CSO 221 Algorithms

Enroll
Page No:
Date: / /

Name: Shruti T. Avhad Roll No.:- 20074030 Branch: CSE(IDD)

1.) a)

Differences between the working of the greedy method and dynamic programming.

	Greedy Method	Dynamic Programming
1.	In this method, we choose the best option at that step and believe that it will lead to the global optimum solution.	Here, we make the decision at each step considering current problem and solution to previously solved sub problem to calculate the optimal soln.
2.	There is no such guarantee of getting optimal solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then chooses the best.
3.	It is more efficient in terms of memory and is generally faster.	It is not as efficient as greedy as it needs to maintain a DP table table and is also relatively slower.
4.	E.g 1.) Fractional Knapsack Problem By using greedy method, the local optimum strategy is to choose the item that has	E.g 2.) 0/1 Knapsack Problem This problem has the properties of optimal substructure and overlapping subproblems and can be solved

Enroll
Page No:
Date: / /

the maximum value vs. wt. ratio. This strategy leads to global optimal soln as well because taking fractions of an item is allowed.

by DP. Recomputation of same sub-problems is avoided by constructing a DP Table in a bottom up manner.

2.) Prim's Algorithm

In prim's algorithm, to find the min. spanning tree of a connected undirected graph, the greedy approach is used by selecting the min. weight edge from the set of edges.

2.) Floyd Warshall Algorithm

It is based on DP to find the shortest distances between every ~~point~~ pair of vertices in directed and weighted graph.

1.) (b)

Let the array P[] denote the order of matrices

$\therefore$ P[] = {10, 30, 5, 60}

i^{th} matrix is of dimension p[i-1] * p[i] i=1,2,3

$A_{10\times30}$, $B_{30\times5}$, $C_{5\times60}$

i) Using Memoization

We construct a dp[4][4] table and initialise all the values with -1 { dp[n][n] where n=4 }

dp[i][j] denotes the max. no. of multiplications for multiplying matrices from i to j.

Enroll

Page No:

Date: / /

Initially, dp[4][4] = $\begin{bmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix}$

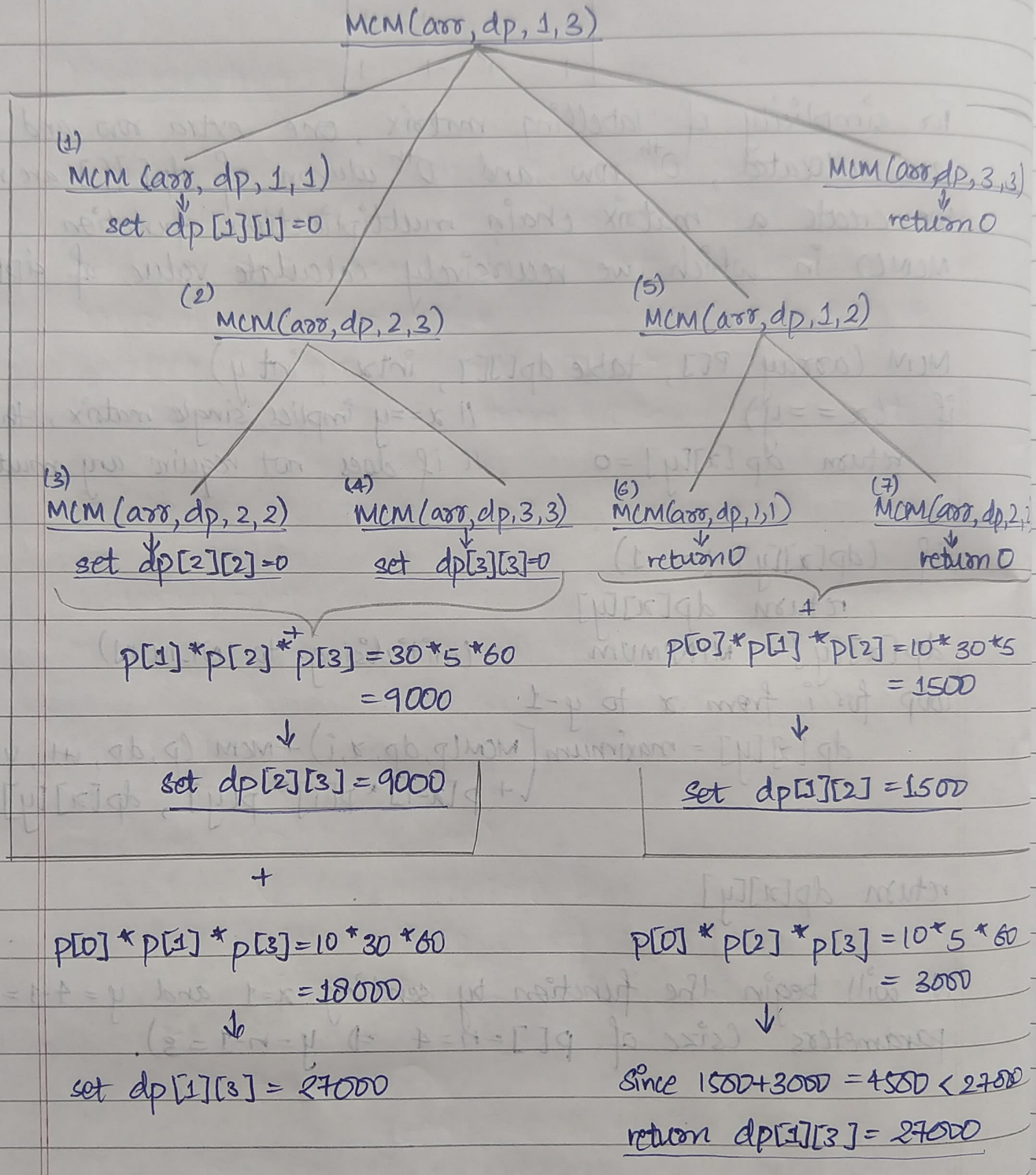
For simplicity of labelling matrix, one extra row and column are allocated, 0^{th} row and 0^{th} column of dp[][] are not used. We create a matrix chain multiplication function MCM() in which we recursively calculate value of dp[i][j]

```
MCM (array P[], table dp[][], int x, int y)
if (x==y)                          // x==y implies single matrix, thus
   return dp[x][y]=0               // it does not require any multiplication

if (dp[x][y] != -1)
      return dp[x][y]
dp[x][y] = MINIMUM                 (initialising to INT_MIN)
loop for i from x to y-1
      dp[x][y] = maximum[MCM(p,dp,x,i)+MCM(p,dp,i+1,y)
                         + p[x-1]*p[i]*p[y], dp[x][y]]

return dp[x][y]
```

