Notes
on
Design and Analysis
of
Algorithm

By

Dr. Satyasundara Mahapatra

(Module –V)

Selected Topics: Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NP-Completeness, Approximation Algorithms and Randomized Algorithms

String Matching:

In General Text editing programs frequently need to find all occurrences of a pattern in the text. Typically the text is a document being edited by the users, and the pattern searched for is a particular word supplied by the user.

Algorithms for such problem known as "string matching" is highly required. These programs are searched the element and edit as per the requirement of users. An efficient algorithms for such problem can greatly aid the responciveness of the text-editing program.

In many applications, string-matching algorithms search particular patterns in DNA sequences, Internet search engines also use them to find Web pages relevant to queries.

Formalization of String-Matching problem.

Let us assume that a text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. We may have

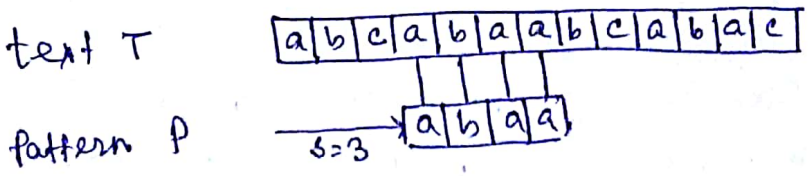
$$\Sigma = \{0, 1\}$$

or $$\Sigma = \{a, b, \ldots, z\},$$

The character arrays P and T are often called strings of character.

Let's discuss with an example.

Scanned by CamScanner



Refferizing to the above figure,

→ the pattern P occurs with shift s in text T, if $0 \leq s \leq n-m$ and $T[s+1..s+m] = P[1..m]$. (i.e $T[s+j] = P[j]$, $\forall 1 \leq j \leq m$).

→ If P occurs with shift 's' in Text 'T' then we call s a valid shift, otherwise s an invalid shift.

The string matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

In the above fig of string matching problem we want to find all occurrances of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text at shift $s = 3$, which we call a valid shift.

Some Algorithms name with Preprocessing time and Matching time is given below.

Algorithm Name	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$.
Finite-automata	$O(m\|\Sigma\|)$	$\Theta(n)$.

Scanned by CamScanner

1. Naive String-Matching Algo.

The Naive String-Matching Algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ possible values of s.

Naive String-Matching Algorithm.

```
NAIVE-STRING-MATCHER(T, P)
1. n = T.length
2. m = P.length.
3. for s = 0 to n-m
4.     if P[1..m] == T[s+1..s+m]
5.         print pattern occurs with shift 's'
```

This algorithm takes $O((n-m+1)m)$ time for producing the results.

2. Rabin-Karp String Matching Algorithm.

→ The Rabin-Karp string matching algorithm calculate a hash value for the pattern, and for each M-character subsequence of text to be compared.

→ If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.

→ If the hash values are equal, the algorithm will compare the pattern and the M-character subsequence of text.

Scanned by CamScanner

→ In this way, there is only one comparision per text subsequence, and character matching is only needed when hash values match.

- For expository purpose, let us assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit.
  - → then view a string k consecutive characters as representing a length-k decimal number.
  - → The character string 31415 thus corresponds to the decimal number 31,415.

Given a pattern $P[1..m]$, let p denote its corresponding decimal value. In a similar manner $T[1..n]$, let t_s denote the decimal value of length-m substring $T[s+1..s+m]$

for $s = 0, 1, \ldots, n-m$

Certainly $t_s = p$ if and only if

$$T[s+1..s+m] = P[1..m]$$

Thus s is a valid shift if and only if $t_s = p$.

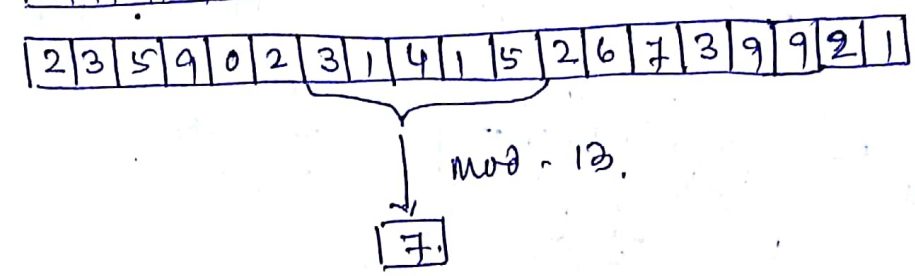
If we could compute p in $\Theta(m)$ time, and all the t_s value in a total of $\Theta(n-m+1)$ time, then we could determine all valid shift s in time $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ by comparing p with each t_s value.

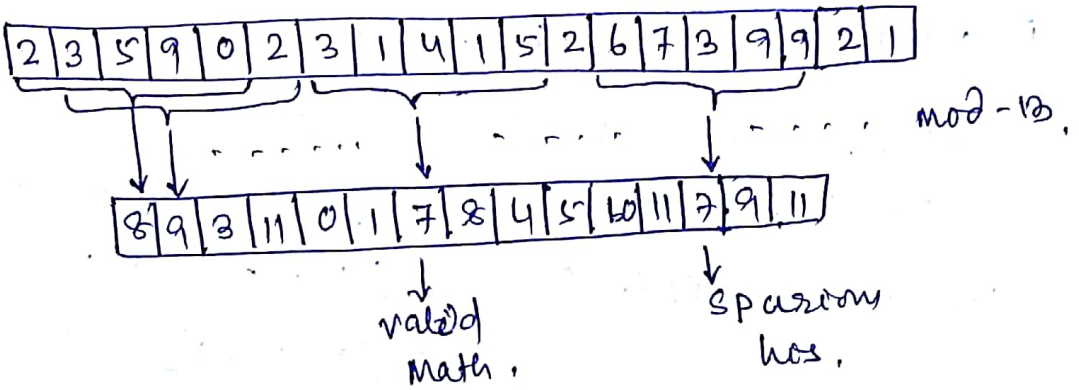
Scanned by CamScanner

For example:

P — | 3 | 1 | 4 | 1 | 5 |

T — | 2 | 3 | 5 | 9 | 0 | 2 | 3 | 1 | 4 | 1 | 5 | 2 | 6 | 7 | 3 | 9 | 9 | 2 | 1 |





For compute p in $\Theta(m)$ time by using Horner's Rule.

i.e $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots)$

So,

$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10[P[2] + 10(P[1])])$

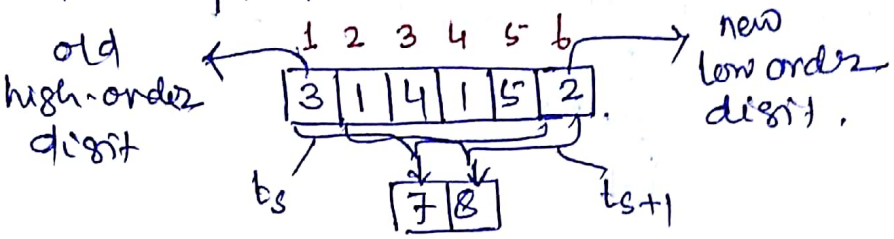
Similarly, we compute to form $T[1..m]$ in time $\Theta(m)$

To compute the remaining value $t_1, t_2, \ldots t_{n-m}$ in $\Theta(n-m)$, we observe that we can compute t_{s+1} from t_s in constant time, since

$$t_{s+1} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1]$$

Subtracting $10^{m-1} T[s+1]$ remove the higher-order digit from t_s, multiplying

the result by 10 shift the number left by one digit position and adding T[s+m+1] brings in the appropriate low-order digit. For example,



If m = 5, and $t_s = 31415$

then we wish to remove the high-order digit T[s+1] = 3 (where s=0), and bring the new low order digit (i.e T[s+m+1] =

$\Rightarrow T[0+5+1] =$

$\Rightarrow T[6] = 2$

$$t_{s+1} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1]$$

$$= 10(31415 - 10^4(3)) + T[6]$$

$$= 10(1415) + 2$$

$$= 14152$$

If p and t_s be too large then it is very difficult for calculation. i.e if P contains m characters then we cannot reasonably assume that each arithmatic operation on p takes "constant time". So such problem can be solved by module. mechanism. i.e compute p and t_s values module a suitable modulus q.

we can compute p modulo q in $\Theta(m)$ time and all the t_s values modulo q in $\Theta(n-m+1)$ time. If we choose the modulus q as a prime such that 10q just fits within one computer word, then we can perform all the necessary computations with single precision arithmatic.

