B.TECH. TCS-503

FIFTH SEMESTER THEORY EXAMINATION 2009-10

DESIGN AND ANALYSIS OF ALGORITHMS

Time: 3 Hours *Total Marks: 100*

Note: *(i) Attempt **all** questions.*

(ii) All parts of a question should be attempted at one contiguous place.

1. Attempt any four parts of the following: (5 × 4 = 20)

(a) Solve the recurrence relation using master method:

$$T(n) = 3T(n^{1/3}) + \log 3^n$$

Ans. Given equation is

$$T(n) = 3T(n^{1/3}) + \log 3^n$$

Let $m = \log_3 n$

$3^m = n$ ...(1)

Now $3^{m/3} = n^{1/3}$ ...(2)

Put these values in the main equation.

$$T(3^m) = 3T(3^{m/3}) + m$$

Rename $T(3^m)$ as $\delta(m)$ and $T(3^{m/3})$ as $\delta\left(\frac{m}{3}\right)$

$$\delta(m) = 3\delta\left(\frac{m}{3h}\right) + m$$

Compare with $\delta(m) = a\delta\left(\frac{m}{f}\right) + f(m)$

here $a = 3, f = 3, f(m) = m$

find $m^{(\log_a f)} = m^{\log_3 3} = m$

here $m^{(\log_a f)} = f(m)$ thus case 2 exist.

So the solution is $\theta(m \log m)$

$\theta(\log_3 n \log \log_3 n)$

(b) What do you understand by 'stable' sort? Name two stable sort algorithms.

Ans. Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say *R* and *S*) with the same key, and *R* appears before *S* in the original list, then *R* will always appear before *S* in the sorted list. When equal elements are distinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue.

Example of stale sort algorithm are:

Counting Sort

Radix Sort

(c) Prove that Heapsort and Mergesort are optimal comparison sorting algorithms.

Ans. Consider a decision tree of higher h with l reachable leaves corresponding to a comparison sort of n elements. Because of each of the $n!$ permutations of the input appears as some leaf,

we have $n! \leq l$

Since a binary tree of height h has no more that 2^h leaves, we have

$$n! \leq l \leq 2^h$$

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

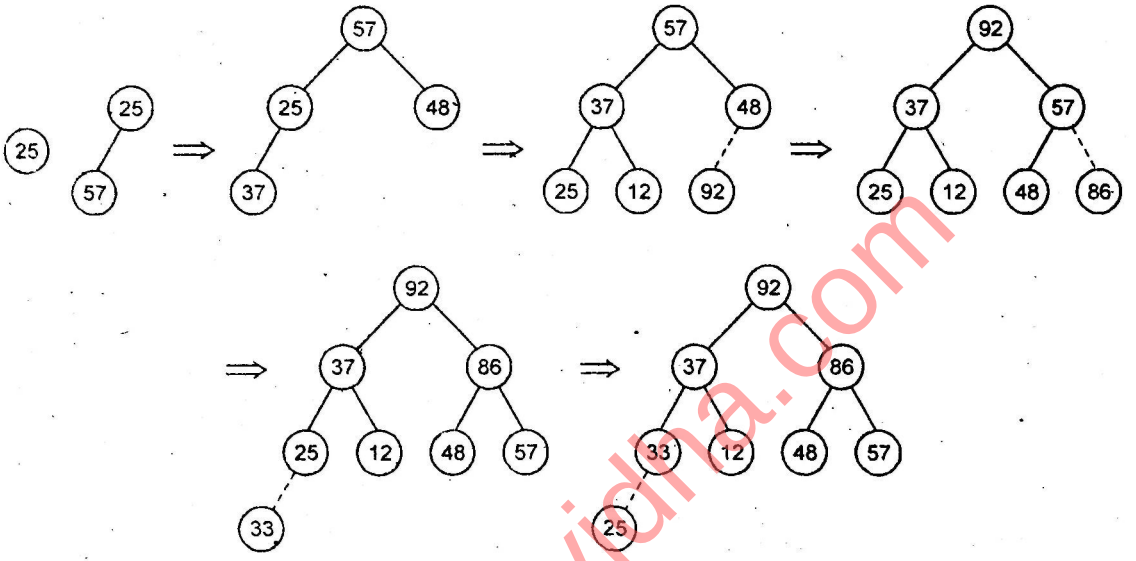
$$h = \Omega(nlgn)$$

The $O(nlgn)$ upper bounds on the running times for heapsort and merge sort mater the $\Omega(nlgn)$ worst care lever bound.

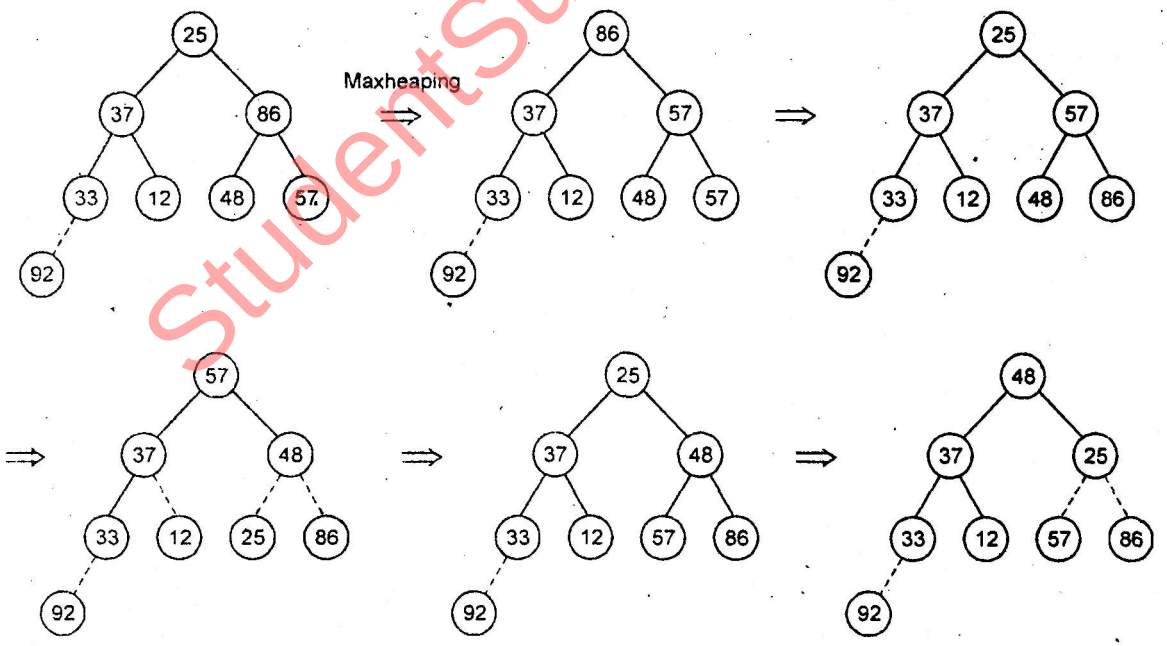
(*d*) **Illustrate the functioning of Heap sort on the following array:**

