Q.1   a. Solve the recurrence relation $T(n) = 27 T(n/3) + \Theta(n^3 \log n)$
Answer:
$n^{\log_3 27} = n^3$ vs. $n^3 \log n$
Therefore $T(n) = O(n^3 \log_2 n)$

b. Given the following code fragment, what is its Big-O running time?
   
   ```
   i = n;
   while i > 0
     k = k + 2;
     i = i / 2;
   ```
Answer:
$O(\log n)$

c. Show the ordering of vertices produced by topological sort in the following graph. What is time complexity of topological sort?

```
   v1 ---- v2
  /      |
 v3 ---- v4 ---- v5
```
Answer:
Topological sort - Order of vertices: V1, V2, V4, V3, V5 or V1, V2, V4, V5, V3
Time complexity: $\Theta(V + E)$

d. Given a sorted array and a value x. Suggest $O(n)$ algorithm to find two values in the array whose sum is equal to x.
Answer:
We keep two indexes one at start and 2nd one at end, and apply following algo. Let the array be sorted in descending order.
```
  2nd_index--;
  1st_index++;
else
  print 1st_index,2nd_index;
do this until 2nd_index > 1st_index
```

e. Suppose that the root of the Red-Black tree is red. If we make it black, does the tree remain a Red Black tree?
Answer:
If we color the root of a relaxed red-black tree black but make no other changes, the resulting tree is a red-black tree. Not even any black-heights change.
f. What are the conditions for a problem to be solved using Dynamic Programming.

Answer:
Optimal substructure and Overlapping sub problems

g. Explain intractable problem with an example.

Answer:
Some problems are intractable as they grow large; we are unable to solve them in reasonable time. e.g. subset-sum problem, TSP etc.

Q.2

a. Give an efficient algorithm that determines whether or not a given directed graph \( G = (V, E) \) contains a cycle. Discuss its time complexity.

Answer:

Function \( \text{iscycle}(G) \)

\( \text{NV}=0; \)  // \( \text{NV} \) is number of vertices visited

select a vertex that has in degree zero

\( \text{NV} = \text{NV} + 1 \)

delete the vertex and all the edges emanating from it from the graph

if \( \text{NV} \neq V[G] \) then return “ cycle is there”

else return “ no cycle is there”

Time complexity:  In case of Adjacency Matrix \( O(V^2) \).

In case of Adjacency List \( O(V + E) \).

b. Suppose we wish to search a linked list of length \( n \), where each element contains a key \( k \) along with a hash value \( h(k) \). Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

Answer:

Searching a list of length \( n \) where each element contains a long key \( k \) and a small hash value \( h(k) \) can be optimized in the following way: Comparing the keys should be done first comparing the hash values and if successful then comparing the keys.

Q.3

a. What is the difference between the binary-search tree property and the heap property? Can the heap property be used to print out the keys of an \( n \)-node tree in sorted order in \( O(n) \) time? Explain how or why not.

Answer:

In a heap, a node’s key is \( \geq \) both of its children’s keys. In a binary search tree, a node’s key is \( \geq \) its left child’s key, but \( \leq \) its right child’s key. The heap property, unlike the binary-search-tree property, doesn’t help print the nodes in sorted order because it doesn’t tell which subtree of a node contains the element to print before that node. In a heap, the largest element smaller than the node could be in either subtree.
Note that if the heap property could be used to print the keys in sorted order in $O(n)$ time, we would have an $O(n)$-time algorithm for sorting, because building the heap takes only $O(n)$ time. But we know that a comparison sort must take $\Omega(n \lg n)$ time.

b. Consider a B-tree with degree $m$, i.e. the number of children $c$, of any internal node (except the root) is such that $m-1 \leq c \leq 2m-1$. Derive the maximum and minimum number of records in the leaf nodes for such a B-tree with height $h$ ($h \geq 1$). (Assume that the root of a tree is at height $0$).