In general a d-any (here d=10) alphabet $\{0, 1, \ldots, d-1\}$, we choose q so that dq fits within a computer word and adjust the recurrance equation ($i.e$ t_{s+1} equation) to work modulo q, so that it becomes

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q.$$

where $h = d^{m-1} \bmod q$.

```
Rabin-Karp-Matcher (T, P, d, q).
1.  n = T.length
2.  m = P.length
3.  h = d^(m-1) mod q
4.  p = 0
5.  to = 0
6.  for i = 1 to m
7.      p = (dp + P[i]) mod q
8.      to = (d to + T[i]) mod q
9.  for s = 0 to n-m
10.     if p == ts
11.         if P[1..m] == T[s+1..s+m]
12.             Print "Pattern occurs with shift 's'
13.     if s < n-m
14.         ts+1 = ((d(ts - T[s+1]h) + T[s+m+1]) mod q
```

Scanned by CamScanner

Here T =

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

P =

1	2	3	4	5
3	1	4	1	5

d = 10 i.e $\{0, 1, \ldots, 9\}$.

q = A prime no. i.e 13.

1. n = T.length = 19
2. m = P.length = 5
3. $h = d^{m-1} \bmod q = 10^{5-1} \bmod 13$
   $= 10^4 \bmod 13. = 3.$
4. p = 0
5. $t_0 = 0$.

for step 6.

i = 1

$p = (dp + P[i]) \bmod q$

$= (10 \times 0 + P[1]) \bmod 13.$

$= (0 + 3) \bmod 13 = (3)$

$t_0 = (dt_0 + T[i]) \bmod q$

$= (10 \times 0 + T[1]) \bmod 13,$

$= (0 + 2) \bmod 13 = (2)$

so p = 3 and $t_0 = 2$

i = 2 $p = (dp + P[i]) \bmod q$

$= (10 \times 3 + P[2]) \bmod 13.$

$= (30 + 1) \bmod 13.$

$= 31 \bmod 13 = 5$

Scanned by CamScanner

to = (d to + T[i]) mod q.

= (10 × 2 + T[2]) mod q.

= (20 + 3) mod 13 = 23 mod 13 = 10.

So p = 5 and to = 10.

i = 3.

p = (d p + P[i]) mod q.

= (10 × 5 + 4) mod 13. = 54 mod 13 = 2

to = (d to + T[i]) mod q

= (10 × 10 + 5) mod 13 = 105 mod 13. = 1

So p = 2 and to = 1

i = 4

p = (d p + P[i]) mod q.

= (10 × 2 + 1) mod 13 = 21 mod 13 = 8

to = (d to + P[i]) mod q.

= (10 × 1 + 9) mod 13. = 19 mod 13 = 6.

So p = 8 and to = 6.

i = 5

p = (d p + P[i]) mod q

= (10 × 8 + 5) mod 13 = 85 mod 13. = (7)

to = (d to + T[i]) mod 13.

= (10 × 6 + 0) mod 13 = 60 mod 13 = (8).

So p = 7 and to = 8

Hence Similarly execute for the remaining 14. of the Algo. (Lets see one operation on next page.)

Scanned by CamScanner

9. $s = 0$

i.e $P = t_s$ False.

i.e $P = t_0$.

13. $s < n - m$

$0 < 14$ True.

$t_{s+1} = ((d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$.

$\Rightarrow t_{0+1} = ((10(t_0 - T[0+1]3) + T[0+5+1]) \bmod 13$

$= ((10(8 - 2 \cdot 3) + 2) \bmod 13$

$= ((10(8-6)+2)) \bmod 13$.

$= (22 \bmod 13)$.

$= 9$.

So. $\Rightarrow t_1 = 9$

Similarly do till same until $s < n-m$.

Rabin-Karp-Matcher takes

$O(m)$ ← preprocessing time

$O((n-m+1)m)$ ← matching time in worst case.

Scanned by CamScanner

String matching with finite automata

Many string-matching algorithms build a finite automaton - a simple machine for processing information - that scans the text string T for all occurrences of the pattern P.

These string matching automata are very efficient: they examine each text character exactly once, taking constant time per text character. The matching time used - after preprocessing the pattern to build the automaton - is therefore $\Theta(n)$. The time to build the automaton, however, can be large if Σ is large.

Finite automata

A finite automaton M, illustrated below (*) is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

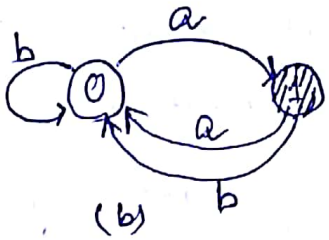
- Q is a finite set of states, (set of states)
- $q_0 \in Q$ is the start state, (initial state)
- $A \subseteq Q$ is a distinguished set of accepting states. (set of final states)
- Σ is a finite input alphabet. (all i/p alphabet)
- δ is a function from $Q \times \Sigma$ into Q, called the transaction function of M. (transaction func)

Scanned by CamScanner

A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{a, b\}$.

State	Input a	Input b
0	1	0
1	0	0

(a)



(b)

(a) A tabular representation of the transaction function δ.

(b) An equivalent state-transition diagram:-

State 1 is the only accepting state. Directed edges represent transitions.

For example:

→ the edge from state 1 to state 0 labelled b indicates that $\delta(1, b) = 0$. This automaton accepts those strings that end in an odd numbers of a's.

Let us solve a string matching with finite automata.

$T = a\ b\ a\ b\ a\ b\ a\ c\ a\ b\ a$.

$P = a\ b\ a\ b\ a\ c\ a$

In order to specify the string matching automaton corresponding to a given pattern $P[1..m]$, we first define an auxillary function σ, called the suffix function corresponding to P.

Scanned by CamScanner

The function σ maps Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x:

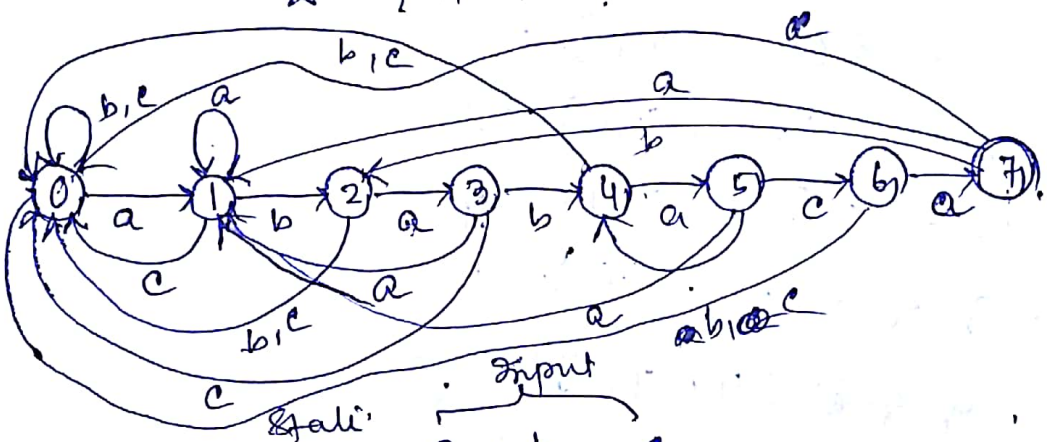
$$\sigma(x) = \max\{k : P_k \sqsupset x\}.$$

T = a b a b a b a c a b a.

P = a b a b a c a.

Total state = 8

$\Sigma = \{a, b, c\}$.



State	Input a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

Scanned by CamScanner

Finite Automata or - Matcher (T, δ, m)

1. n = T.length
2. q = 0.
3. for i = 1 to n
4. q = δ (q, T[i])
5. if q == m
6. print "pattern occurs with shift" i − m.

i = 1 to 11

1. q ← δ (0, T[1]) = 1
2. q ← δ (1, T[2]) = 2
3. q ← δ (2, T[3]) = 3.
4. q ← δ (3, T[4]) = 4
5. q ← δ (4, T[5]) = 5
6. q ← δ (5, T[6]) = 4
7. q ← δ (4, T[7]) = 5
8. q ← δ (5, T[8]) = 6.
9. q ← δ (6, T[9]) = 7.

Shift = 9 − 7 = 2 position onwards.

Time complexity = O(n).

Solve the following string matching problem with finite automata.

T = a b b a b b c b.

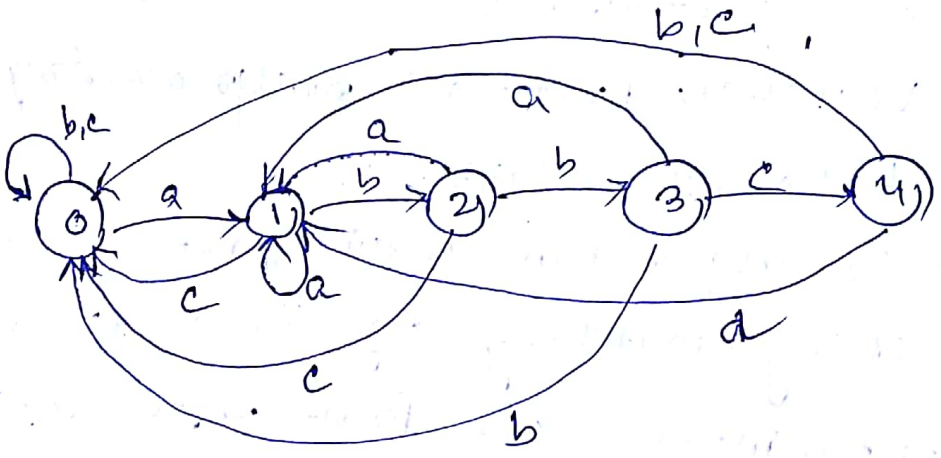
P = a b b c.

