Name: Nam Huynh

ID: nsh1507

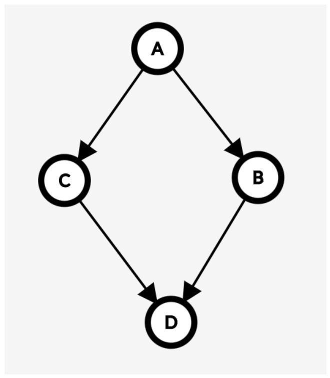
1)

a)

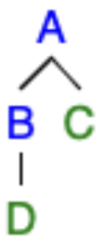
In the BFS algorithm, once a vertex is dequeued, it's not enqueued again. Therefore, the algorithm won't revisit the same vertex multiple times. Even if the color of a vertex is not explicitly set to black, it won't be enqueued again, ensuring that each vertex is explored only once; since the purpose of the color Black and Gray is to distinguish whether a node has been dequeued or not. Therefore, using a single bit to store each vertex color wouldn't affect the result of the BFS algorithm.

b)

In a breadth-first search (BFS) algorithm, the value d[u] assigned to a vertex u represents the shortest distance from the starting vertex (source) to vertex u. This distance is calculated based on the number of edges in the shortest path from the source to u. According to the theorem, "*Given a directed graph G = (V,E), and a vertex s ∈ V, after BFS(G, s) executes, d[v] = δ(s,v) for all v ∈ V. Further, for any v ∈ V, if v ≠ s and v is reachable from s then a shortest path from s to v is a shortest path from s to π[v] followed by v.*" The BFS algorithm ensures that this distance is correctly computed regardless of the order in which the vertices in each adjacency list are given.

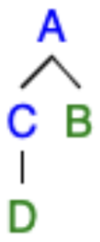


Suppose that the vertex B precedes vertex C in A’s adjacency list:



A: 0 B: 1 C: 1 D: 2

Now suppose that the vertex C precedes vertex B in A’s adjacency list:



A: 0 C: 1 B: 1 D: 2

The two BFS trees show that the order of the adjacency list matters because if the vertex C precedes B, then the neighbors of C will get added to the BFS tree first instead of the neighbors of B. However, despite the difference in ordering of the BFS tree, the path computed from A to D still remains the same. It shows that a BFS tree computed can depend on the ordering within adjacency lists but the discovery time of the vertices can be independent.

2)

a)

Table containing the optimal number of scalar operations:

i/j	***0***	***1***	***2***	***3***	***4***	***5***
0		150	330	405	1655	2010
1		0	360	330	2430	1950
2			0	180	930	1770
3				0	3000	1860
4					0	1500
5						0

Choices table

i/j	***0***	***1***	***2***	***3***	***4***	***5***
0		0	1	1	3	1
1			1	1	1	1
2				2	3	3
3					3	3
4						4
5						

The optimal number of scalar operation is 2010 and the optimal parenthesization of this matrix-chain is: $(\quad (A_0 A_1) \quad (\quad (A_2 A_3)(A_4 A_5) \quad) \quad)$

b)

Theorem: For any $n \in N$, if $n \geq 1$ then a full parenthesization of an n-element expression has $n - 1$ pairs of parentheses.

Proof: By the strong form of induction

Observe that when $n = 1$ we have:

The expression has 0 pairs of parenthesis.

Since the full parenthesization of a 1 expression matrix chain has 0 pairs of parentheses, the base case holds.

Assume that for any $k \in N$, if $0 < k < n$ then a full parenthesization of an k-element expression has $k - 1$ pairs of parentheses.

Suppose there exists a full parenthesization of a d-element expression where d = k+1, then the expression must have d – 1 pairs of parentheses. Since d = k + 1, then the expression must have k + (1 – 1) pairs of parentheses.

c)

Running the recursive algorithm is the more efficient way to determine the optimal number of scalar multiplications in a matrix-chain multiplication problem. The recursive algorithm would split the matrix chain and find the optimal way to parenthesize the left half and right half, while the enumeration method would require an exhaustive search (exponential time) through all possible parenthesized expressions, resulting in a more inefficient and costly computational cost.

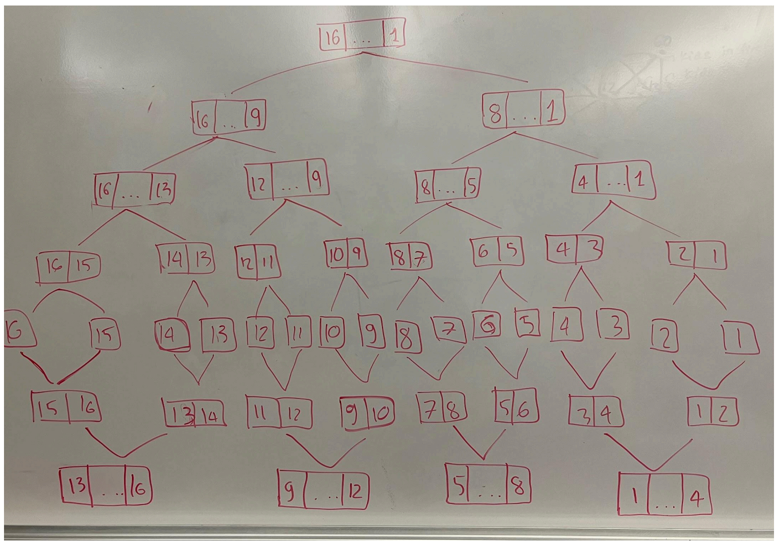
d)