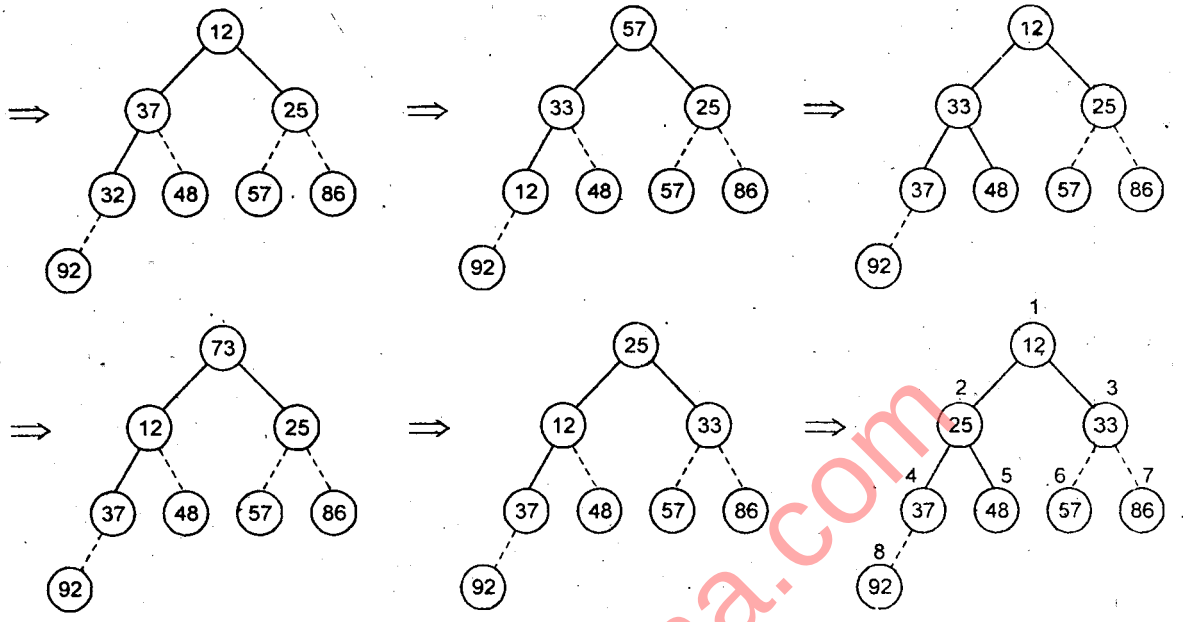
$$A = \langle 25, 57, 48, 37, 12, 92, 86, 33 \rangle$$

Ans. $A = \langle 25, 57, 48, 37, 12, 92, 86, 33 \rangle$



Apply Heap sort:





(e) How can you modify quicksort algorithm to search an item in a list of elements?

Ans. We can use randomized select algorithm to search an item in the list of elements which require little modification in quick sort algorithm.

Randomized-select only works on one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected time of *Randomized-select* is $\Theta(n)$.

Randomized-select uses the procedure *Randomized-partition* introduced in. Thus, like *Randomized-Quicksort*, it is randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The following code for *Randomized-select* returns the ith smallest element of the array $A[P...r]$.

Randomized-select (A, p, r, i).

1. If $p = r$
2. then return $A[p]$
3. $q \leftarrow$ *Randomized-partition* (A, p, r)
4. $k \leftarrow q - p + 1$
5. If $i = k$, the pivot value is the answer
6. then return $A[q]$
7. elseif $i < k$
8. then return *Randomized-select* $(A, p, q - 1, i)$
9. else return Randomized-select $(A, q + 1, r, i - k)$

(f) What is the importance of 'average-case analysis' of algorithms?

Ans. Worst-case performance analysis and average case performance analysis have some similarities, but in practice usually require different tools and approaches. Determining what average input means is difficult, and often that average input has properties which make it difficult to characterise mathematically (consider, for instance, algorithms that are designed to operate on strings of text). Similarly, even when a sensible description of a particular "average case" (which will probably only be applicable for some uses of the algorithm) is possible, they tend to result in more difficult to analyse equations. Worst-case analysis has similar problems: it is typically i, possible to determine the exact worst-case scenario.

Instead, a scenario is considered such that it is at least as bad as the worst case. For example, when

analysing an algorithm, it may be possible to find the longest possible path through the algorithm (by considering the maximum number of loops, for instance) even if it is not possible to determine the exact input that would generate this path (indeed, such an input may not exist). This gives a safe analysis (the worst case is never underestimated), but one which is pessimistic, since there may be no input that would require this path. Alternatively, a scenario which is thought to be close to (but not necessarily worse than) the real worst case may be considered. This may lead to an optimistic result, meaning that the analysis may actually underestimate the true worst case. In some situations, it may be necessary to use a pessimistic analysis in order to guarantee safety. Often however, a pessimistic analysis may be too pessimistic, so an analysis that gets closer to the real value but may be optimistic (perhaps with some known low probability of failure) can be a much more practical approach. When analyzing algorithms which often take a small time to complete, but periodically require a much larger time, amortized analysis can be used to determine the worst-case running time over a (possibly infinite) series of operations. This *amortized worst-case* cost can be much closer to the average case cost, while still providing a guaranteed upper limit on the running time.

2. Attempt any four parts of the following: **(5 × 4 = 20)**

(a) Two stacks are kept in a single array STK [MAX] to utilize the array memory optimally: STK [ ]:

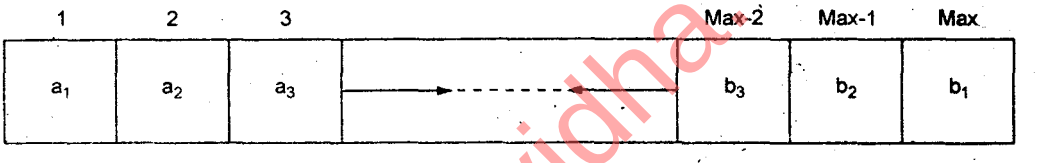


Fig. 1

First stack grows in forward direction from start whereas second grows backwards from end. Write PUSH 1, PUSH 2, POP 1, POP 2 for the two stacks.

Ans. PUSH1(STK, TOP1, TOP2, MAX, ITEM)

STEP 1: IF TOP1 + 1 = TOP2 OR TOP 1 = MAX THEN
WRITE "OVERFLOW"
RETURN

STEP 2: IF TOP 1 = 0 AND TOP 2 = 1 THEN
WRITE "OVER FLOW"
RETURN

STEP 3: IF TOP 1 = 0 THEN SET TOP 1 = 1
ELSE
SET TOP1 = TOP1 +1
SET STK[TOP1] = ITEM
RETURN

STEP 4: END

PUSH 2 (STK, TOP1, TOP2, MAX, ITEM)

STEP 1: IF TOP 2 – 1 = TOP1 OR TOP2 = 1 THEN
WRITE "OVERFLOW"
RETURN

STEP 2: IF TOP2 = 0 AND TOP1 = MAX THEN
WRITE "OVERFLOW"
RETURN

STEP 3: IF TOP2 = 0 THEN
SET TOP2 = MAX
ELSE
SET TOP2 = TOP2 – 1
SET STK[TOP2] = ITEM
RETURN

STEP 4: END

POP1 (STK, TOP1, TOP2)

STEP 1: IF TOP1 = 0 THEN

WRITE "UNDERFLOW"

RETURN

STEP 2: SET ITEM = STK[TOP1]

IF TOP1 = 1 THEN

SET TOP1 = 0

ELSE SET TOP1 = TOP1 – 1

RETURN ITEM

STEP 3: END

POP2 (STK, TOP1, TOP2)

STEP 1: IF TOP2 = 0 THEN

WRITE "UNDERFLOW"

RETURN

STEP 2: SET ITEM = STK [TOP2]

IF TOP2 = MAX THEN

SET TOP2 = 0

ELSE SET TOP2 = TOP2 + 1

RETURN ITEM

STEP 3: END

(b) Define Red-black trees and state their applications.