Scanned by CamScanner

Total State = 5 . i.e $\{0,1,2,3,4\}$

$\Sigma = \{a, b, c\}$

State	a	b	c
	o/p.		
0	1	0	0
1	1	2	0
2	1	3	0
3	1	0	4
4	1	0	0



i = 1 to 8.

1. $q = \delta(0, T[1]) = 1$
2. $q = \delta(1, T[2]) = 2$
3. $q = \delta(2, T[3]) = 3$
4. $q = \delta(3, T[4]) = 1$
5. $q = \delta(1, T[5]) = 2$
6. $q = \delta(2, T[6]) = 3$
7. $q = \delta(3, T[7]) = 4$

Shift = 7 − 4 = 3 positions towards.

Scanned by CamScanner

Knuth-Morris-Pratt algorithm. (KMP Algorithm)

This algorithm avoids computing the transaction function δ together, and its matching time is $O(n)$ using an auxiliary function π or LSP (longest Prefix string), which computed from the pattern in time $O(m)$ and stored in an array $\pi[1..m]$.

→ This algorithm was designed by Donald Knuth, Vaughan Pratt and ~~independently by~~ James H. Morris in 1977.

→ This algorithm is the first linear time string-matching algorithm discovered by Knuth, Morris and Pratt after analysis the naive Algorithm.

→ The basic algorithm, (naive) does not study the pattern and blindly check every alphabet. By avoiding this waste of information, it achieves a running time $O(m+n)$

→ The implementation of Knuth-Morris-Pratt algorithm is efficient because it minimizes the total number of comparisons of the patter against the input string.

Scanned by CamScanner

Components of KMP.

→ The prefix function Π:
- → It preprocesses the pattern to find all matches of prefixes of the pattern with the pattern itself.
- → It is defined as the size of the largest Prefix of $P[0..j-1]$ that is also a Suffix of $P[1..j]$
- → It also indicates how much of the last comparison can be reused if it fails.
- → It enables avoiding backtracking on the string S.

```
KMP-MATCHER(T, P)
1.  n = T.length
2.  m = P.length
3.  Π = Compute-Prefix-function(P)
4.  q = 0
5.  for i = 1 to n
6.      while q > 0 and P[q+1] ≠ T[i]
7.          q = Π[q]
8.      if P[q+1] == T[i]
9.          q = q+1
10.     if q == m
11.         Print pattern occurs with shift i-m
12.         q = Π[q]
```

Scanned by CamScanner

Compute-Prefix-function (P)

1. $m = P.length$
2. let $\Pi[1..m]$ be a new array
3. $\Pi[1] = 0$
4. $k = 0$
5. for $q = 2$ to m
6. while $((k > 0)$ and $(P[k+1] \neq P[q]))$
7. $k = \Pi[k]$
8. if $P[k+1] == P[q]$
9. $k = k+1$
10. $\Pi[q] = k$
11. return Π.

Let us execute an example with following Text T and Pattern P.

$T = b\ a\ c\ b\ a\ b\ a\ b\ a\ b\ a\ c\ a\ a\ b$

$P = a\ b\ a\ b\ a\ c\ a.$

First calculate Π (Prefix) for Pattern P.

P =	a	b	a	b	a	c	a

Initially $m = length[P] = 7$

$\Pi[1] = 0$

$k = 0.$

Where m = length of Pattern

$\Pi[1]$ ← Prefix function

k ← initial Potential value.

Scanned by CamScanner

Step-1 $q = 2$ $k = 0$ $\Pi[2] = 0$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0					

Step-2 $q = 3$, $k = 1$, $\Pi[3] = 1$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0	1				

Step-3 $q = 4$ $k = 2$, $\Pi[4] = 2$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0	1	2			

Step-4 $q = 5$ $k = 3$ $\Pi[5] = 3$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step-5 $q = 6$ $k = 0$ $\Pi[6] = 0$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	

Step-6 $q = 7$ $k = 1$ $\Pi[7] = 1$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

Scanned by CamScanner

Now let us execute KMP algorithm for finding wheather pattern 'P' occurs in text 'T'

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

	1	2	3	4	5	6	7
P =	a	b	a	b	a	c	a

Initially n = T.length = 15

m = P.length = 7.

	1	2	3	4	5	6	7
Π =	0	0	1	2	3	0	1

q = 0.

Step 1 i = 1 q = 0

T = b a c b a b a b a b a c a a b.

P = a b a b a c a.

P[1] does not match with T[1], hence i incremented by 1

Step 2 i = 2 q = 0.

T = b a c b a b a b a b a c a a b

P = a b a b a c a.

Comparing P[1] with T[2] and q is 1

Scanned by CamScanner

Step-3

$i = 3 \quad q = 1$

T = b a c b a b a b a c a a b

P = a b a b a c a

P[1] compare with T[3] Match unsuccessfully

So $q = \Pi[q]$ i.e $q = 0$.

Step-4 $i = 4 \quad q = 0$.

T = b a c b a b a b a c a a b,

P = a b a b a c a

Compare P[q+1] with Ti i.e P[1] with T[4] False.

Hence $q = 0$.

Step-5 $i = 5 \quad q = 0$.

T = b a c b a b a b a c a a b

P = a b a b a c a.

Compare P[q+1] with Ti i.e P[1] with T[5] ⇒ True

So $q = 1$

Step-6 $i = 6 \quad q = 1$

T = b a c b a b a b a c a a b.

P = a b a b a c a

Compare P[q+1] with Ti, i.e P[2] with T[6] → True

So $q = 2$

Step-7 $i = 7 \quad q = 2$

T = b a c b a b a b a c a a b

P = a b a b a c a.

Compare P[q+1] with Ti i.e P[3] with T[7] ⇒ True

So $q = 3$.

Scanned by CamScanner

Step-8

i = 8 q = 3

T = b a c b a b a (b) a b a c a a b.

P = a b a (b) a c a

Compare P_{q+1} with T_i i.e $P_3[4]$ with T[8] True.

So q = 4

Step-9 i = 9 q = 4

T = b a c b a b a b (a) b a c a a b

P = a b a b (a) c a

Compare P_{q+1} with T_i i.e P[5] with T[9] → True

So q = 5

Step-10 i = 10 q = 5

T = b a c b a b a b a (b) a c a a b.

P = a b a b a (c) a

Compare P_{q+1} with T_i i.e P[6] with T[10] – False.

hence q = Π[q] i.e q = 3.

again P_{q+1} compare with T_{10}.

i.e P_4 with T_{10}. – True.

So q = 4

Step-11 i = 11 q = 4

T = a b a c b a b a b a b (a) c a a b.

P = a b a b (a) c a.

Compare P_{q+1} with T_i i.e ($P_3[5]$ with T[11]) → True.

q = 5

Step-12 i = 12 q = 5

T = b a c b a b a b a b a (c) a a b.

P = a b a b a (c) a.

Compare P_{q+1} with T_i i.e P[6] with T[12] True.

q = 6

Scanned by CamScanner

Step-13 $i = 13$ $q = 6$.

T = b a c b a b a b a c a a b,

P = a b a b a c a.

Compare P_{q+1} with T_i i.e P[7] with T[13] → True.

So $q = 7$

Here $q = m$

So pattern occurs with shift 13 − 7 positions onwards. i.e. b in position onwards are $q = \Pi[q]$ i.e. $q = 1$.

and again search is there any other position where pattern available.

Runtime Analysis.

$O(m)$ ← time required to compute the Prefix function value.

$O(n)$ ← time required to compare the pattern to the text.

Hence total time required to compute the KMP Algorithm is $O(m+n)$.

Scanned by CamScanner

NP-Completeness. (Theory.)

So far all the algorithm that we have learned are written below in two categories. They are Polynomial time based algorithms or exponential time based algorithms. Some of them are

Polynomial time based Algorithm	Exponential time based Algorithm.
→ Linear search. - $O(n)$	→ 0/1 knapsack - $O(2^n)$
→ Binary search - $O(\log n)$	→ Travelling Salesman problem - $O(2^n)$
→ Insertion sort - $O(n^2)$	→ Sum of subset - $O(2^n)$
→ Merge sort - $O(n \log n)$	→ N Queen - $O(2^n)$
→ Matrix Multiplication - $O(n^3)$.	→ Graph Colouring - $O(2^n)$
	→ Hamiltonian Cycle - $O(2^n)$

→ The problem for which we know the algorithm in polynomial time algorithm. otherwise known as

→ Like searching Algorithm, this problem is searching and the algorithms are linear search and binary search, which takes $O(n)$ and $O(\log n)$ time for execution respectively. polynomial.

→ Similarly Sorting Algorithm, the problem is sorting and the algorithms are Insertion sort, Merge sort, which takes $O(n^2)$ and $O(n \lg n)$ polynomial time for execution respectively.