We will begin the function by sending x=1 and y = 4-1 = 3 as parameters (size of p[] = n = 4 ⇒ y = n-1 = 3).

Enroll
Page No:
Date: / /

Function call tree:

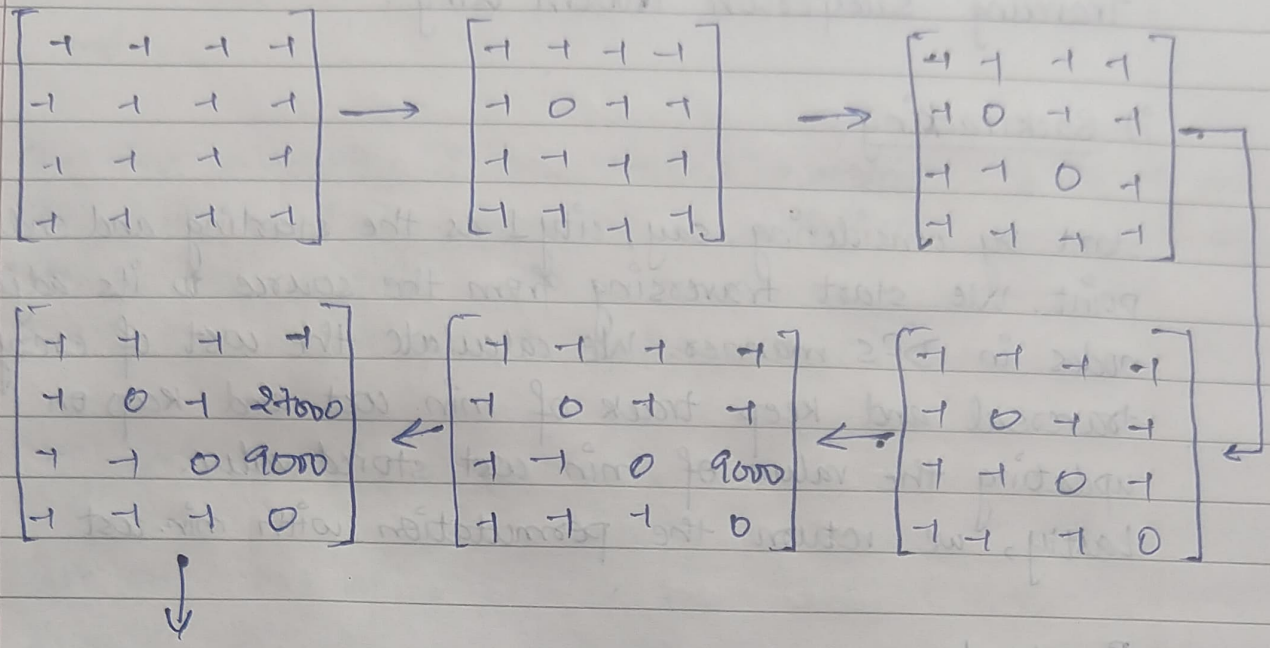


Enroll

Page No:

Date: / /

STATUS OF DP TABLE:



$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & 0 & 1500 & 27000 \\ \infty & \infty & 0 & 9000 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

$$T(N) = \sum_{l=1}^{N} (N-l+1) * l = \sum_{l=1}^{N} (Nl - l^2 + l)$$

$$= \frac{N^2(N+1)}{2} - \frac{N(N+1)(2N+1)}{6} + \frac{N(N+1)}{2}$$

Ignoring powers of N less than 3, $T(N) = \frac{N^3}{2} - \frac{N^3}{3} = \frac{N^3}{6} = \underline{O(N^3)}$

The min. no. of multiplications: $dp[1][3] = 27000$

The Time complexity is $O(n^3)$, since inside the for loop we made recursive calls 2 times.

Enroll
Page No:
Date: / /

Without Memoization

i) Using tabulation

dp[4][4] table created
for i=1 to n-1 do:
    dp[i][i] = 0
(Since the multiplication are not needed for a single matrix.)
For chain length 'L' from 2 to n-1 do:
    for i=1 to n-L do:
        j = i+L-1
        dp[i][j] = INT_MIN (Initialising).
        for k=i to j-1 do:
            x = dp[i][k] + dp[k+1][j] + p[i] p[k] p[j]
            if x > dp[i][j]:
                dp[i][j] = x.
Return dp[1][n-1].

Here, x is the max. no. of multiplications needed to multiply matrices from i to k and from k+1 to j added with the multiplications required to multiply these two products together.
dp[1][3] stores the required max. no. of multiplications.
- The initial and final state of dp table are the same as memoization method. Again the maximum no. of multiplications are 27000
The running time of 'without memoization' method is also $O(n^3)$, because there are 3 nested for loops used.

Enroll
Page No:
Date: / /

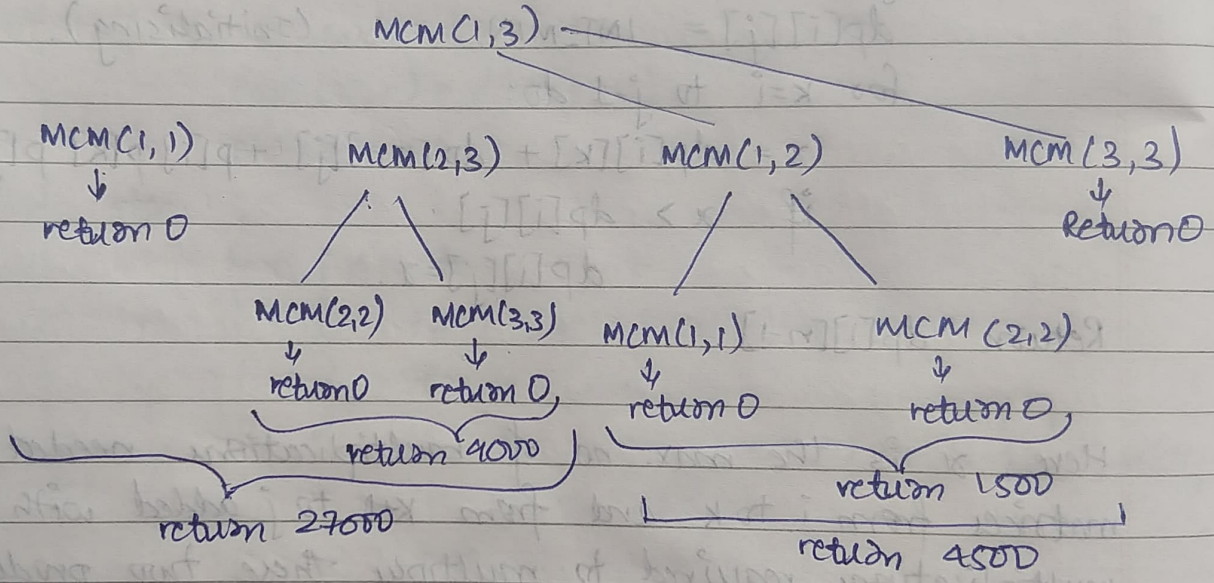
ii) Without using Dynamic Programming

Here, we simply do,

if x == y
    return 0
else
    return max[MCM(x,i) + MCM(i+1,y) + p[x-1] * p[i] * p[y], MAX]
    for all i from x to y-1.