Ans. A binary search tree is a red-black tree if it satisfies the following *red-black* properties:

1. Root node is always black.
2. Every node is either red or black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Application of red Black tree

Various applications of RB tree are as follows:

1. Red Black Tree is a special type of self balancing binary search tree. This is used as Syntax Trees in major compilers and as implementations of Sorted Dictionary.
2. Red Black tree can be augmented to create order statistic tree.
3. Red Black tree can be augmented to create interval search tree.
4. Order statistic tree constructed from the RB tree can be used to search in *lgn* time.

(c) Prove that the maximum degree of any node in a n-node binomial tree is log *n*.

Ans. For the binomial tree *Bk*,

1. There are 2^k nodes,
2. The height of the tree is *k*,
3. There are exactly nodes at depth *i* for *i* = 0, 1, ..., *k*, and
4. The root has degree *k*, which is greater than that of any other node; moreover if *i* the children of the root are numbered from left to right by *k* – 1, *k* – 2,...,0, child *i* is the root of a subtree *Bi*.

Let *n* be the total no. of nodes and *k* be the degree of the binomial tree B_k then the degree of the root node is maximum which is equal to *k*.

As from the properties of binomial tree, the binomial tree B_k has 2^k nodes.

Thus $n = 2^k$

Hence $k = \log_2 n = lgn$

(d) What is a disjoint-set data structure? How running times of disjoint set data structures is analyzed?

Ans. Disjoint set data structure: *A disjoint-set data structure* maintains a collection of disjoint dynamic sets. Each set is identified by a *representative*, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; we only care that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times. In other applications, there may be a prespecified rule for choosing the representative, such as choosing the smallest member in the set (assuming, of course, that the elements can be ordered). As in the other dynamic-set

implementations we have studied, each element of a set is represented by an object. Letting x denote an object, we wish to support the following operations:

- **Make-set (x)** creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.
- **Union (x, y)** unites the dynamic sets that contain x and y, say Sx and Sy, into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is any member of $Sx \_ Sy$, although many implementations of *union* specifically choose the representative of either Sx or Sy as the new representative. Since we require the sets in the collection to be disjoint, we "destroy" sets Sx and Sy, removing them from the collection.
- **Find-set (x)** returns a pointer to the representative of the (unique) set containing x.

Analysing the running time: We start by computing, for each object in a set of size n, an upper bound on the number of times the object's pointer back to the representative has been updated. Consider a fixed object x. We know that each time x's representative pointer was updated, x must have started in the smaller set. The first time x's representative pointer was updated, therefore, the resulting set must have had at least 2 members. Similarly, the next time x's representative pointer was updated, the resulting set must have had at least 4 members. Continuing on, we observe that for any $k \le n$, after x's representative pointer has been updated $[\lg k]$ times, the resulting set must have at least k members. Since the largest set has at most n members, each object's representative pointer has been updated at most $[\lg n]$ times over all the *union* operations. We must also account for updating the *head* and *tail* pointers and the list lengths, which take only (1) time per *union* operation. The total time used in updating the n objects is thus $O(n \lg n)$.

The time for the entire sequence of m operations follows easily. Each MAKE-SET and FINDSET operation takes $O(1)$ time, and there are $O(m)$ of them. The total time for the entire sequence is thus $O(m + n \lg n)$.

(e) Show the results of inserting the keys:

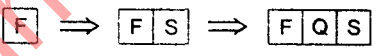
F, S, Q, K, C, L, H, T, V, W, M, R, N

in order into an Empty B-tree with minimum degree 2.

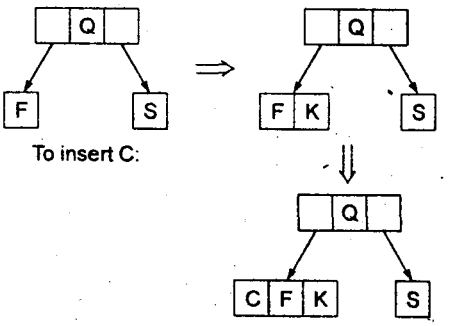
Ans. *F, S, Q, K, C, L, H, T, V, W, M, R, N*

Minimum no. of keys at a node = $t - 1 = 1$.

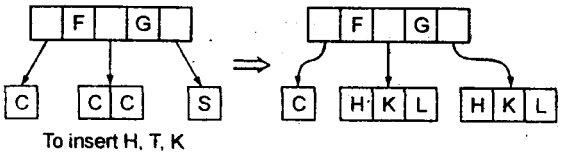
Maximum no. of keys at a node = $2t - 1 = 3$



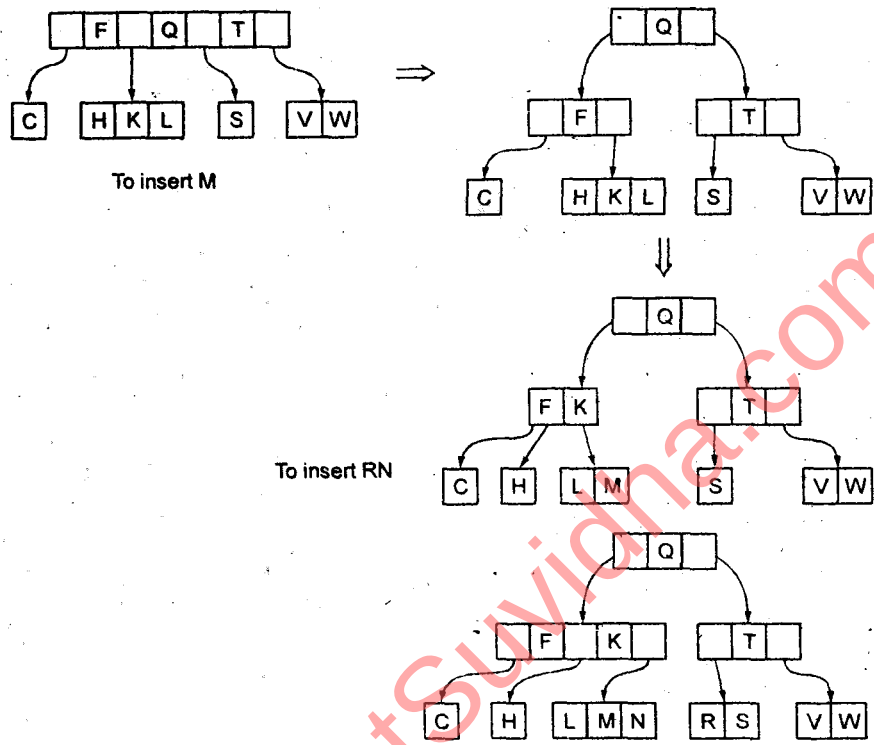
To insert node C: Since node is full then before inserting k, the root node must be split.



