Q2 a. Explain the fundamentals of algorithm problem solving.

Ans: Page No. 9 –14 of Textbook

Q2. b. Explain various data structures used in algorithm design. Give their applications.

Ans: Page No. 24-35 of Textbook

Q3 a. What is the difference between time complexity and space complexity?

Ans: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input to the problem. The time complexity of an algorithm is commonly expressed using big O notation, which suppresses multiplicative constants and lower order terms. When expressed this way, the time complexity is said to be described *asymptotically*, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm may take a different amount of time even on inputs of the same size, the most commonly used measure of time complexity, the worst-case time complexity of an algorithm, denoted as $T(n)$, is the maximum amount of time taken on any input of size n. Time complexities are classified by the nature of the function $T(n)$. For instance, an algorithm with $T(n) = O(n)$ is called a linear time algorithm, and an algorithm with $T(n) = O(2^n)$ is said to be an exponential time algorithm.

The space complexity of a program (for a given input) is the number of elementary objects that this program needs to store during its execution.
This number is computed with respect to the size n of the input data.
For an algorithm T and an input x, DSPACE(T, x) denotes the number of cells used during the (deterministic) computation T(x).
We will note DSPACE(T) = O(f (n)) if DSPACE(T, x) = O(f (n)) with n = |x | (length of x).

Note: DSPACE(T) is undefined whenever T(x) does not halt.

Example

// x is an unsorted array

```
int findMin(int[] x) {
int k = 0; int n = x.length;
for (int i = 1; i < n; i++) {
if (x[i] < x[k]) {
k = i;
}
}
return k;
}
```

Q3 b. Write an algorithm for analyzing the efficiency of recursive algorithms.

Ans: Steps in mathematical analysis of recursive algorithms:

- Decide on parameter *n* indicating *input size.*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* case for input of size *n*
  " if the basic operation count also depends on other conditions.
- Set up a recurrence relation and initial condition(s) for *C*(*n*)-the number of times the basic operation will be executed for an input of size *n*
  " Alternatively count recursive calls.
- Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution

Q4 a. What is the best, average and worst case inputs for the algorithm of sequential search.

Ans: Sequential search for '*q*' in array A

```
for i = 1 to n do
   if A [i] ≥ q then
      return index i
   return n + 1
```

This algorithm clearly takes a θ(r), where r is the index returned. This is Ω(n) in the worst case and O(1) in the best case.
If the elements of an array A are distinct and query point *q* is indeed in the array then loop executed (*n* + 1) / 2 average number of times. On average (as well as the worst case), sequential search takes θ(n) time.

Q4. b. Explain "Divide & Conquer Technique".

Ans: Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller sub-problems hoping that the solutions of the sub-problems are easier to find and then composing the partial solutions into the solution of the original problem.

Divide-and-Conquer paradigm consists of following major phases:
Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
Solve the sub-problem recursively (successively and independently), and then
Combine these solutions to sub-problems to create a solution to the original problem.

Q4 c. Explain Brute force string matching algorithm.

Ans: The simplest algorithm for string matching is a brute force algorithm, where we simply try to match the first character of the pattern with the first character of the text, and if we succeed, try to match the second character, and so on; if we hit a failure point, slide the pattern over one character and try again. When we find a match, return its starting location.

code for the brute force method:

```
for (int i = 0; i < n-m; i++) {
int j = 0;
while (j < m && t[i+j] == p[j]) {
j++;
}
if (j == m) return i;
}
System.out.println(‘‘No match found’’);
return -1;
```

The outer loop is executed at most n-m+1 times, and the inner loop m times, for each iteration of the outer loop. Therefore, the running time of this algorithm is in O(nm).

Q5 a. Define BFS. Explain with the help of example how it differs from DFS.

Ans: Breadth-first search is a way to find all the vertices reachable from the a given source vertex, *s*. Like depth first search, BFS traverse a connected component of a given graph and defines a spanning tree. Intuitively, the basic idea of the breath-first search is this: send a wave out from source *s*. The wave hits all vertices 1 edge from *s*. From there, the wave hits all vertices 2 edges from *s*. Etc. We use FIFO queue Q to maintain the wave front: *v* is in Q if and only if wave has hit *v* but has not come out of *v* yet.
Depth-first search is a systematic way to find all the vertices reachable from a source vertex, s. historically, depth-first was first stated formally hundreds of years ago as a method for traversing mazes. Like breadth-first search, DFS traverse a connected component of a given graph and defines a spanning tree. The basic idea of depth-first search is this: It methodically explores every edge. We start over from different vertices as necessary. As soon as we discover a vertex, DFS starts exploring from it (unlike BFS, which puts a vertex on a queue so that it explores from it later).