Scanned by CamScanner

Like wise Matrix Multiplication takes $O(n^3)$ Polynomial time.

Similarly in case of exponential time 0/1 Knapsack, TSP, Sum of Subset, Graph Coloring etc. takes $O(2^n)$ ~~exponential~~ time for execution. Some algorithm also take $O(n^n)$ execution time, but for convenient. we take $O(2^n)$. in common.

Hence we catogries the algorithms in to two - one is polynomial time taking Algorithm and another is exponential time taking Algorithm.

The topic NP-complete and NP-Hard is a most important as well as most confusing topic. Actually this topic is research based topic, then the question arise in it that what is in bit for research?.

Let's see that for Searching Problem it was observed that 'Linear search takes $O(n)$ times Binary Search takes $O(\lg n)$ times. So can we develop a logic ~~with~~ which takes $O(1)$ times.?.

Similary for Sorting problem Insertion Sort takes $O(n^2)$ time and Merge Sort takes $O(n \lg n)$ time. Can we develop a logic which takes only $O(n)$ time?

Now ~~for~~ for exponential time taken Problem, can we develop polynomial time algorithm?. i.e develop a faster and

Scanned by CamScanner

easy methods which takes polynomial time to solve these problem, because

exponential time > polynomial time.

i.e $2^n > n^{10}$ large.

$> n^{100}$ for some value of n.

So these are very time consuming algorithm, hence the requirement is polynomial algorithm for the exponential time consuming problems. So the persons from computer science or mathematics can solve these problems by taking it as their resource problem.

But till now there is no polynomial time algo for these exponential problem. Then, when the research work is going fruitless, we want something such that what ever the work has been done should be useful.

So there are guidelines or frames are made for doing research in these exponential problem. and that frame work is NP-hard and NP-complete. So let us see the basic idea behind this. Two types of reasearch, their area two points (objective) behind this. They are.

1. When we are unable to solve these exponential problems (i.e unable to find polynomial time algorithm) then at least we find the similarity between the exponential

Problems, so that one problem is solved then all other problems can be solved. Hence we find the relationship between these problems.

2) When we are unable to write the algorithms for such problems (i.e deterministic polynomial algorithm) then try to write non-deterministic algorithm. (i.e non-deterministic polynomial algorithm)

Let us discussed the non-deterministic polynomial algorithm:

In general the algorithms what we are writing are known as deterministic algorithm. because we know that, how the individual statements are execute and how much time is required for its execution.

But In case of non-deterministic algorithm we don't know how to write and also how it was executed. So the question is how to how to write such algorithm?.

If I am trying to write an non-deterministic algorithm for 0/1 Knapsack or Travelling salesman Problem, then I know the execution process of some statements of the algorithm. but I may not be know the execution process of some lines i.e I may not be figure out

how to make polynomial. So leave such statement so that it can be done in feature by somebody or some researcher.

So, In an non deterministic algorithm maximum ~~number of the~~ statements are deterministic ~~loops~~ but some statements are non-deterministic. In this way what we are doing?, we just preserve the research work for feature. by somebody ~~else~~ with same problem. He may make the deterministic solution for the non-deterministic statements.

Let's see how to write a non-deterministic ~~problem~~ polynomial time problem.

```
Algorithm nSearch (A, n, key)
{
    j = choice();   ---> non deterministic. (O(1))
    if (key = A[j]).  ---> Deterministic
    {
        print(j);   ---> Deterministic. Bo
        success();  ---> non-deterministic. [O(1)
    }
    write(0);   ---> Deterministic.
    failure();  ---> non-deterministic [O(1)
}
```

The above algorithm is used for searching key from an Array A of size n., which required O(1) time for execution. (i.e Constant time).

Scanned by CamScanner

which is better than Binary Search. But this algorithm is non-deterministic Algorithm.

In the NSearch( ), the choice( ) function Return the index of the key. and we assume that this happed in O(1) time. which is a magic for us. So the Statement is non-deterministic Similarly Success() and failure() are the two other non-deterministic Statement in NSearch( ) Algorithm.

So writing a non-deterministic Polynomial time algorithm may be a magic for today but tomorrow it may be a deterministic Polinomial time algorithm. Any Person may work on the non-deterministic Statement and may find the polynomial solution for such Statement.

So like this, we can write down the non-deterministic polynomial time algorithm for exponential time algorithm problem. for this we define two classes of Problem i.e.

1. P. - (set of deterministic Algorithms which takes polynomial time), Examples. linear search, Binary Search, Huffman Code, MST Problem etc.)

Scanned by CamScanner

2. NP : (Non deterministic Polynomial time algorithm. which we want to write for exponential problems like. 0/1 Knapsack, TSP, Sum of Subset, n Queen, Graph Coloring etc.)

The famous diagaram for which show the relationship for between P and NP is given below.

The idea behind this diagram is today if the algorithm is in P class then it must be in NP class Previously.

Let's more formally define the class P and NP.

Complexity class P. : The complexity class P can be defined as follows:

"P is the class of problems for which an algorithm exists that solves instances of size n in time $O(n^c)$ for some constant c."

That is usually referred as a polynomial time algorithm. Problems in this class are considered easy to solve, Some examples are: Linear Search, MST, BFS, Quicksort, etc.

Scanned by CamScanner

Complexity class NP (co-NP): The complexity class NP is based on the time it takes to verify a solution is correct. More formally:

NP is the class of problems A of the following form:

x is a yes-instance of A iff there exists a w such that (x, w) is a yes-instance of B, where B is a decision problem in P regarding pairs (x, w) and where $|w| = poly(|x|)$.

In other words, NP is the class of problems for which $w = O(c^n)$, where w is the number of possible solutions, and n is the size of the problem and where a solution w can be checked in polynomial time. w is known as the "certificate" (at worst all solutions we must be checked, giving exponential running time). The complexity class P is fully contained in the class NP since it takes polynomial time to solve the problem, it also takes polynomial time to verify that the solution is correct. A few examples of class NP problems are: Knapsack, Hamiltonian, Vertex Cover TSP ~~etc~~ etc,

Scanned by CamScanner

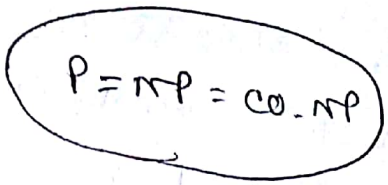
More formally, a Language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

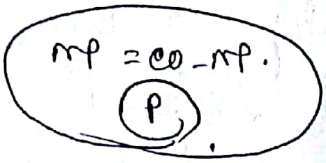
$$L = \{ x \in \{0,1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1 \}.$$

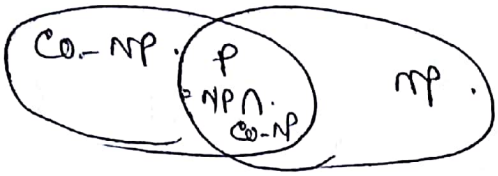
We say that algorithm A verifies language L in polynomial time. Moreover, if $L \in P$, then $L \in NP$, since if there is a polynomial time algorithm to decide L, the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in L. Thus $P \subseteq NP$.

It is unknown whether $P = NP$, but most researchers believe that P and NP are not the same class. ~~Theoretical computer scientists generally believe that~~ Intuitively, the class P consists of problems that can be solved quickly. The class NP consists of problems for which a solution can be verified quickly. From the different researchers point of view, the relationship is presented in next page.

Scanned by CamScanner







Scanned by CamScanner

P vs. NP.

→ One of the great undecided questions in theoretical computer science is whether the class P is a subset of NP or if the classes are equivalent.

→ It is generally believed that P and NP are different but there exists no proof as of now proving that P is equivalent to NP, nor is there a proof that P is a subset of NP.

→ The interesting side effect of this debate which make it so important is if any NP-complete problem can be solved in polynomial time, then All NP-complete problems can be solved in polynomial time by reducing the problem of interest to the problem that had been solved in polynomial time. Therefore P would equal to NP.

NP.-Complete:

Problems that can be verified to be in the class NP but not P are considered to be NP-Complete. The set of NP-complete problems require exponential time to find a solution and polynomial time to check that a solution is correct. For all NP-Complete problems, there exists

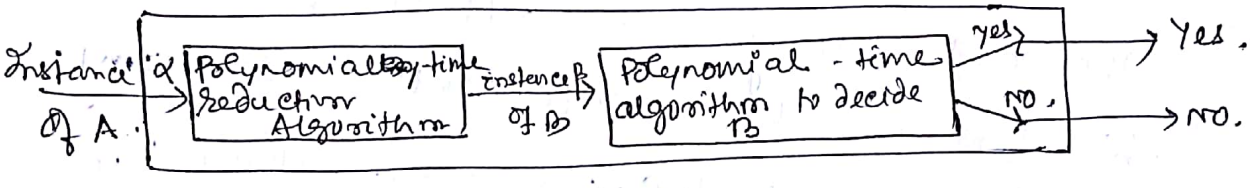
Scanned by CamScanner

an algorithm to convert an instance of that problem to an instance of any other NP-complete problem in polynomial time, called reduction.