To insert L: Node is full and must be split. The nuclein by a will more to parent node.



To insert w = Note is full there before mining at node must be split ans middle by more lop.



(f) What is implied by augmenting a data-structure? Explain with an example.

Ans. Augmenting a data structure can be broken into four steps:

1. Choosing an underlying data structure,
2. Determining additional information to be maintained in the underlying data structure,
3. Verifying that the additional information can be maintained for the basic modifying operations on the underlying data structure, and
4. Developing new operations.

Example: Interval Tree

We can augment red-black tree to support operations on dynamic sets of intervals. A *closed interval* is an ordered pair of real numbers $[t_1, t_2]$, with $t_1 \le t_2$. The interval $[t_1, t_2]$ represents the set $\{t - R: t_1 \le t \le t_2\}$. *Open* and *half-open* intervals omit both or one of the endpoints from the set, respectively. In this section, we shall assume that intervals are closed; extending the results to open and half-open intervals is conceptually straightforward. Intervals are convenient for representing events that each occupy a continuous period of time. We might, for example, wish to query a database of time intervals to find out what events occurred during a given interval. The data structure in this section provides an efficient means for maintaining such an interval database.

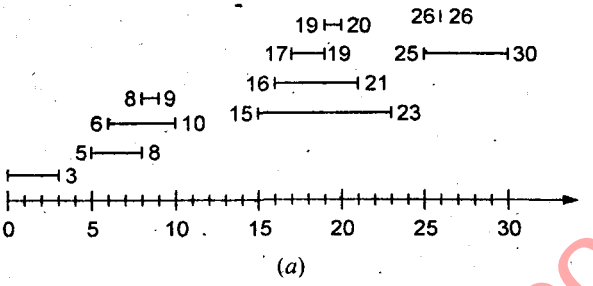
We can represent an interval $[t_1, t_2]$ as an object i, with fields low $[i] = t_1$ (the *low endpoint*) and *high* $[i] = t_2$ (the *high endpoint*). We say that intervals i and i' *overlap* if $i \cap i' \neq \phi$, that is, if low$[i] \le$ high $[i']$ and *low*$[i'] \le$ high$[i]$. Any two intervals i and i' satisfy the *interval trichotomy*; that is, exactly one of the following three properties holds:

a. i and i' overlap,

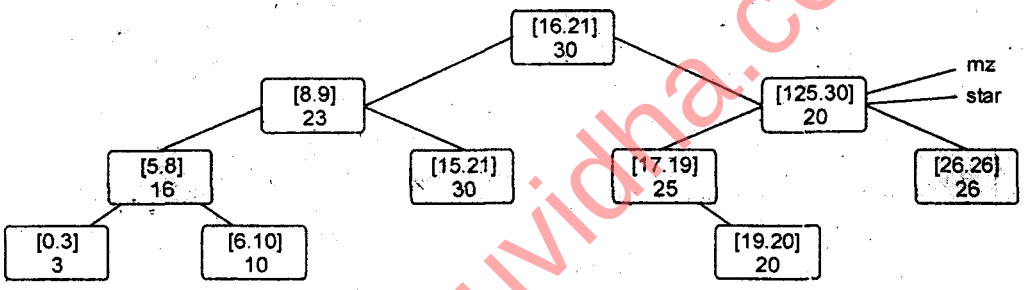
b. i is to the left of i'(i.e., high[i] < low[i']),

c. i is to the right of i'(i.e., high[i'] < low[i])

Figure shows the three possibilities.



(*a*)



(*b*)

Step 1: Underlying Data Structure

We choose a red-black tree in which each node x contains an interval *int*[x] and the key of x is the low endpoint, low[*int*[x]], of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

Step 2: Additional Information

In addition to the intervals themselves, each node x contains a value *max*[x], which is the maximum value of any interval endpoint stored in the subtree rooted at x.

Step 3: Maintaining the Information

We must verify that insertion and deletion can be performed in $O(lgn)$ time on an interval tree of n nodes. We can determine max[x] given interval *int*[x] and the *max* values of node x's children: max[x] = max(high[*int*[x]], *max*[*left* [x]], *max*[*right* [x]]).

Step 4: Developing New Operations

The only new operation we need in Interval-search (T, i), which finds a node in tree T whose interval overlaps interval i. If there is no interval that overlaps i in the tree, a pointer to the sentinel *nil*[T] is returned.

3. Attempt any two parts of the following:

(10 × 2 = 20)

(a) When and how Dynamic Programming approach is applicable?

Discuss the matrix-chain multiplication with respect to Dynamic programming technique.

Ans. Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. The divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when 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 subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

Dynamic programming is typically applied to *optimization problems*. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Matrix-chain multiplication problem

The matrix-chain multiplication problem can be stated as follows: given a chain $\{A_1, A_2, ...A_n\}$ of n matrices, where for $i = 1, 2, ..., n$, matrix Ai has dimension $p_{i-1} \times p_i$, fully parenthesize the product A_1, A_2, A_n in a way that minimizes the number of scalar multiplications.

Step 1: The structure of an optimal parenthesization

For the matrix-chain multiplication problem, we can perform this step as follows.

For convenience, let us adopt the notation A_{ij}, where $i \le j$, for the matrix that results from evaluating the product $A_i A_{i+1} A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then any parenthesization of the product $A_i A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, we first compute the matrices A_{i-k} and A_{k+1-j} and then multiply them together to produce the final product A_{i-j}. The cost of this parenthesization is thus the cost of computing the matrix A_{i-k}, plus the cost of computing A_{k+i-j}, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows.

Suppose that an optimal parenthesization of $A_i A_{i+1} A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the "prefix" subchain $A_i A_{i+1} A_k$ within this optimal parenthesization of $A_i A_{i+1} A_j$ must be an optimal parenthesization of $A_i A_{i+1} A_k$.

Step 2: A recursive solution

We can give the recursive solution to the equation as follows:

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

where $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix A_{ij}.