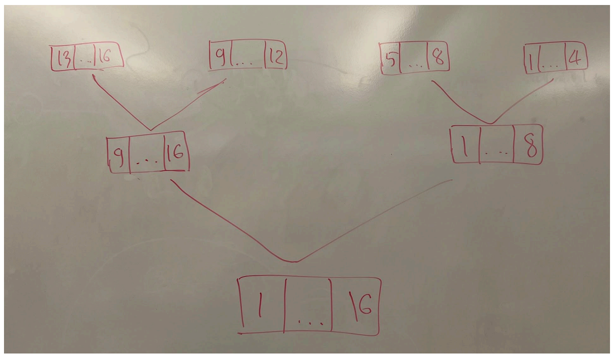
This problem does exhibit optimal substructure. Assume make a cut in the sequence: $A_0 \ldots A_k | A_{k+1} \ldots A_{n-1}$. Recursively we'll find the maximum number of scalar multiplications for both $A_0 \ldots A_k$ and $A_{k+1} \ldots A_{n-1}$. The maximum number of scalar multiplications for the entire sequence is the sum of those plus the number of scalar multiplications required to multiply those two results together.

Pseudocode to compute the largest number of scalar multiplication:

```
def MAXMUL(S) :
    $n \leftarrow |S|$
    $m \leftarrow array(n, n)$; $c \leftarrow array(n, n)$
    for i ←0 to n−1:
        $j \leftarrow i$
        $m[i, j] \leftarrow 0$
    for $l \leftarrow 2$ $to$ $n$:
        for $i \leftarrow 0$ to $n - l$:
            $j \leftarrow l + i - 1$
            $champ \leftarrow -\infty$
            for $k \leftarrow i$ to $j - 1$:
                $p \leftarrow rows(S[i])$
                $q \leftarrow cols(S[k])$
                $r \leftarrow cols(S[j])$
                $t \leftarrow m[i, k] + m[k + 1, j] + p \cdot q \cdot r$
                if $t > champ$:
                    $champ \leftarrow t$; $c[i, j] \leftarrow k$
            $m[i, j] \leftarrow champ$
    **return** $m[0, n - 1]$, c
```

3)





The merge sort algorithm performs at most a single call to any pair of indices of the array that is being sorted. The subproblems do not overlap and therefore memoization will not improve the running time.

4)

a) Project Problem

b)

a)

```
knapSack( [], capacity )      =  0
knapSack( d::ds, 0)           =  0
knapSack( d::ds, capacity)    =  knapSack( ds, capacity )        if d.weight > capacity
                              =  max(d.value + knapSack(ds, capacity – d.weight)),
                                 knapSack(ds, capacity))         otherwise
```

b)

```
knapSack(items, capacity)  =        0                            if |items| = 0 or capacity = 0
knapSack( items, capacity) =  knapSack( items[1:], capacity )    if d.weight > capacity
                           =  max(items[0].value + knapSack(items[1:], capacity – items[0].weight)),
                              knapSack(items[1:], capacity))     otherwise
```

c)

```
knapSack(items, capacity)          =   knapSackSlice( items, capacity, |items|)


knapSackSlice(items, capacity, l) =   0                          if l = 0 or capacity = 0
knapSackSlice(items, capacity, l) =  knapSackSlice( items, capacity, l – 1)
                                                                 if items[0].weight > capacity
                                   =  max(items[0].value + knapSackSlice(items, capacity –
                                      items[0].weight, l – 1)),  knapSackSlice(items, capacity, l – 1))
                                                                 otherwise
```

d) Project Problem

e) Project Problem

c)

a)

$e(S, M, i, j) = M - |S[i]|$ ***if i = j***

$= e(S, M - |S[i]| - 1, i + 1, j)$ **otherwise**

c)

$bl(S, M, i, j) = \infty$ ***if*** $e(S, M, i, j) < 0$

$= e(S, M, i, j)$ **otherwise**

e)

$mb(S, M) = 0$ ***if*** $\boldsymbol{bl(S, M, 0, |S|-1) \neq \infty}$

$= \min\{ \max\{bl(S, M, 0, k), mb(S[k+1:], M)\} \mid$

$0 \leq k \leq |S| - 1 \}$ ***otherwise***

f)

$mb'(S, M, i) = 0$ ***if*** $\boldsymbol{i \geq |S|}$ **or** $\boldsymbol{bl(S, M, i, |S|-1) \neq \infty}$

$= \min\{ \max\{bl(S, M, i, k), mb'(S, M, k+1)\} \mid$

$i \leq k \leq |S| - 1)$ ***otherwise***