Suppose that we have a procedure that transforms any instance α of A into some instance β of B with following characteristics:

→ The transformation takes polynomial time.

→ The answers are the same.
(i.e the answer for α is 'yes' if and only if the answer for β is also 'yes').



The above figure represent how to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, we transfer an instance α of A into an instance β of B, we solve B in polynomial time and we use the answer for β as the answer for α.

This polynomial time reduction Algorithm provides a way to solve problem A in polynomial time:-

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.

Scanned by CamScanner

2. Run the polynomial-time decision algorithm for B on the instance β.

3. Use the answer for β as the answer for α.

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial time factor. i.e if $L_1 \leq_p L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is why the "Less than or equal to" notation for reduction is mnemonic. So the formal definition of NP-complete is is given below.

A Language $L \subseteq \{0,1\}^*$ is NP-complete if

1. $L \in NP$ and
2. $L' \leq_p L$ for every $L' \in NP$.

If a language L satisfies property 2, but not property 1, we say that L is NP-hard.

For example. Let us take two problems.

1. clique and 2. CNF-Satisfiability (Conjunctive Normal form.)

Scanned by CamScanner

If we want to prove that

Clique is NPC

then

we must prove that

2. → Clique is NP-hard. ($3\Phi \leq_P$ Clique)

1. → Clique is in NP. (Clique $\in$ NP.)

Before this we must know that what is CNF-satisfiable and clique problem.

CNF-satisfiable, proble (Conjuctive Normal form).

CNF-satisfiable (3 CNF) example. (one instal.

$= (x_1 \vee x_2 \vee \bar{x_3}) \wedge (x_1 \vee \bar{x_2} \vee x_3) \wedge (\bar{x_1} \vee x_2 \vee x_3)$.

A 3 CNF has K clauses. and each clause has 3 literals (3 CNF).

clause.

i.e $(x \vee y \vee z)$.

literals.

The formula of 3 cnf is given below,

$\Phi = (x_1 \vee x_2 \vee \bar{x_3}) \wedge (x_1 \vee \bar{x_2} \vee x_3) \wedge (\bar{x_1} \vee x_2 \vee x_3)$

now, we need to transform (reduce) $\Phi 3\Phi$ problem to instance of K-clique problem

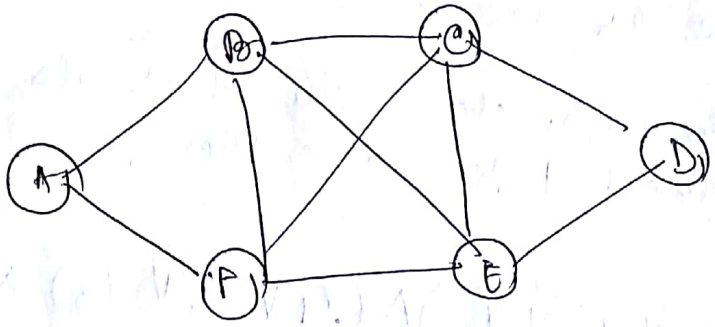
Scanned by CamScanner

Clique Problem:

* A clique, in an undirected graph $G = (V, E)$, is a subset $V' \subseteq V$.
* each pair of which is connected by an edge in E (i.e $e_i \in E$).
* The size of a clique is the no. of vertices it contains.

What is clique problem?

In a clique of a graph, each node is connected to each other nodes.



In the above graph there is a clique of size 4. i.e (B, C, E, F), they connected with each others.

we can verify this with the help of Adjancy Matrix.

So due to the verification in polynomial time this problem clique is in NP.

i.e clique $\in$ NP.

Scanned by CamScanner

now we prove that.

$3\Phi \leq_p$ clique.

for each clause, we have three distinct literals in 3Φ. (i.e $C_r = l_1^r \vee l_2^r \vee l_3^r$ in Φ)

now we convert the 3Φ to clique with the help of following procedure.

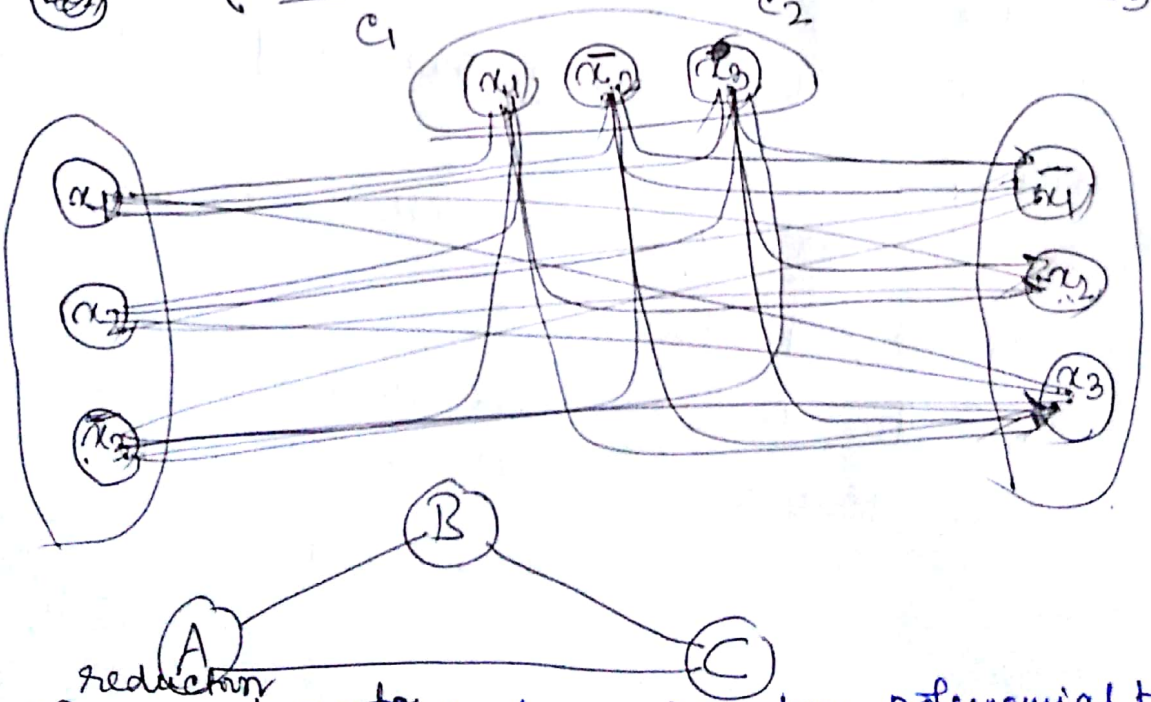
we put edges in Graph G (i.e between $V_i^r \leftrightarrow V_j^s$).

if →

1) V_i^r and V_j^s are in different tripple $r \neq s$
2) Literals are consistent. i.e $l_i^r \neq \overline{l_j^s}$.
3) In each clause the literals are not connected to each other.

$\Phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$

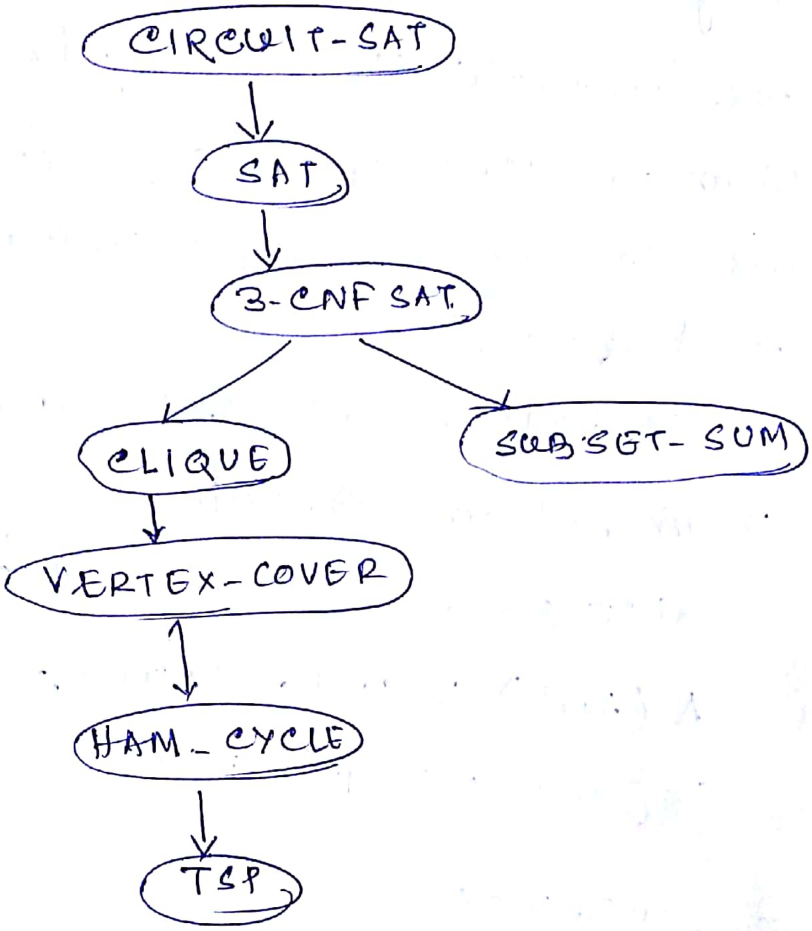
C_1 C_2 C_3.



reduction is done in poly. polynomial time hence $3\Phi \leq_p$ clique.

