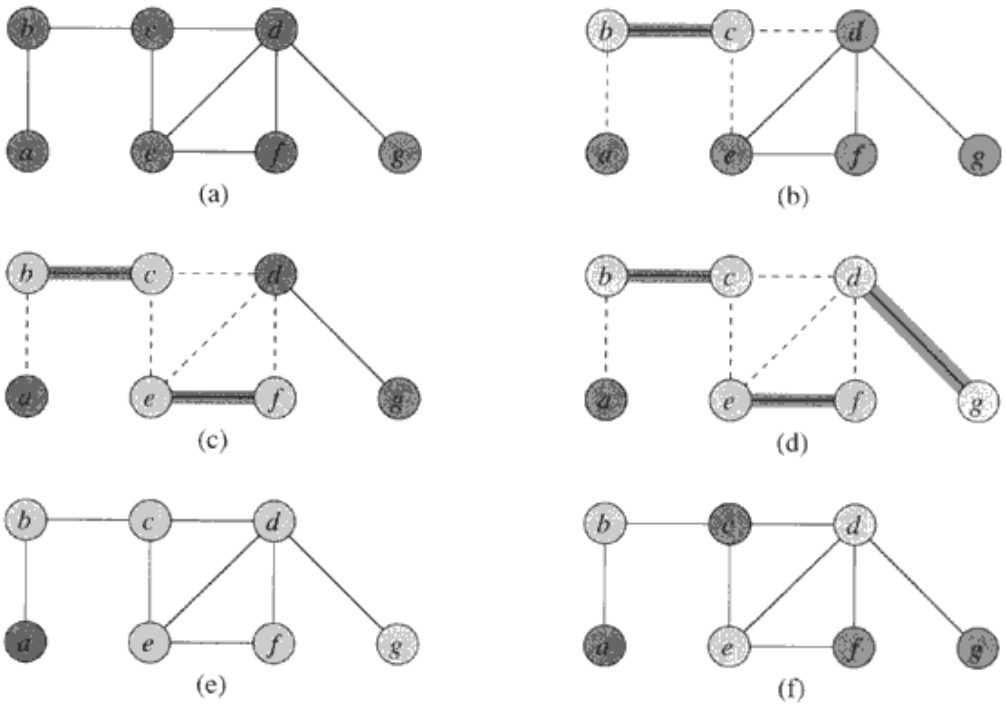
This is NP-Complete problem which states: vertex cover in a graph is a set of vertices such that every edge is incident to (touches) at least one of them. The vertex cover problem is to find the smallest such set of vertices.

Formally, 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 $both$. 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:



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

Syllabus:

Design and Analysis of Algorithms (CSC-303)
Tribhuvan University
Institute of Science and Technology
BSc. CSIT 5th sem syllabus:Design and Analysis of Algorithms

Course Title: Design and Analysis of Algorithms
Course no: CSC-303 **Full Marks: 80+20**
Credit hours: 3 **Pass Marks: 32+8**
Nature of course: Theory (3 Hrs.)

Course Synopsis:
Methods and tools for analyzing different algorithms. Different approaches of designing efficient algorithms like divide and conquer paradigm, greedy paradigm, dynamic programming. Algorithms pertaining various problems like sorting, searching, shortest path, spanning trees, geometric problems etc. NP-complete problems.

Goal:
Competency in analyzing different algorithms encountered. Ability to conquer the problem with efficient algorithm using the algorithm development paradigms.

Course Contents:
Unit 1: **10 Hrs.**
1.1 Algorithm Analysis: worst, best and average cases, space and time complexities. Mathematical background: asymptotic behavior, solving recurrences.
1.2 Data Structures Review: linear data structures, hierarchical data structures, data structures for representing graphs and their properties. Search structures: heaps, balanced trees, hash tables.
Unit 2: **14 Hrs.**
2.1 Divide and Conquer: Concepts, applications, sorting problems(quick, merge), searching (binary), median finding problem and general order statistics, matrix multiplications.
2.2 Greedy Paradigm: Concepts, applications, Knapsack problem, job sequencing, Huffman codes.
2.3 Dynamic Programming: Concepts, applications, Knapsack problem, longest common subsequence, matrix chain multiplications.
Unit 3: **21 Hrs.**
3.1 Graph Algorithms: breadth-first and depth-first search and their applications, minimum spanning trees (Prim's and Kruskal's algorithms), shortest path problems (Dijkstra's and flyod's algorithms), algorithm for directed acyclic graphs (DAGs).
3.2 Geometric Algorithms: Concepts, polygon triangulation, Convex hull computation.
3.3 NP Completeness: Introduction, class P and NP, cooks theorem, NP complete problems: vertex cover problem.
3.4 Introductions: Randomized algorithms concepts, randomized quick sort, approximation algorithms concepts, vertex cover problem.

Textbook: T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, 2001 ISBN: 0-262-530-910.

Reference: G. Brassard and P. Bratley, Fundamentals of Algorithmics, Prentice-Hall, 1996 ISBN: 0-13-335068-1.

Prerequisites: Good programming concepts (any language), Data structures and their properties, mathematical concepts like methods of proof, algorithmic complexity, recurrences, probability.

Assignments: This course deals with wide range of problem domain so sufficient number of assignments from each unit and subunit should be given to the students to familiarize the concepts in depth.

Lab: The motive of this course is to provide good theoretical and mathematical background of algorithms and their analysis; however it is advisable to provide programming assignments that aid the students learn the behavior of the algorithms.