DP table is not used.

Recursion Tree



Time complexity without memoisation is exponential, since we solve the same subproblems again in recursive calls. Therefore, time complexity with memoization in $O(n^3)$

Time complexity without memoization → Tabular → $O(n^3)$
                                    → without dp → exponential.

Although, in the given testcase / example, both the methods called the same number of function calls (recursive). Thus, for the given example both of them performed equally well in terms of time complexity.

Enroll
Page No:
Date: / /

2. (a)

Traveling salesperson Problem Using:-

- i) Backtracking -

Start by considering, say, city 1 as the starting and ending point. We start traversing from the source to its adjacent nodes in DFS manner. We calculate the cost of every traversal and keep track of min. cost and keep on updating the value of min. cost stored value.
Lastly, we return the permutation with min. cost.

Pseudocode:

s: starting point, N: number of cities / vertices, c: count of cities, Cn: adjacency matrix (Cn[N][N]) denoting cost of edges.

```
Visited[N] = 0;
Ans = INT_MAX;
Procedure TSP (i, c, cost)
    if c=N and Cn[i][s] ≠ 0 then
        Ans = min [Ans, cost + Cn[i][s]];
        Return Ans;
    else
        for j ∈ N do
            if visited[j] = 0 and Cn[i][j] ≠ 0 then
                Visited[j] = 1;
                Ans = min (Ans, TSP(j, c+1, cost + Cn[i][j]));
                Visited[j] = 0;
            end
        end
```

```
        Return Ans;
    end
end
```

The time complexity of this backtracking algorithm is O(N!). Since, for first node/city there are N possibilities, for second there (N-1) and so on.

$\therefore T(N) = N * (N-1) * \dots 1 = O(N!)$

ii) Dynamic Programming

Take an array of cities $\{1, 2, 3, \dots, n\}$
Initially, all cities are unvisited, and the visit starts from the city s. We assume that the initial cost of travelling is equal to 0. Next, the TSP distance is calculated based on a recursive function. We take a subset N of the required cities that needs to be visited and distance among the cities dist()

- If the no. of cities in the subset is 2, then the recursive function returns their distance as a BASE CASE.
- Else, if the number of cities > 2, then we calculate the dist. from the current city to the nearest city, and the min. dist. among the remaining cities is calculated recursively.

Enroll
Page No:
Date: / /

Pseudocode:

```
s: starting point, N: subset of input cities
dist(): distance among the cities.
Visited[N] = 0;
cost = 0;
Procedure TSP(N, s)
    Visited[s] = 1;
    if |N| - 2 and k ≠ s then
        cost(N, k) = dist(s, k);
        Return cost;
    else
        for j ∈ N do
            for i ∈ N and visited[i] = 0 do
                if j ≠ 1 and j ≠ s then
                    cost(N, j) = min[cost(N, j), TSP(N - {i}, j) + dist(j, i)]
                    Visited[j] = 1;
                end
            end
        end
    end
    Return cost;
end
```

Since, the no. of possible subsets can be at most $N \times 2^N$ and each subproblem can be solved in $O(N)$ time.
Therefore, the time complexity of this algorithm is $O(N^2 \times 2^N)$.

2.) b)

Let's say we have been given the three sides of the triangle - a, b, c

If (a equals b) AND (b equals c)
    then the triangle is equilateral
Else if (a equals b AND a not equals c) OR (b equals c AND b not equals a)
    then the triangle is Isosceles
Else
    the triangle is scalene.

As a precheck, we will also check whether the given sides even form a triangle or not.

The pseudocode is a follows:

```
start
    if a+b ≤ c OR b+c ≤ a OR c+a ≤ b then
        print " Triangle not possible"
    else
        if a=b AND b=c then
            print "Equilateral Triangle"
        else if (a=b AND a≠c) OR (b=c AND b≠a) OR (a=c AND a≠b)
            print "Isosceles Triangle"
        else
            print "Scalene Triangle"
        end
    end
end.
```

Enroll
Page No:
Date: / /

2.) c)

Given two integers x and n, we need to find x^n.