Scanned by CamScanner

The below figure outlines the structure of the NP-completeness proofs. In this fig, we shall use the reduction methodology to provide NP-completeness proof for a variety of problems drawn from graph theory and set partitioning.



Theorem:

Satisfiability of boolean formulas is NP-complete.

i.e

1. SAT $\in$ NP.
2. CIRCUIT-SAT $\leq_p$ SAT for all CIRCUIT-SAT $\in$ NP.

Scanned by CamScanner

Let's first illustrate the reduction methodology by giving an NP-completeness proof for the problem of determining whether a boolean formula is satisfiable. (This problem has the historical honor of being the problem ever shown to be NP-complete)

We formulate the formula for satisfiability problem in terms of the Language SAT as follows. An instance of SAT is boolean formula ϕ composed of.

1. n boolean variables: $x_1, x_2, \ldots, x_n$;
2. m boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if) and
3. parantheses.

We can easily encode a boolean formula ϕ in length that is polynomial in $n+m$.

A formula with a satisfying assignment is a satisfiable formula. The satisfiability problem asks whether a given boolean formula is satisfiable; in formal language terms,

$SAT = \{\langle\phi\rangle : \phi$ is satisfiable boolean formula$\}$

Scanned by CamScanner

As an example: (the formula)

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment

$(x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1)$ since.

$$\phi = ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0.$$

$$= (1 \vee \neg(1 \vee 1)) \wedge 1$$

$$= (1 \vee 0) \wedge 1$$

$$= 1 \wedge 1 = 1$$

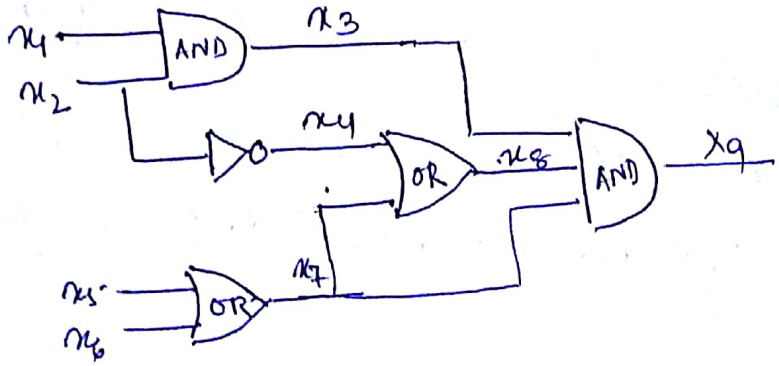
and thus the formula ϕ belong to SAT. (satisfiable)

and achieved in polynomial time.

There are 2^n possibilities if there are n variables, so we can verify in polynomial time. Hence the problem SAT is in NP. i.e (SAT $\in$ NP)

Next we proove SAT is NP-Hard.

i.e CIRSAT $\leq_p$ SAT.



For each wire x_i in the circuit C, the formula ϕ has a variable x_i, Now we express

Scanned by CamScanner

how each gate operates as a small formula involving the variables of its incident wires. For example: the operation of the output AND gate of circuit C is $x_9 \leftrightarrow (x_3 \wedge x_8 \wedge x_7)$. We call each of these small formulas a clause.

The formula ϕ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate. For the circuit C the formula is given below.

$$\phi = x_9 \wedge (x_3 \leftrightarrow (x_1 \wedge x_2)) \wedge (x_4 \leftrightarrow \neg x_2) \wedge (x_7 \leftrightarrow (x_5 \vee x_6)) \wedge (x_8 \leftrightarrow (x_4 \vee x_7)) \wedge (x_9 \leftrightarrow (x_3 \wedge x_8 \wedge x_7))$$

It is just look like formula ϕ. It is straightforward to produce such a formula ϕ for polynomial time. Hence CIRCUIT-SAT $\leq_P$ SAT is proved.

So SAT is NP-complete. Proved

Scanned by CamScanner

Approximation Algorithm.

In real world, many Problems of Practical significance are NP-complete. They are too important to abandon merely because we don't know how to find an optimal solution in polynomial time. We have at least three ways to get around NP-completeness.

First:
If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.

Second:
We may be able to isolate important special cases that we can solve in polynomial time.

Third:
We might come up with approaches to find near-optimal solutions in polynomial time (i.e either worst case or the expected case).

In practice, near optimality is often good enough. An algorithm is called for returning near optimal solutions as an approximation algorithm.

Scanned by CamScanner

So.

An approximation algorithm is a way of dealing with NP-completeness for optimization problems. 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.

Suppose we have some optimization problem instance i, which has a large number of feasible solutions. Also, let.

$C(i)$ ← be the cost of solution produced by the approximation algorithm

$C^*(i)$ ← be the cost of optimal solution.

For minimization problem, we are interested to find a solution of a given instance i in the set of feasible solution, such that $C(i)/C^*(i)$ be as small as possible.

For maximization problem, we are interested to find a solution of a given instance i in the set of feasible solution, such that $C^*(i)/C(i)$ be as small as possible.

Scanned by CamScanner

We say that an approximation algorithm for a given problem instance i, has a ratio bound of $p(n)$ if for any input size n, the cost c of the solution produced by the approximation algorithm is within a factor of $p(n)$ of the cost c^* of an optimal solution. i.e.

$$\max\left(\frac{c(i)}{c^*(i)}, \frac{c^*(i)}{c(i)}\right) \leq p(n)$$

(note that: $p(n)$ is always ≥ 1).

For minimization problem, $0 < c^*(i) \leq c(i)$ and the ratio $c(i)/c^*(i)$ gives the factor by which the cost of the approximation solution is larger than the cost of optimal solution. Similarly, for maximization problem $0 < c(i) \leq c^*(i)$ and the ratio $c^*(i)/c(i)$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

Vertex Cover Problem.

Definition: A vertex cover of an undirected graph $G(V, E)$ is a subset of $V' \subseteq V$ such that if edge (u, v) is an edge of G then either u in V' or v in V' (or both).

Scanned by CamScanner

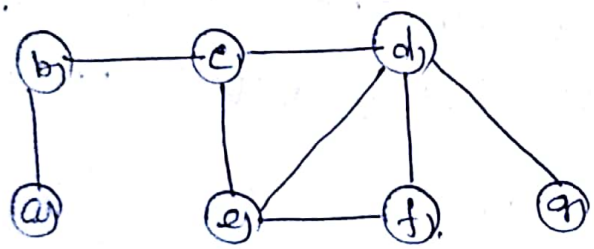
Problem: Find a vertex cover of maximum size in a given undirected graph.

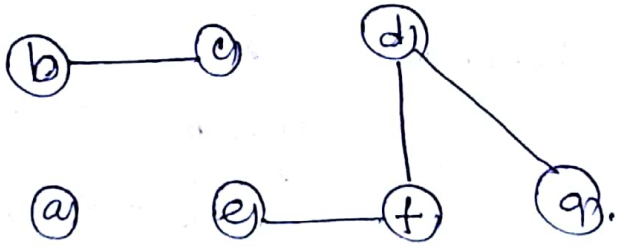
This optimal vertex cover is the optimization version of an NP-complete problem but it is not too hard to find a vertex-cover that is near optimal.

APPROX-VERTEX-COVER (G: Graph)

1. $C \leftarrow \{\}$
2. $E' \leftarrow E[G]$.
3. while E' is not empty do.
4. Let (u, v) be an arbitrary edge of E'
5. $C \leftarrow C \cup \{u, v\}$
6. Remove from E' every edge incident on either u or v.
7. Return C.

Example:

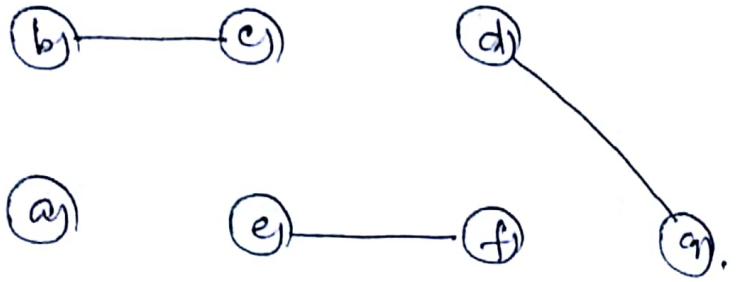




Select {b, c}

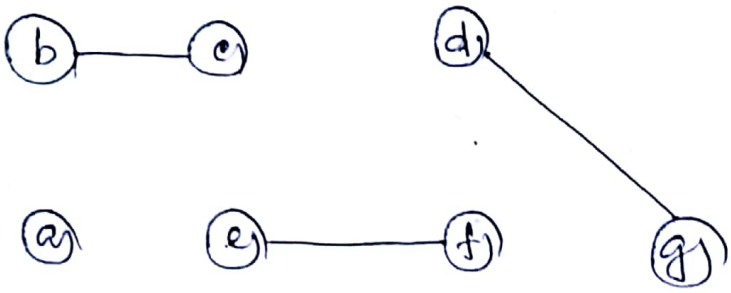
So remove {b, a} {c, e} {c, d}
(step-6)

Scanned by CamScanner



now select {e, f}

then remove {e d} & {d, f}



select {d, g}.

