DEPARTMENT OF CYBER SECURITY

# DAA Home Work 5

| | |
|---|---|
| **SUBMITTED BY** | 211042 Noman Masood Khan A |
| **SEMESTER** | Fifth |
| **SUBJECT** | DAA |
| **SUBMITTED TO** | Sir Ammar Masood |

## Part 11.1-1

To find the maximum element in the direct address table of length m we have to traverse over the complete table and compare the elements to find the maximum.

- One possible way to find the maximum is:

```
FindMax (T, m)
{
        max=T[0]
        for i = 1 to m
        {
        if T[i] != NIL && max < T[i]
        max = T[i]
        }
        return max
}
Here, T is table.
```

**In the Worst case, searching of the compete table would be needed till the loop reaches Nill. Thus, the Worst case Time complexity is O(m) time. Where m is the length of table.**

## Part 11.1-2

Bit Vector is an array which stores bits. Bit at each index will be either 1 or 0. Where 1 Bit value corresponds to value of array S and 0 represents NILL place.

For example, storing the set S{5,6,2,10,1} in the bit vector of length 10:

The Bit Vector Array:

- 01100110001

i.) Searching Operation :

```
BITMAP-SEARCH(V, k)

   if V[k] != 0
           return k
   else return NIL
```

For example, k= 7. It will check whether we have 0 or 1 at the 7 indices in the V. If we have 1, then the element is present in the array else 7 is not present in the bitmap array V.

ii.) Insertion Operation:

```
BITMAP-INSERT(V, x)
   V[x] = 1
```

iii.) Delete Operation:

```
BITMAP-DELETE(V, x)
 V[x] = 0
```

- Each of these Operation will run O(1) time.

## Part 11.2-3

The professor modifications will affect the chaining scheme in the following way:
Hera α is the load factor on the hash table.

- Successful Search Operation:
  Theta of (1+ α) which is same as the original running time. Still in the sorted array, an element is likely to be any of the element in the hash table.
- Unsuccessful Search Operation:
  Half of the original running time, which finally becomes theta of (1+ α). In the sorted list, we can divide the size to half and then on the basis of the value we are searching for, we can decide whether we have to search on the right side of the list or on the left side of an array. This divides the running time by half, but asymptotically it still writes theta of (1+ α).
- Insertion Operation:
  Takes theta of (1+ α) time. It takes more time than the original running time of theta of (1). This is because we have sorted the list and insertion have to traverse to the tail of the list.
- Deletion Operation:
  Running time is theta of (1+ α) as original.

## Part 11.3-1

We have a linked list of elements which have the both the keys and their respective hash values in them. Now, when we are searching for any element, first we take its hash and compare it with the hash value of element in the list. If hash matches, then we compare the strings else we move on. Also, when the hashes don't meet, we can identify that the string values are different before comparing the strings. This makes the runtime of the algorithm faster.

## Part 11.3-2

If the string of r characters, treated as a radix-128 number, takes more storage than the available constant number of words, you can use the division method without requiring additional space by performing the calculations incrementally as you read the characters of the string.

Here is the approach to hash the string incrementally using the constant space:

```
sum = 0
  for i = 1 to r
      sum = (sum * 128 + s[i]) % m
```

For each character in for loop iteration, it multiplies the current sum by the 128 and adds the numeric value of the character (s[i]). Then, it takes the modulo operation by m to ensure the resulting hash value falls within the range of 0 to m-1.

## Part 14.2-1

The corresponding matrix are:

A1 – 5 x 10

A2 – 10 x3

A3 – 3 x12

A4 - 12 x5

A5 – 5 x50

A6 – 50 x 6

From the algorithm, we have,

m[1,2] = m[1,1] + m[2,2] + p0 * p1 * p2

m[1,2] = 0 + 150

m[1,2] = 150

m[3,4] = m[3,3] + m[4,4] + p2 * p3 * p4

m[3,4]=0+180

m[3,4] = 180

m[4,5] = m[4,4] + m[5,5] + p3* p4 * p5

m[4,5] = 0+3000

m[4,5] = 3000

m[5,6] = m[5,5] + m[6,6] + p4 * p5 * p6

m[5,6]=0+1500

m[5,6] = 1500

m[1,3] = min of {m[1,1]+m[2,3]+p0* p1* p3=750 }
{m[1,2]+m[3.3] + p0*p2* p3=330)