Step 3: Computing the optimal costs

We can compute the optimal cost with the help of the following algorithm:

MATRIX-CHAIN-ORDER(p)

1. $n \leftarrow length[p] - 1$
2. for $i \leftarrow 1$ to n
3. do $m[i, i] \leftarrow 0$
4. for $l \leftarrow 2$ to $n - l$ is the chain length.
5. do for $i \leftarrow 1$ to $n - l + 1$
6. do $j \leftarrow i + l - 1$
7. $m[i, j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j - 1$
9. do $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
10. if $q < m[i, j]$

11. then $m[i, j] \neg q$

12. $s[i, j] \leftarrow k$

13. return m and s

Step 4: Constructing an optimal solution

We can construct an optimal solution as follows:

Print-optimal-parens (s, i, j)

1. If $i = j$

2. then print "A"i

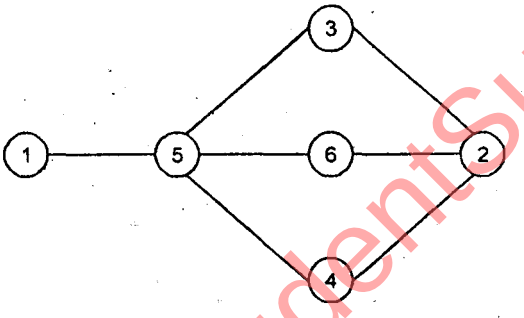
3. else print "("

4. Print-optimal-parens (s, i, $s[i, j]$)

5. Print-optimal-parens(s, $s[i, j] + 1$, j)

6. print ")"

(b) What is "Greedy algorithm"? Write its pseudo code. Apply greedy algorithm on coloring the vertices of the following graph.



Ans. Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. *A greedy algorithm* always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Elements of the greedy strategy

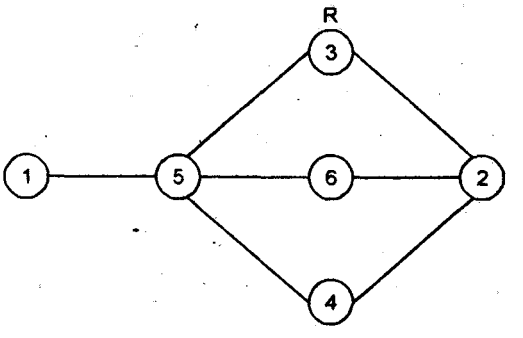
The various elements of the greedy strategy are

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
4. Show that all but one of the subproblems induced by having made the greedy choice are empty.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

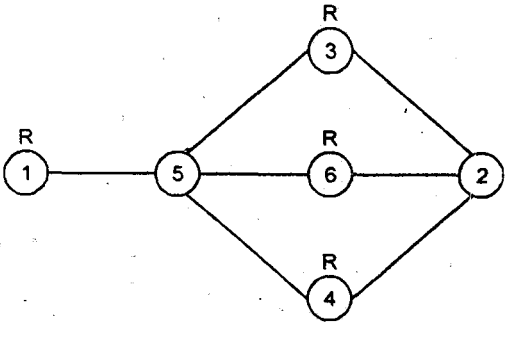
Applying greedy algorithm to the given graph

We can apply the greedy algorithm to the given graph as follows:

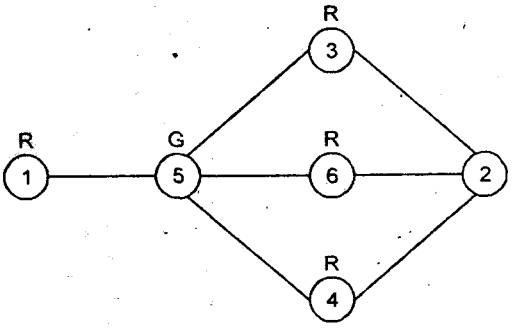
Step 1: First we color any of the vertices with a color say, Red



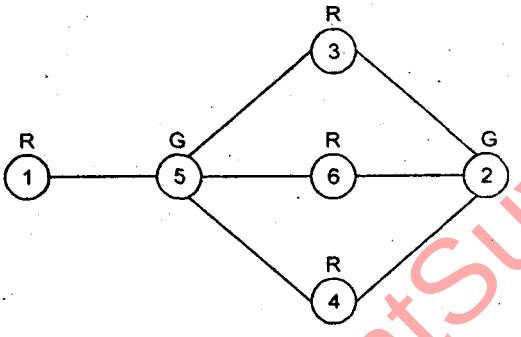
Step 2: Now all the remaining vertices that can be colored with red are as:



Step 3: Now we select one of the remaining vertices and color it with a color say, Green



Step 4: Now the last vertex can be colored with green color as given below



Hence, when we apply the greedy algorithm we find that only 2 colors are needed to color the graph

(c) Discuss backtracking problem solving approach with the help of an example.

Ans. *Backtracking* is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate *c*("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.

Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, and many other puzzles. It is often the most convenient (if not the most efficient) technique for parsing, for the knapsack problem and other combinatorial optimization problems. It is also the basis of the so called logic programming languages such as Icon, Planner and Prolog. Backtracking is also utilized in the (diff) difference engine for the Media Wiki software.

Backtracking depends on user-given "black box procedures" that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates. It is, therefore a materialistic rather than a specific algorithm although, unlike many other meta-heuristics, it is guaranteed to find all solutions to a finite problem in a bounded amount of time.

Example: In the Eight Queens problem the challenge is to place eight queens pieces from the game of Chess on a chessboard so that no queen piece is threatening another queen on the board. In the game of chess the queen is a powerful piece and has the ability to attack any other playing piece positioned anywhere else on the same row, column, or diagonals. This makes the challenge quite tricky for humans who often declare after several failed attempts "...there can't be any solution!".

However there are in fact ninety-two valid solutions to the problem although many of those ninety-two are symmetrical mirrors. All of the solutions can be found using a recursive backtracking algorithm. The algorithm works by placing queens on various positions, adding one at a time until either eight queens have been placed on the chess board or less than eight queens are on the board but there are no more safe positions left on the board.

When the latter situation is reached the algorithm backtracks and tries another layout of queens.

4. Attempt any two of the following: (10 × 2 = 20)

(a) Given a graph $G = (V_1, E)$ and let V_1 and V be two distinct vertics. Explain how to modify Dijekstra's shortest path algorithm to determine the number of distinct shortest path from U to V.

Also, comment on whether Dijekstra's shortest path algorithm work correctly if weights are negative.

Ans. Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. Therefore, we assume that $w(u, v) \geq 0$ for each edge (u, v) in E.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u in $\{V, \neg S\}$ with the minimum shortest path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a minpriority queue Q of vertices, keyed by their d values.

Dijkstra (G, w, s)

1. Initialize-single-source (G, s)
2. $S \leftarrow \phi$
3. $Q \leftarrow V[G]$
4. while $Q \neq \phi$
5. do $u \leftarrow$ EXTRACT-MIN(Q)
6. $S \leftarrow S\_\{u\}$
7. for each vertex $v\_Adj[u]$
8. do relax (u, v, w)