Hence $C = \{\{b, c\}, \{e, f\}, \{d, g\}\}$.

Analysis

It is easy to see that the running time of this algorithm is $O(V+E)$, using adjacency list to represent E'.

Travelling Salesman Problem:

Problem: Given a complete undirected graph $G = \langle V, E \rangle$ that has non-negative integer cost $c(u, v)$ associated with each edge (u, v) in E, the problem is find a Hamiltonian cycle tour of G with minimum cost.

Scanned by CamScanner

Approx-TSP-Tour (G, c)

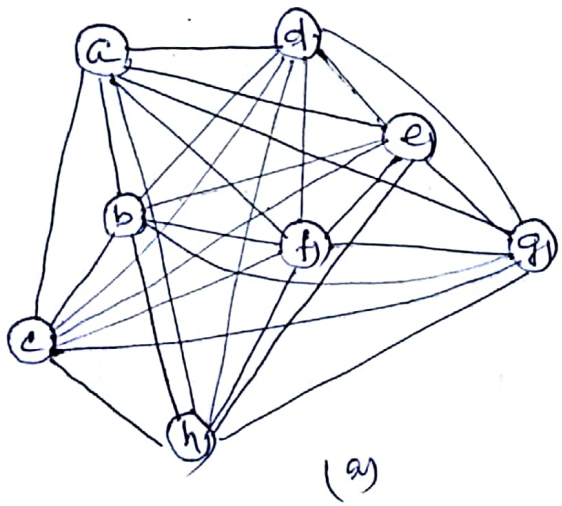
[Note: The parameter G is a complete undirected Graph, and the cost function c satisfies the triangle inequality. i.e

$\forall$ vertex $u, v, w \in V$

$c(u, w) \leq c(u, v) + c(v, w)$.]

1. Select a vertex $r \in G.(V)$ to be a "root" vertex
2. Compute a minimum Spanning tree T for G from root r. using MST-Prime (G, c, r)
3. let H be a list of vertices, ordered according to when they are visited in a preorder tree walk of T.
4. Return the hamiltonian cycle H.

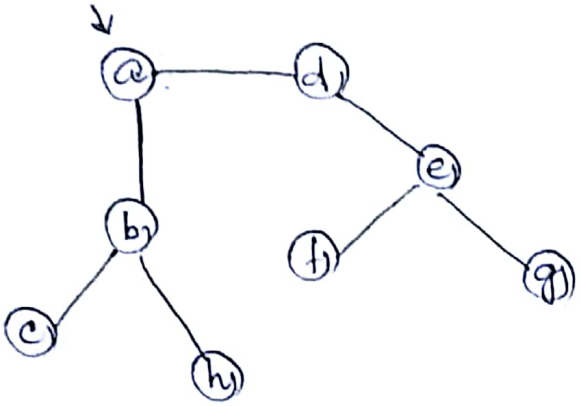
The Example below illustrate the operation of Approx-TSP-Tour.



(a)

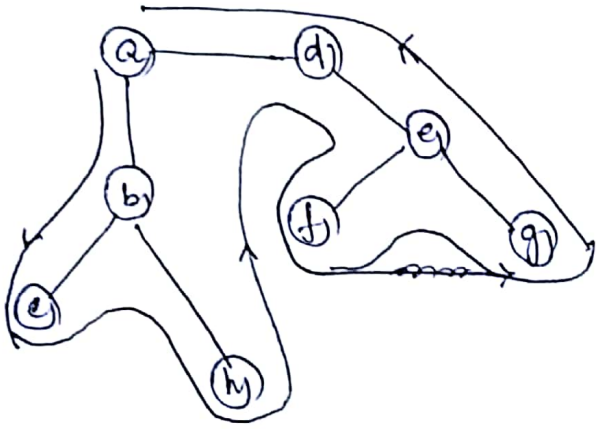
← fig (a) shows a complete undirected graph.

Scanned by CamScanner



(b)

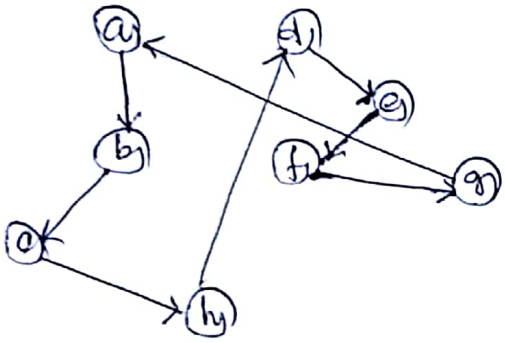
fig (b) shows the minimum spanning tree T grown from root vertex a by MST-Prime.



(c)

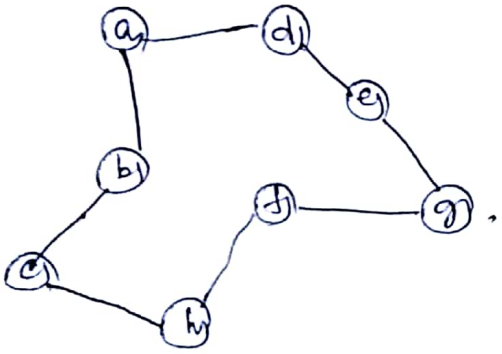
fig (c) shows how a preorder walk of T visits the vertices. The walk order is

a → b → c → b → h → b → a → d → e → g → e → f → e



(d)

fig (d) shows the corresponding tour, which is the tour returned by Approx-TSP-Tour.



(e)

fig (e), displays an optimal tour, which is about 23% shorter.

The running time of Approx-TSP-Tour is $O(E \lg V)$.

Scanned by CamScanner

Randomization

When the average case of quicksort algorithm was observed, then an assumption have to made by us that all permutations of the input numbers are equally likely. But in engineering we cannot always expect this assumptions to hold.

Hence randomization was created as tools for selecting an input. by making the behaviour of part of the problem, and play a major role for design and analyses of algorithm.

More generally, we call an algorithm randomized if its behaviour is determined not only by its input but also by values produced by a random-number generator. Lets execute randomization concept with Quicksort.

Algo:

Randomized-Partition (A, p, r)

1. i = Random (p, r)
2. exchange A[r] with A[i]
3. return Partition (A, p, r)

Scanned by CamScanner

Randomized – Quicksort (A, p, r)

1. if p < r
2. q = Randomized-Partition(A, p, r)
3. Randomized – Quicksort (A, p, q-1)
4. Randomized – Quicksort (A, q+1, r)

Partition (A, p, r)

1. x = A[r]
2. i = p-1
3. for j = p to r-1
4. if A[j] ≤ x
5. i = i+1
6. exchange A[i] with A[j]
7. exchange A[i+1] with A[r]
8. Return (i+1)

Analysis:

The worst case complexity – $\Theta(n^2)$

Expected Running time – $O(n \lg n)$

Scanned by CamScanner

Polynomials and FFT.

The addition of two polynomial equation of degree n takes $\Theta(n)$ execution time and the multiplication takes $\Theta(n^2)$ time. Let's see how FFT (Fast Fourier series) reduce the polynomial time from $\Theta(n^2)$ to $\Theta(n \lg n)$.

Due to the less time complexity of polynomial multiplication i.e Fourier Transforms, FFT is for signal processing. A signal is given in the time domain: as a function mapping time to amplitude. The Fourier Analysis allows us to express the signal as a weighted sum of phase-sinusoids of varying frequencies. The weight and phases associated with the frequencies characterize the signal in the frequency domain.

In day to day application FFT is used as compression techniques to encode digital video and audio information, including MP3 files.

What is polynomial?

A polynomial in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum:

Scanned by CamScanner

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

where a_j ← coefficient of the polynomial.

We can define a variety of operations on polynomials. For polynomial addition, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their sum is a polynomial $C(x)$ also of degree-bound n, such that $C(x) = A(x) + B(x)$ for all x in the underlying field. That is if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and $$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

then $$C(x) = \sum_{j=0}^{n-1} c_j x^j$$

where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$.

For example:

$$A(x) = 6x^3 + 7x^2 - 10x + 9$$

$$B(x) = -2x^3 + 4x - 5$$

$$C(x) = 4x^3 + 7x^2 - 6x + 4$$

(note. $A(x)$ → has degree 3.