We can write n as a sum of powers of 2

$$n = 2^{K_1} + 2^{K_2} + 2^{K_3} + \cdots + 2^{K_m}$$

$$\therefore x^n = x^{2^{K_1}} * x^{2^{K_2}} * x^{2^{K_3}} * \cdots * x^{2^{K_m}}$$

⇒ To find x^n, we need to look at each bit in binary representation of n and multiply all $x^{2^{K_i}}$ together, where K_i is i^{th} bit.

```
Procedure pow(x, n)
    res = 1;
    while n>0 do
        if n & 1 then        // checking if the current LSB is set(1) or unset(0)
            res = res * x;
        end
        x = x * x;
        n >>= 1;    // Right shifting the bits by 1 position (same as n=n/2)
    end
    return res;
end
```

For Eg., $x=2$ and $n=10$

$\therefore$ to find 2^{10}

$(10)_{10} = (1010)_2$ (Binary representation).

Enroll

Page No:

Date: / /

Steps

1. $res = 1$, $x = 2$, $n = 10 = (1010)_2$
2. $n \& 1 = 0$
   $\therefore res = 1$, $x = 2^2$, $n = (101)_2$
3. $n \& 1 = 1$
   $\therefore res = 2^2$, $x = 2^4$, $n = (10)_2$
4. $n \& 1 = 0$
   $\therefore res = 2^2$, $x = 2^8$, $n = (1)_2$
5. $n \& 1 = 1$
   $\therefore res = 2^{10}$, $x = 2^{16}$, $n = (0)_2$
6. $n = 0$
   $\therefore$ return res.

Page No:
Date: / /

3.) (a)

SAT (Boolean ~~satisfying~~ satisfiability Problem) is the problem of determining if there exists an interpretation that satisfies a given boolean formula (i.e. assigning the variables of the formula with values in such a way that the formula evaluates to TRUE).

To prove that SAT problem is NP-complete, we need to show:

i) SAT problem is in NP class
ii) SAT problem is NP-Hard.

→ i) If any problem is in NP class, then if given a certificate (a solution) to an instance of that problem, we can check or verify that it is correct in polynomial time.

Suppose, we have been given a boolean formula 'f' and a certificate i.e solution (satisfying set of inputs) to that problem, then we can verify the given solution by checking if the given assignment of variables satisfies the boolean formula in polynomial time.

∴ ⇒ SAT problem is in NP class.

ii) We know that the CIRCUIT SAT problem is an NP-complete problem. A boolean circuit c can be reduced / transformed into a boolean formula by:-

- adding a new variable for every input wire (y_i)
- adding a new variable for every output wire (z)
- an equation is formed for each gate.

- These set of equations are conjucted together along with = at the end.

This transformation can be done in linear (polynomial) time.
Now, if there is a set of input variable values satisfying the circuit then it can derive an assignment for the formula f that satisfies the formula.
Also, if there is a satisfying assignment for the formula f, this can satisfy the boolean circuit c by removal of newly added variables.
Therefore, this is a valid reduction.
Since circuit SAT is an NP-complete Problem
⇒ from (i) and (ii) part, we can say that
SAT problem is NP-complete
Hence Proved.

3.) b)
The topological sort algorithm works as follows:
→ Create an empty list L.
→ Run Dfs on G. Whenever a vertex v turns black (i.e it is ~~blocked~~ popped from the stack) append it to L.
→ Output the reverse order of L as the topological sort.
Time complexity: $O(|V| + |E|)$.

Proof of correctness of Topological sort Algorithm
Let's take any edge (u, v) of the given DAG (Directed Acyclic Graph.) we will show that u turn black after v, which completes the proof.

consider the moment when u enters the stack. we argue that currently v cannot be in the stack. Suppose that v was already in the stack. As there must be a path changing up all the vertices in the stack bottom up, we know that there is a path from v to u. Then adding the (u, v) forms a cycle, which contradicts the fact that G is a DAG.

∴ v cannot be already in the stack before u has entered

So,

- If v is black at this moment, then obviously u will turn black after v.
- If v is white, then by white path theorem of DAG, we know that u will become a proper descendant of u in the DFS-forest.

∴ u will turn black after v.

This proves the correctness of Topological sort Algorithm.

3.) (c)

The asymptotic running time of Heapsort Algorithm on an array of length n that is already sorted in ascending order is $O(n \log n)$.

Reason: Considering min-heap, then the build heap procedure won't do any swaps since the array is already sorted in ascending order. However, it would consume $O(n)$ time.

Enroll
Page No:
Date: / /

Now, the heapsort procedure will swap the first and last element and then heapify the root, which taken $O(\log n)$ time.
since, we do the same thing for all array elements.
$\therefore T(n) = O(n \log n)$.

3.) (d)
The asymptotic running time of Heapsort algorithm on an array of length n that is sorted in descending order is $O(n \log n)$

Reason: Just as in part (c), the build heap will take $O(n)$ time and the heapsort will take $O(\log n)$ for each element. Therefore $T(n) = O(n * \log n)$.