Example

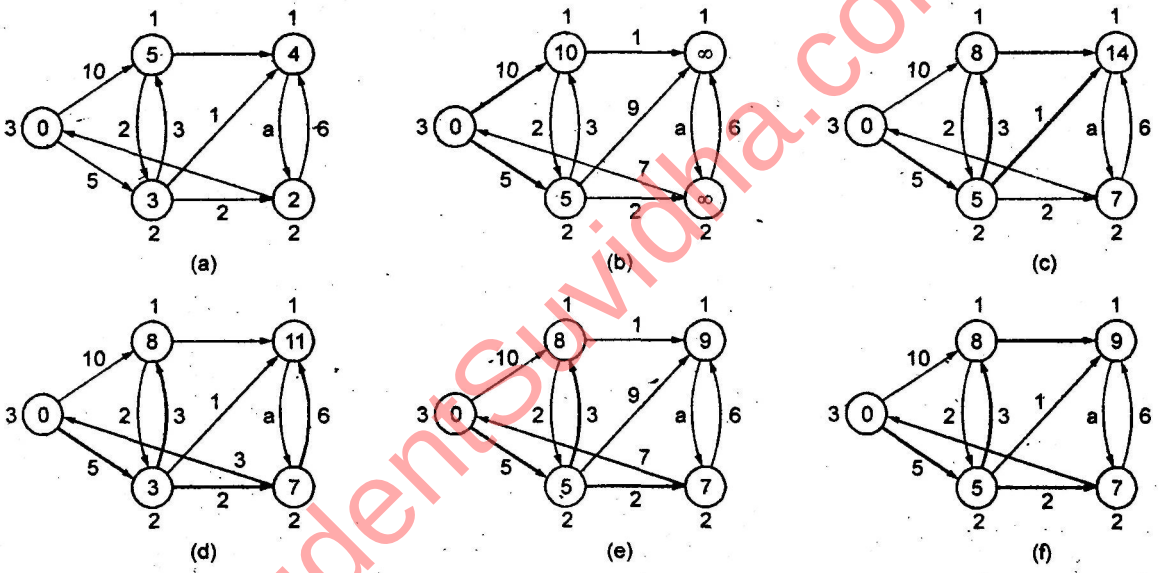


Figure. The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$.

(a) The situation just before the first iteration of the while loop of lines 4 – 8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5.

(b)–(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d and π values shown in part (f) are the final values.

Dijkstra algorithm fails if the graph contains the negative weight edges.

(b) Discuss Travelling salesman Problem and various approaches to solve the problem with complexity analysis of each.

Ans. The traveling-salesman problem

In the traveling-salesman problem, we are given a complete undirected graph $G = (V, E)$ that has a connegative integer cost $c(u, v)$ associated with each edge (u, v) in E, and we must find a Hamiltonian cycle (a tour) of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset A in E.

In many practical situations, it is always cheapest to go directly from a place u to a place w; going by way of any intermediate stop v can't be less expensive. Putting it another way, cutting out an intermediate stop never increases the cost. We formalize this notion by saying that the cost function c satisfies the *triangle inequality* if for all vertices u, v, w in V,

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

The triangle inequality is a natural one, and in many applications it is automatically satisfied. We can solve the traveling salesman problem with the help of following approximation algorithm

APPROX-TSP-TOUR (G, cn>

1. select a vertex r (V [G] to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using MST-PRIM (G, c, r)
3. Let L be the list of vertices visited in a preorder tree walk of T
4. return the Hamiltonian cycle H that visits the vertices in the order L.

The figure below explains this algorithm.

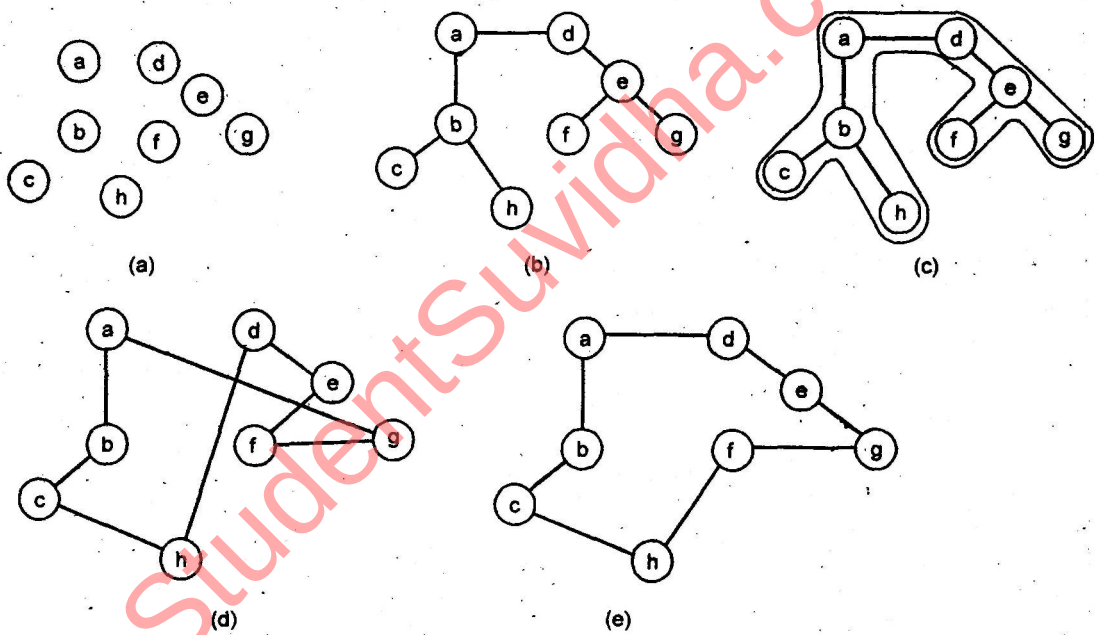


Figure: The operation of APPROX-TSP-TOUR.

(a) The given set of points, which lie on vertices of an integer grid. For example, f is one unit to the right and two units up from h. The ordinary euclidean distance is used as the cost function between two points.

(b) A minimum spanning tree T of these points, as computed by MST-PRIM. Vertex a is the root vertex. The vertices happen to be labeled in such a way that they are added to the main tree by MSTPRIM in alphabetical order.

(c) A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g.

(d) A tour of the vertices obtained by visiting the vertices in the order given by the preorder walk. This is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074.

(e) An optimal tour H^* for the given set of vertices. Its total cost is approximately 14.715.

(c) Explain the Floyd Warshall algorithm with Example. Which design strategy the algorithm uses?

Ans. The Floyd-Warshall algorithm

A different dynamic-programming formulation to solve the all pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the Floyd-Warshall algorithm, runs in $\Theta(V^3)$ time. Here negative-weight edges may be present, but we assume that there are no negative-weight cycles. We shall follow the dynamic-programming process to develop the algorithm.

The structure of a shortest path

In the Floyd-Warshall algorithm, the algorithm considers the "intermediate" vertices of a shortest path, where an *intermediate* vertex of a simple path $p = v_1, v_2,...,v_l$ is any vertex of p other than v_1 or v_l that is, any vertex in the set $\{v_2, v_3,...,v_{l-1}\}$. The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of G are $V = \{1, 2,...n\}$, let us consider a subset $\{1, 2,..., k\}$ of vertices for some. For any pair of vertices i, j in V, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2,..., k\}$, and let p be a minimum-weight path from among them. (Path p is simple.)

The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2,..., k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2,..., k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2,..., k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2,..., k\}$.
- If k is an intermediate vertex of path p, then we break p down vertices in the set $\{1, 2,..., k\}$.

A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we define a recursive formulation of shortest-path estimates. Let be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2,..., k\}$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence. A recursive definition following the above discussion is given by

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

Computing the shortest-path weights bottom up

Based on recurrence, the following bottom-up procedure can be usd to compute the values in order of increasing values of k. Its input is an $n \times n$ matrix W. The procedure returns the matrix $D(n)$ of shortest-path weights.