Answer:
The root which is at height $0$ can have minimum two children. Each of these children can have minimum of $m$ children each of which can have a minimum of $m$ children. Thus the minimum number of records in leaf nodes with height $h$ is $2 \times m^{h-1}$.
Similarly the maximum number of records in leaf nodes with height $h$ is $2(2m - 1)^{h-1}$.

Q.5

a. Consider the problem of "Making Change". Coins available are:
- dollars (100 cents)
- quarters (25 cents)
- dimes (10 cents)
- nickels (5 cents)
- pennies (1 cent)

Design an algorithm using greedy approach to make a change of a given amount using the smallest possible number of coins.

Answer:

Informal Algorithm
- Start with nothing.
- at every stage without passing the given amount.
  - add the largest to the coins already chosen.

Formal Algorithm
Make change for $n$ units using the least possible number of coins.

MAKE-CHANGE (n)

C ← {100, 25, 10, 5, 1} // constant.
Sol ← \{\}; // set that will hold the solution set.
Sum ← 0 sum of item in solution set
WHILE sum not = n
  x = largest item in set C such that sum + x \leq n
  IF no such item THEN
    RETURN "No Solution"
  S ← S \{value of x\}
  sum ← sum + x
RETURN S
b. Write a program to merge two arrays in sorted order, so that if an integer is in both the arrays it gets added into the final array only once.

Answer:

Algorithm Union(arr1[], arr2[]):
For union of two arrays, follow the following merge procedure.
1) Use two index variables i and j, initial values i = 0, j = 0
2) If arr1[i] is smaller than arr2[j] then print arr1[i] and increment i.
3) If arr1[i] is greater than arr2[j] then print arr2[j] and increment j.
4) If both are same then print any of them and increment both i and j.
5) Print remaining elements of the larger array.

Q.6 a. How can the output of the Floyd-Warshall algorithm be used to detect the presence of a negative-weight cycle?

Answer:
Here are two ways to detect negative-weight cycles:
(a) Check the main-diagonal entries of the result matrix for a negative value.

There is a negative weight cycle if and only if $d_{ii}^{(n)} < 0$ for some vertex $i$:

- $d_{ii}^{(n)}$ is a path weight from $i$ to itself; so if it is negative, there is a path from $i$ to itself (i.e., a cycle), with negative weight.
- If there is a negative-weight cycle, consider the one with the fewest vertices.
- If it has just one vertex, then some $w_{ii} < 0$, so $d_{ii}$ starts out negative, and since $d$ values are never increased, it is also negative when the algorithm terminates.
- If it has at least two vertices, let $k$ be the highest-numbered vertex in the cycle, and let $i$ be some other vertex in the cycle. $d_{ik}^{(k-1)}$ and $d_{ki}^{(k-1)}$ have correct shortest-path weights, because they are not based on negative weight cycles. (Neither $d_{ik}^{(k-1)}$ nor $d_{ki}^{(k-1)}$ can include $k$ as an intermediate vertex, and $i$ and $k$ are on the negative-weight cycle with the fewest vertices.) Since $i \rightarrow k \rightarrow i$ is a negative-weight cycle, the sum of those two weights is negative, so $d_{ii}^{(k)}$ will be set to a negative value.

Since $d$ values are never increased, it is also negative when the algorithm terminates.

In fact, it suffices to check whether $d_{ii}^{(n-1)} < 0$ for some vertex $i$. Here’s why. A negative-weight cycle containing vertex $i$ either contains vertex $n$ or it does not. If it does not, then clearly $d_{ii}^{(n-1)} < 0$. If the negative-weight cycle contains vertex $n$, then consider $d_{nn}^{(n-1)}$. This value must be negative, since the cycle, starting and ending at vertex $n$, does not include vertex $n$ as an intermediate vertex.

(b) Alternatively, one could just run the normal FLOYD-WARSHALL algorithm one extra iteration to see if any of the $d$ values change. If there are negative cycles, then some shortest-path cost will be cheaper. If there are no such cycles, then no $d$ values will change because the algorithm gives the correct shortest paths.
Text Book