Q5 b. What is the time efficiency of the DFS based algorithm for topological sorting?

Ans: A cycle in a diagraph or directed graph G is a set of edges, $\{(v_1, v_2), (v_2, v_3),... (v_{r-1}, v_r)\}$ where $v_1 = v_r$. A diagraph is acyclic if it has no cycles. Such a graph is often referred to as a directed acyclic graph, or DAG, for short. DAGs are used in many applications to indicate precedence among events.

DFS traversal stack with the subscript numbers indicating the popping off order.
All edges in the sorted list point from left to right.
Time efficiency is in O(|*V*2|) for the adjacency matrix representation and O(|*V*
|+|*E* |) for the adjacency linked list representation.
Since the reversing requires only Θ (|*V* |) and it can stop before processing the entire digraph if a back edge is encountered.

Q6 a. Define AVL trees. Explain four rotation types for AVL trees with three nodes. Give an illustration.

Ans: These are self-adjusting, height-balanced binary search trees .The height of a binary tree is the maximum path length from the root to a leaf. A single-node binary tree has height 0, and an empty binary tree has height -1.
Types of rotation :
• Single
– left (L)
– right (R)
• Double
– LR (L followed by R)
RL (R followed by L)

There are four cases which need to be considered, of which two are symmetric to the other two. Let P be the root of the unbalanced subtree. Let R be the right child of P. Let L be the left child of P.

Right-Right case and Right-Left case: If the balance factor of P is −2, then the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must be checked. If the balance factor of R is < 0, a left rotation is needed with P as the root. If the balance factor of R is -1, a double left rotation (with respect to P and then R) is needed . If the balance factor of R is +1, two different rotations are needed. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

Left-Left case and Left-Right case: If the balance factor of P is +2, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must be checked. If the balance factor of L is > 0, a right rotation is needed with P as the root. If the balance factor of L is +1, a double right rotation (with respect to P and then L)

is needed. If the balance factor of R is -1, two different rotations are needed. The first rotation is a left rotation with L as the root. The second is a right rotation with P as the root.

Q6 b. Explain the heap sort in detail. Give its complexity.

The running time of heap sort is O(N*lgN) i.e. it achieves the lower bound for computational based sorting.

HEAP
The heap data structure is an array object which can be easily visualized as a complete binary tree.There is a one to one correspondence between elements of the array and nodes of the tree.The tree is completely filled on all levels except possibly the lowest, which is filled from the left upto a point.All nodes of heap also satisfy the relation that the key value at each node is at least as large as the value at its children.

Step I: The user inputs the size of the heap (within a specified limit).The program generates a corresponding binary tree with nodes having randomly generated key Values.

Step II: Build Heap Operation: Let n be the number of nodes in the tree and i be the key of a tree.For this,the program uses operation Heapify.when Heapify is called both the left and right subtree of the i are Heaps. The function of Heapify is to let i settle down to a position (by swapping itself with the larger of its children,whenever the heap property is not satisfied)till the heap property is satisfied in the tree which was rooted at (i).This operationcalls

Step III: Remove maximum element:The program removes the largest element of the heap(the root) by swapping it with the last element.

Step IV: The program executes Heapify(new root) so that the resulting tree satisfies the heap property.

Step V: Goto step III till heap is empty

Q7 a. With the help of example, Differentiate between Prim's and Kruskal's algorithm.

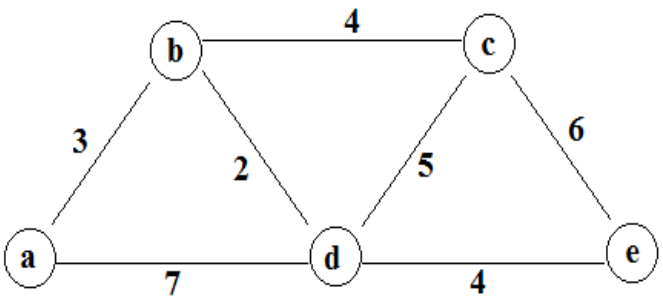
Ans: Prims Algorithm is used when the given graph is dense , whereas Kruskals is used when the given is sparse, we consider this because of their time complexities even though both of them perform the same function of finding minimum spanning tree. It starts from an arbitrary vertex (root) and at each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have been reached. This strategy is greedy in the sense that at each step the partial spanning tree is augmented with an edge that is the smallest among all possible adjacent edges.