FLOYD-WARSHALL(W)

1. $n \leftarrow$ rows[W]
2. $d(0) \leftarrow$ W
3. for $k \leftarrow$ 1 to n
4. do for i – 1 to n
5. do for j – 1 to n
6. ******
7. return $D^{[n]}$

5. Write short notes on any four of the following: (5 × 4 = 20)

(a) Approximation of a NP-complete problem.

Ans. We can explain the approximation of a NP Complete Problem by taking the example of vertex cover problem as given below:

Approximation Algorithm

If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it exactly, but even so, there may be hope. There are at least three approaches to getting 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 are solvable in polynomial time. Third, it may still be possible to find *near-optimal* solutions in polynomial time (either in the worst case or on average). In practice, near-optimally is often good enough. An algorithm that returns near-optimal solutions is called an *approximation algorithm*.

Vertex-over problem

The vertex-cover problem

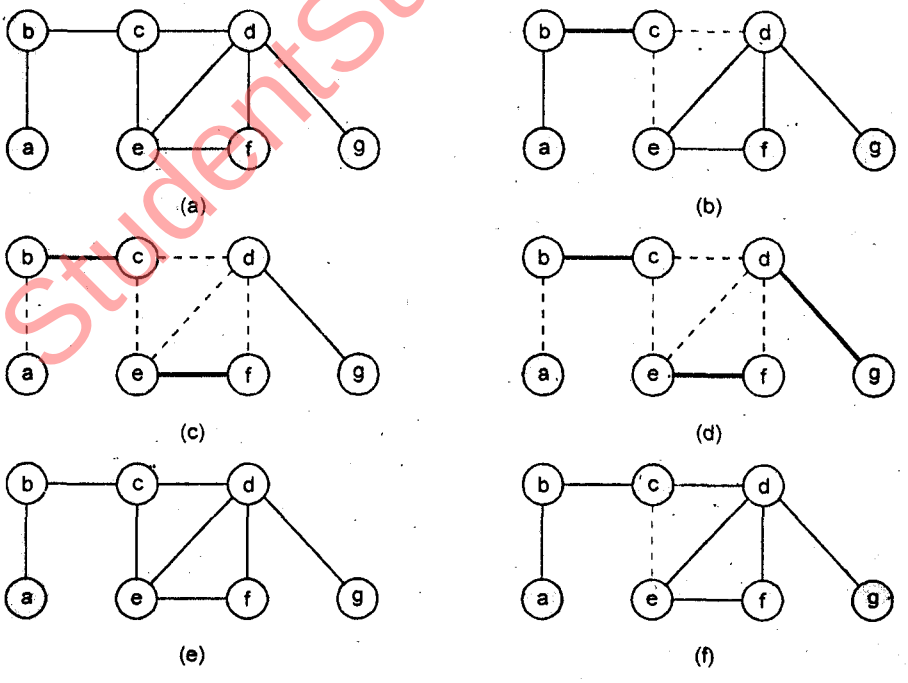
The vertex-cover problem was defined as given a vertex cover of an undirected graph $G = (V, E)$ is a subset V' of set V such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an optimal vertex cover. This problem is the optimization version of an NP-complete decision problem. Even though it may be difficult to find an optimal vertex cover in a graph G, it is not too hard to find a vertex cover that is near-optimal. The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover..

APPROX-VERTEX-COVER(G''>

1. $C \leftarrow \phi$
2. $E' \leftarrow E[G]$
3. while $E' \neq \phi$
4. do let (u, v) be an arbitrary edge of E'
5. $C \leftarrow CU \{u, v\}$
6. remove from E′ every edge incident on either u or v
7. return C

The figure below illustrates the operation of APPROX-VERTEX-COVER.



The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E'. The figure shows the operation of APPROX-VERTEX-COVER.

(a) The input graph G, which has 7 vertices and 8 edges.

(b) The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C.

(c) Edge (e, f) is chosen; vertices e and f are added to C.

(d) Edge (d, g) is chosen; vertices d and g are added to C.

(e) The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g.

(f) The optimal vertex cover for this problem contains only three vertices: b, d and e.

(b) Randomized sorting algorithm.

Ans. The algorithm for the Randomized-quicksort is as given below:

Randomized-quicksort (A, p, r)

1. if $p < r$
2. then $q \leftarrow$ Randomized-partition (A, p, r)
3. Randomized-quicksort $(A, p, q - 1)$
4. Randomized-quicksort $(A, q + 1, r)$

Randomized-partition (A, p, r)

1. $\leftarrow$ Random (p, r)
2. exchanger $A[r] \leftrightarrow A[i]$
3. return partition (A, p, r)

Partition (A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. for $j \leftarrow p$ to $r - 1$
4. do if $A[j] \le x$
5. then $i \leftarrow i + 1$
6. exchange $A[i] \leftrightarrow A[j]$
7. exchange $A[i + 1] \leftrightarrow A[j]$
8. return $i + 1$

Analysis for finding the expected running time

Elements are compared only to the pivot element and, after a particular call of *Partition* finishes, the pivot element used in that call is never again compared to any other elements.

Our analysis uses indicator random variables. We defind

$$X_{ij} = I\{z_i \text{ is compared to } z_j\},$$

where we are considering whether the comparison takes place at any time during the execution of the algorithm, not just during one iteration or one call of *partition*. Since each pair is compared at most once, we can easily characterize the total number of comparisons performed by the algorithm:

$$X = \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

Taking expectations of both sides and then using linearity of expectation, we obtain

$$E[X] = \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

It remains to compute $Pr\{z_i$ is compared to $z_j\}$.

It is useful to think about when two items are *not* compared. Consider an input to quicksort of the numbers 1 through 10 (in any order), and assume that

the first pivot element is 7. Then the first call to *partition* separates the numbers into two sets: {1, 2, 3, 4, 5, 6} and {8, 9, 10}. In doing so, the pivot element 7 is compared to all other elements, but no number from the first set (e.g., 2) is or ever will be compared to any number from the second set (e.g., 9). In general, once a pivot x is chosen with $z_i < x < z_j$, we know that z_i and z_j cannot be compared at any subsequent time. If, on the other hand, z_i is chosen as a pivot before any other item in Z_{ij}, then z_i will be compared to each item in Z_{ij}, except for itself. Similarly, if z_j is chosen as a pivot before any other item in Z_{ij}, then z_j will be compared to each item in Z_{ij}, except for itself. In our example, the values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as a pivot. In contrast, 2 and 9 will never be compared because the first pivot element chosen from $Z_{2,9}$ is 7. Thus, z_i and z_j are compared if and only if the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j. We now compute the probability that this event occurs. Prior to the point at which an element from Z_{ij} has been chosen as a pivot, the whole set Z_{ij} is together in the same partition. Therefore, any element of Z_{ij} is equally likely to be the first one chosen as a pivot. Because the set Z_{ij} has $j - i + 1$ elements, the probability that any given element is the first one chosen as a pivot is $1/(j - i + 1)$. Thus, we have

$$Pr\{z_i \text{ is compared to } z_j\} = Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}$$

$$= Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} + Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

The second line follows because the two events are mutually exclusive. Combining equations we get that

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

We can evaluate this sum using a change of variables (k = j – i) and the bound on the harmonic series in equation

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)$$

Thus we conclude that, using *Randomized-partition*, the expected running time of quicksort is $O(n \lg n)$

(c) Proving the problem of finding maximum clique of a graph to be NPC.

