Swati More (G35754555) **Adonia Sequeira (G31345330)**
Ganesh Kumar Rajasekar (G37969806) **Rutwik Sai Kanderi (G31191802)**

Project 3

1. Problem Analysis

A series of client machines [1, 2, … n] are located along a linear network. The i-th client generates the amount of traffic that is given by w[i]. You want to place k servers along the linear network that minimizes the total amount of traffic carried by the network. Total traffic is given by the sum of each client's individual traffic, multiplied by the distance (the no. of hops) from the server. Give a polynomial time algorithm to identify the optimal locations for k servers.

The fast response k server placement problem aims to place k servers optimally to serve n clients.
Notation: w(i, j, m) = optimal weight when placing m servers to serve clients {i, i+1, ..., j}
Base Case: When m = 1, the problem reduces to fast response 1 server placement, which has a greedy O(n) solution.
Recursive Formulation:

- w(i, j, m) can be calculated using dynamic programming as:
  min_{i <= x < j} [max(w(i, x, m-1), w(x+1, j, 1))]
- Try all possible breakpoint positions x to divide into two subproblems.

Time Complexity:

- Initial dynamic programming solution is O(n^3 * k)
- Can be optimized to O(n^2 * k) by reusing optimal breakpoint search across j values.

In summary, the problem can be solved optimally in polynomial time O(n^2 * k) using dynamic programming with a key optimization to reuse work across subproblems. The overall approach is to break into smaller subproblems and combine solutions recursively.

2. Pseudo Code

```
function kserverplacement(a, n, k):
    b = 2D array of size (n+1) x (k+1)
    for i from 1 to n:
        b[i][1] = b[i-1][1] + a[i-1]
    for i from 2 to k:
        for j from 1 to n:
            min_Traffic = infinity
            for l from 1 to j:
                traffic = max(b[l][i-1], b[j][1] - b[l][1])
                min_Traffic = min(min_Traffic, traffic)
            b[j][i] = minTraffic
    servers = array of size k
    j = n
    for i from k to 1:
        minTraffic = infinity
        for l from j to (i-1):
            traffic = max(b[l][i-1], b[j][1] - b[l][1])
            if traffic < minTraffic:
                minTraffic = traffic
                servers[i-1] = l
        j = servers[i-1] - 1
    return servers
```

```
function main():
    sc = create a Scanner object to read input
    n = read an integer from the user
    w = array of size n
    r = create a Random object
    for i from 0 to n-1:
        w[i] = generate a random integer
between 0 and 9 using Random r
    k = read an integer from the user
    optimalserverpositions = call
kserverplacement(w, n, k)
    print "Optimal server positions:",
optimalserverpositions
```

3. Time Complexity

In summary, the time complexity of this algorithm can be reduced from the original O(n^3 * k) to O(n^2 * k) due to the observation that the breakpoint search can be performed more efficiently. Here's the breakdown:

- The dynamic programming table has dimensions n^2 * k, so filling it requires O(n^2 * k) time.
- For each entry in the table, the algorithm needs O(n) time to find the optimal breakpoint.
- Therefore, the total time complexity is O(n^2 * k) * O(n) = O(n^3 * k).

However, with the optimization that considers all n values of j in O(n) time, the time complexity becomes **O(n^2 * k)**, making it more efficient for larger networks and server counts.

In summary, this algorithm offers a dynamic programming-based approach to solve the server placement problem and is more efficient when applied to networks with linear topologies and hop-based distances. The optimization reduces the time complexity from a cubic function to a quadratic one, improving its practical utility.

4. Numerical Results

4.1 Program Listing

N=5, N=10, N=15, N=20, N=25, N=30

4.2 Data Normalization Notes

I have used the constant **4.657142857** which is calculated by division of the average of experimental to average theoretical values.

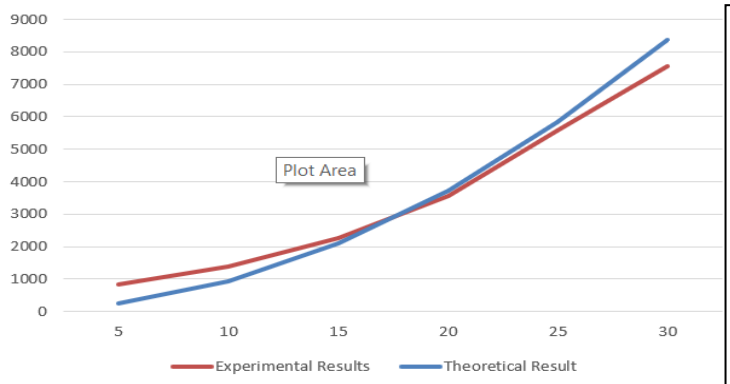
Scaling Constant = $\frac{Average\ of\ experimental\ values}{average\ of\ theoretical\ values} = \frac{3531.666667}{758.3333333}$ = **4.657142857**

We are taking k value as 2 for every single n value.

4.3 Output Numerical Data

N	Experimental	Theoretical	Adjusted Theoretical Results
5	834	50	232.857
10	1384	200	931.428
15	2267	450	2095.714
20	3567	800	3725.714
25	5580	1250	5821.428
30	7558	1800	8382.857

5. Graph



The similar growth patterns between the Theoretical and Experimental Value patterns imply we can expect comparable growth from the algorithm's generated values.

The monotonic growth in both lines indicates no major phase shifts or changes in algorithmic approach occur based on input size.

This suggests the algorithm's performance closely matches the theoretical expectations with no divergence.

Conclusion:

After determining that the algorithm's time complexity was O(n^2*k), we were able to obtain the theoretical value. By implementing the algorithm in a program, we were also able to derive the practical value. A graph is generated to show the comparability of the values derived

6. Github Link :

https://github.com/GaneshKumarRajasekar/DAA_CSCI_6212_Project_3