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

Like Prim's algorithm, *Kruskal's algorithm* also constructs the minimum spanning tree of a graph by adding edges to the spanning tree one-by-one. At all points during its execution the set of edges selected by Prim's algorithm forms exactly one tree. On the other hand, the set of edges selected by Kruskal's algorithm forms a forest of trees.
Kruskal's algorithm is conceptually quite simple. The edges are selected and added to the spanning tree in increasing order of their weights. An edge is added to the tree only if it does not create a cycle.

The beauty of Kruskal's algorithm is the way that potential cycles are detected. Consider an undirected graph $G = (\mathcal{V}, \mathcal{E})$. We can view the set of vertices, $\mathcal{V}$, as a *universal set* and the set of edges, $\mathcal{E}$, as the definition of an *equivalence relation* over the universe $\mathcal{V}$. . In general, an equivalence relation partitions a universal set into a set of equivalence classes. If the graph is connected, there is only one equivalence class--all the elements of the universal set are *equivalent*. Therefore, a *spanning tree* is a minimal set of equivalences that result in a single equivalence class.

Q7 b. Using Dijkra's algorithm, Find the shortest path from a to e?



Ans: Page No 308 of Textbook

Q8 a. Explain the Hashing technique in detail. What is the difference between open hashing and closed hashing? Explain B- trees.

Ans: Hashing is key-to-address translation. This means the value of a key is transformed into a disk address by means of an algorithm, usually a relative block and anchor point within the block. It's closely related to statistical probability as to how well the algorithms work.

The hash function calculates an index within the array from the data key. Array Length is the size of the array. For assembly language or other low-level programs, a trivial hash function can often create an index with just one or two inline machine instructions

A hash table is where data storage for a key-value pair is done by generating an index using a hash function.

Open Hashing (Separate chaining) is simpler to implement, and more efficient for large records or sparse tables.

Closed Hashing (Open Addressing) is more complex but can be more efficient, especially for small data records.

An alternative to open addressing as a method of collision resolution is separate chaining hashing. This again uses an array as the primary hash table, except that the array is an array of lists of entries, each list initially being empty. When an entry is inserted, it is inserted at the end of the list at the index corresponding to the hash code of the key in question. Searching the hash table now involves walking down the list at the given index though, with good design, it should be a relatively short list. This method has some attractive features and offers an alternative way of implementing the LookupTable interface, via a Chaining Hash Table class.
closed hashing, is a method of collision resolution in hash tables. With this method a hash collision is resolved by probing, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. Well known probe sequences include:
Linear probing

in which the interval between probes is fixed—often at 1.

Quadratic probing

in which the interval between probes increases linearly (hence, the indices are described by a quadratic function).

Double hashing

in which the interval between probes is fixed for each record but is computed by another hash function.

Q8 b. What are NP, NP complete and NP hard problems?

Ans: NP is the set of decision problems solvable in polynomial time on a nondeterministic Turing machine. Or, equivalently, YES answers are checkable in polynomial time on a deterministic Turing machine

The complexity class NP-complete is the set of problems that are the hardest problems in NP, in the sense that they are the ones most likely not to be in P. If you can find a way to solve an NP-complete problem quickly, then you can use that algorithm to solve all NP problems quickly.

A set or property of computational search problems (generally: all construction problems of corresponding NP-complete decision problems are NP-hard) A problem is NP-hard if solving it in polynomial time would make it possible to solve all problems in class NP in polynomial time. Some NP-hard problems are also in NP (these are called "NP-complete"), some are not. If you could reduce an NP problem to an NP-hard problem and then solve it in polynomial time, you could solve all NP problems. Also, there are decision problems in NP-hard but are not NP-complete, such as the infamous halting problem

9 a. Solve the following instance of the knapsack problem by the branch –and-bound algorithm. (W=10)

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

Ans:

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Remarks:

- The final output is $V[4, 10] = 90$.
- The method described does not tell which subset gives the optimal solution. (It is $\{2, 4\}$ in this example).

TEXTBOOK

Introduction to The Design & Analysis of Algorithms, Anany Levitin, Second Edition, Pearson Education, 2007