Ans. Proof: To show that CLIQUE $\in$ NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. Checking whether V' is a clique can be accomplished in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

We next prove that 3-CNF-SAT $\leq P$ CLIQUE, which shows that the clique problem is NP-hard. That we should be able to prove this result is somewhat surprising. Since on the surface logical formulas seen to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses. For $r = 1, 2, \ldots, k$ each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r. We shall construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k.

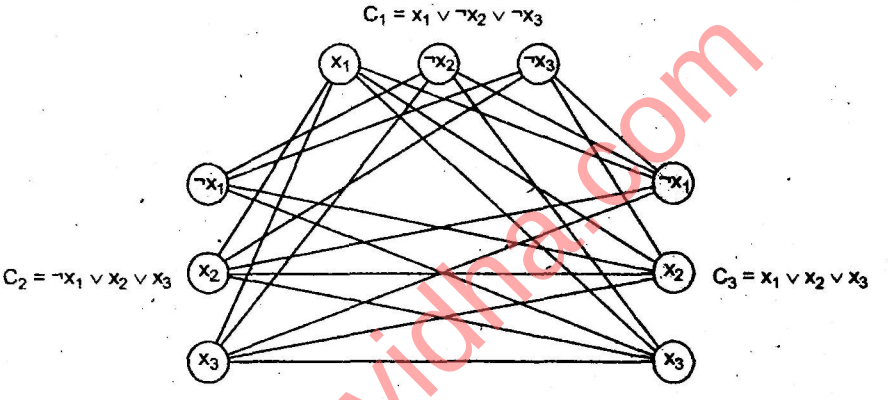
The graph $G = (V \cdot E)$ is constructed as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, we place a triple of vertices v_1^r, v_2^r and v_3^r into V. We put an edge between two vertices v_r^i and v_r^j if both of the following hold.

- v_r^i and v_j^r are in different triples, that is, $r \neq s$, and
- their corresponding literals are *consistent*, that is l_s^r is not the negative of l_s^j.

This graph can easily be computed from ϕ in polynomial time. A an example of this construction, if we have

$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$.

then G is the graph shown in Figure below.



We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s are mapped to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple and so V' contains exactly one vertex per triple. We can assign 1 to each literal l_i^r such that $v_j^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so ϕ is satisfied. (Any variables that do not correspond to a vertex in the clique may be set arbitrarily.)

(d) Problem classes and their implications.

Ans. We have the following classes of problems

P Problems

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(nk)$ for some constant k, where n is the size of the input to the problem.

NP Problems

The class NP consists of those problems that are "verifiable" in polynomial time. What we mean here is that if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem. For example, in the Hamiltonian-cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $<v_1, v_2, v_3, \ldots v_{|V|}>$ of $|V|$ vertices. It is easy to check in polynomial time that (v_i, v_{i+1}) in E for $i = 1, 2, 3, \ldots, |V| - 1$ and that $(v_{|V|}, v_1) - E$ as well. As another example, for 3-CNF satisfiability, a certificate would be an assignment of values to variables. We can easily check in polynomial time that this assignment satisfies the boolean formula.

PC Problems A decision problem C is NP-complete if:

1. C is in NP, and

2. Every problem in NP is reducible to C in polynomial time can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time.

ρ Hard

Problem is NP Hard if it necessarily satisfy the condition given above.

(e) Maximum Flow Problem.

Ans. Let $N = (V, E)$ be a network with s and t being the source and the sink of N respectively.

The capacity of an edge is a mapping $c: E \rightarrow R^+$, denoted by c_{uv} or $c(u, v)$. It represents the maximum amount of flow that can pass through an edge.

A flow is a mapping $f: E \rightarrow R^+$, denoted by f_{uv} or $f(u, v)$, subject to the following two constraints.

1. $f_{uv} \leq c_{uv}$, for each $(u, v) \in E$ (capacity constraint)

2. $\Sigma c_{(u, v)} \in E f_{uv} = \Sigma v_{(v, u)} \in E f_{vu}$, for each $v \in V / \{s, t\}$ (conservation of flows)

The value of flow is defined by $|f| = \Sigma v \in v f_{sv}$, where s is the source of N. It represents the amount of flow passing from the source to the sink.

The maximum flow problem is to maximize $|f|$, that is, to route as much flow as possible from s to t.

We use Ford Fulkerson method to solve the problem of maximum flow:

Pseudocode for Ford-Fulkerson algorithm:

The algorithm takes as argument a graph, a source vertex and the target vertex.

Ford-Fulkerson (G, s, t)

1. for each edge $(u, v) \in E[G]$

2. do $f[u, v] \leftarrow 0$

3. $f[v, u] \leftarrow 0$

4. while there exists a path p from s to t in the residual network G_f

5. do $c_f(p) \leftarrow \min \{c_f(u, v)$: (u, v) is in $p\}$

6. for each edge (u, v) in p

7. do $f[u, v] \leftarrow f[u, v] + c_f(p)$

8. $f[v, u] \leftarrow -f[u, v]$

Under this assumption, a straightforward implementation of Ford-Fulkerson runs in time $O(E|f^*|)$, where f^* is the maximum flow found by the algorithm.

(f) Knuth-Morris-Pratt algorithm for pattern matching.

Ans. The Knuth-Morris-Pratt algorithm

KMP-MATCHER calls the auxiliary procedure **compute-prefix-function** compute π.

KMP-MATCHER (T, P)

1. $n \leftarrow$ length $[T]$

2. $m \leftarrow$ length $[P]$

3. $\pi \leftarrow$ COMPUTE-PREFIX-FUNCTION (P)

4. $q \leftarrow 0$

Number of characters matched.

5. for $i \leftarrow 1$ to n

Scan the text from left to right.

6. do while $q > 0$ and $P[q + 1] \neq T[i]$

7. do $q \leftarrow \pi[q]$

Next character does not match.

8. if $P[q + 1] = T[i]$

9. then $q \leftarrow q + 1$

Next character matches.

10. if $q = m$

Is all of P matched?

11. then print "Pattern occurs with shift" $i - m$

12. $q \leftarrow \pi[q]$ Look for the next match.

COMPUTE-PREFIX-FUNCTION(P)

1. $m \leftarrow$ length $[P]$
2. $\pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. for $q \leftarrow 2$ to m
5. do while $k > 0$ and $P[k + 1] \neq P[q]$
6. do $k \leftarrow \pi[k]$
7. if $P[k + 1] = P[q]$
8. then $k \leftarrow k + 1$
9. $\pi[q] \leftarrow k$
10. return π

Prefix function for the pattern P = ababbabaa

Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$, using the potential method of amortized analysis.

A similar amortized analysis, matching time of KMP-MATCHER is $\Theta(n)$.