$A(x)$ → has degree-bounds 4, 5, 6 ... or all values > degree

$A(x)$ → has coefficients (9, −10, 7, 6)

Similarly for $B(x)$)

Scanned by CamScanner

Similarly For Polynomial Multiplication

If $A(x)$ and $B(x)$ are polynomials of degree bound n, their product $C(x)$ is a polynomial of degree bound $2n-1$ such that $C(x) = A(x) \cdot B(x)$. for all x in the underlying field. In polynomial multiplication each term of $A(x)$ is multiply with each terms of $B(x)$ and then combining the terms with equal powers. For example,

$$A(x) = 6x^3 + 7x^2 - 10x + 9$$

$$B(x) = -2x^3 + 4x - 5$$

$$-30x^6 - 35x^5 + 50x^4 - 45x^3$$

$$+36x^4 + 28x^3 - 40x^2 + 36x$$

$$-12x^6 - 14x^5 + 20x^4 - 18x^3$$

$$-12x^6 - 14x^5 + 44x^4 - 20x^3 - 35x^2 + 86x - 45$$

Another way to express the product $C(x)$ is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \quad \ldots (1)$$

(note: degree (C) = degree (A) ⊕ degree (B), implies that A is polynomials of degree bound n_a

Scanned by CamScanner

and B is a polynomial of degree-bound n_b, then C is a polynomial degree bound $n_a + n_b - 1$. Since a polynomial degree bound k is also a polynomial of degree-bound k+1, then we will normally say that the product polynomial C is a polynomial of degree bound $n_a + n_b$.

Polynomial Representation.

A polynomial can be represented in two ways.

1) Coefficient Representation.
2) Point value Representation.

1.) Coefficient Representation

A coefficient representation of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree bound n is a vector of coefficients $a_j = (a_0, a_1, \ldots a_{n-1})$. The operation of evaluating polynomial A(x) at point x_0 consists of computing the value of $A(x_0)$, using Horners method

2) Point-value Representation

A point value representation of a polynomial A(x) of degree-bound n is a set of n point-value pairs.

$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$.

Scanned by CamScanner

such that all of the x_k are distinct and

$y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$.

A polynomial has many different point-value representations. So we can use any set of n distinct points $x_0, x_1, \ldots x_{n-1}$ as a basis for the representation.

for example: $A(x) = x^3 - 2x + 1$

x_k	0	1	2	3
$A(x_k)$	1	0	5	22

$\{(0,1), (1,0), (2,5), (3,22)\}$.

using horner's method n-point evaluation takes time $\Theta(n^2)$.

The converse of evaluation is called interpolation.

- determines coefficient form of polynomial from point value representation.

→ For any set $\{(x_0, y_0), (x_1, y_1) \ldots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$

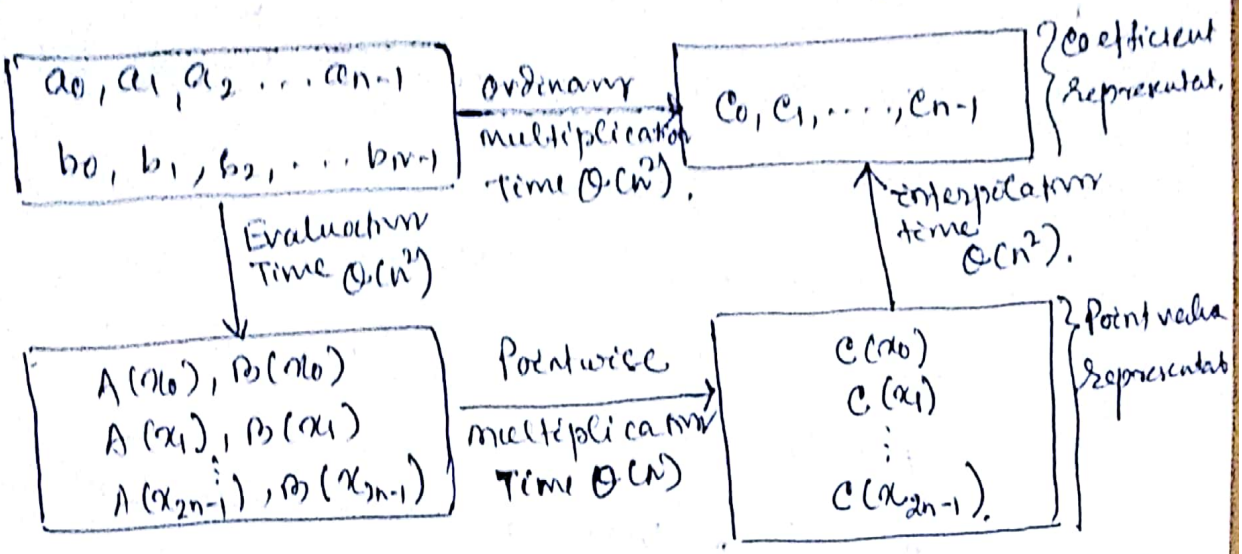
→ Lagarange's formula.

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

Scanned by CamScanner

Using Lagrange's formula, interpolation takes time $\Theta(n^2)$.

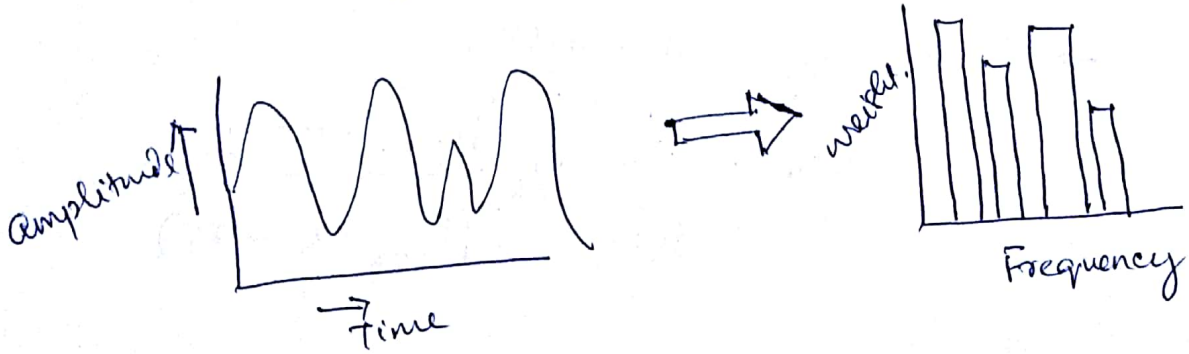
Graphical outline of efficient polynomial multiplication.



By using FFT and its inverse, we can compute the evaluation and interpolation in time $\Theta(n \lg n)$. The product of two polynomials of degree-bound n can be computed for time $\Theta(n \lg n)$, with both input and output in coefficient form.

Fourier transforms originates from Signal Processing.

→ Transform signal from the time domain to frequency domain.



Scanned by CamScanner

→ I/p signal is a function mapping time to amplitude
→ output is a weighted sum of phase-shifted sinusoids of varying frequencies.

Algorithm for Fast Multiplication of Polynomials.

1. Add n higher order zero coefficients to A(x) and B(x).
2. Evaluate A(x) and B(x) using FFT for 2n points.
3. Pointwise multiplication of point-value forms.
4. Interpolate C(x) using FFT to compute inverse DFT.

The FFT method employs a divide and conquer strategy, using the even-indexed and odd-indexed coefficients of A(x) separately to defined the two new polynomials $A^0(x)$ and $A^1(x)$ of degree-bound $n/2$

i.e

$$A^0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$

$$A^1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

A^0 contains all even-indexed coefficients of A and A^1 contains all odd-indexed coefficients.

Scanned by CamScanner

Recursive FFT (a)

1. n = a.length
2. if n == 1
3. return a.
4. $\omega_n = e^{2\pi i/n}$ [principal nth root of unity].
5. $\omega = 1$ [complex number]
6. $a^{[0]} = (a_0, a_2, a_4 \ldots, a_{n-2})$
7. $a^{[1]} = (a_1, a_3, a_5, \ldots, a_{n-1})$.
8. $y^{[0]}$ = Recursive-FFT ($a^{[0]}$).
9. $y^{[1]}$ = Recursive-FFT ($a^{[1]}$).
10. for k = 0 to n/2 - 1
11. $y_k = y_k^{[0]} + \omega y_k^{[1]}$
12. $y_{k+(n/2)} = y_k^{[0]} - \omega y_k^{[1]}$
13. $\omega = \omega \omega_n$
14. return y.

The recursive-FFT procedure works as follows,

line 2-3 :- represents the base of the recursions.

Line 6-7 ← define the coefficient vectors for the polynomial $A^{[0]}$ and $A^{[1]}$

Line 4, 5, & 13 ← is responsible for properly updation of ω i.e complex no.) so that coherence line no. 11–12 are executed we have $\omega = \omega_n^k$ (i.e keeping the running value of ω from iteration to iteration

Line-8,9 → Performs the recursive FFT

Scanned by CamScanner