m[2,4] = min of {m[2,2]+m[3,4] + p1*p2*p4 = 330) }
{ m[2, 3] + m[4, 4] +p1 * p3 * p4=960 }

m[3,5] = min of {m[3,3] + m[4,5] + p2* p3 * p5 = 4800}
{m[3,4] + m[5,5] + p2 * p4* p5 = 930 }

m[4,6] = min of {m[4,4] + m[5,6] + p3 * p4* p6 = 1860 }
{m[4,5] + m[6,6] + p3* p5 * p6 = 6600 }

m[1,4] = min of {m[1,1] + m[2,4] + p0 * p1 * p4 = 580}
{m[1,2] + m[3,4] + p0 * p2 * p4 = 405 }
{m[1,3] + m[4,4] + p0* p3 *p4 = 630 }

m[2.5] = min of {m[2,2] + m[3,5] + p1* p2 * p5 = 2430 }
{m[2,3] + m[4,5] + p1*p3 * p5 = 9360 }
{m[2,4] + m[5,5] + p1* p4* p5 = 2830 }

m[3,6] = min of {m[3,3]+m[4,6] + p2* p3* p6 = 2076}
{m[3,4] + m[5,6] + p2* p4* p6=1770}
{m[3,5]+m[6,6] + p2*p5*p6= 1830)

m[1,5] = {m[1,1]+m[2,5]+ p0 * p1* p5=4930}
{m[1,2]+m[3,5]+ p0 *p2*p51=830)
{m[1,3]+m[1,4]+ p0 *p3*p5=6330}
{m[1,4]+m[1,5] + p0 *p4*p5 = 1655)

m[2,6] = {m[2,2]+m[3,6] + p1*p2*p6 = 1950)
{m[2,3]+m[4,6] + p1 * p3* p6 = 2940)
(m[2,4]+m[5,6]+p1 * p4*p6 = 2130)
m[2,5]+m[6,6] + p1 * p5*p6= 5430

m[1,6] = [m[1,1]+m[2,6)+p0*p1*p6 = 2250 }
{m[1,2] + m 3,6j+p0* p2*p6 = 2010)
{m[1,3]+m[4,6)+p0* p3* p6 = 2550 }
{m[1,4]+m[5,6]+p0 *p4*p6 2055 }
{m[1,5]+m[6,6]+p0 *p5 *p6=3155}

| M | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 6 | 2010 | 1950 | 1770 | 1840 | 1500 | 0 |
| 5 | 1655 | 2430 | 930 | 3000 | 0 | |
| 4 | 405 | 330 | 180 | 0 | | |
| 3 | 330 | 360 | | | | |
| 2 | 150 | | | | | |
| 1 | 0 | | | | | |

Using this we make the S table:

| S | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 2 | 2 | 4 | 4 | 5 |
| 5 | 4 | 2 | 4 | 4 | |
| 4 | 2 | 2 | 3 | | |
| 3 | 2 | 2 | | | |
| 2 | 1 | | | | |

From the above calculation, the optimal parameterization is:

- ((A1 * A2) * (A3 * A4) * (A5 * A6))

## Part 14.2-2

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
{
        if (i >= j)
        {
                return A[i];
        }
        else
        {
           return
                MATRIX-MULTIPLY(MATRIX-CHAIN-MULTIPLY(A, s, i, s[i][j]), MATRIX-
                CHAIN-MULTIPLY(A, s, s[i][j] + 1, j);
   }
}
```

## Part 14.2-3

- **Substitution Method :**

$$P(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

$$\geq \sum_{k=1}^{n-1} c.c.2^k.c.2^{n-k}$$

$$= c^2 \sum_{k=1}^{n-1} 2^k 2^{n-k}$$

$$= c^2 \sum_{k=1}^{n-1} 2^n$$

$$= c^2 \sum_{k=1}^{n-1} 2^n$$

$$= c^2 (n-1) 2^n$$

$$\geq c.\, 2^n$$

## Part 14.2-5

The information we get about the loops is:

- The third loop (*k*) references *m* twice.
- The body of the third loop gets executed j - i = L-1 times.
- The body of the second loop gets executed *n* - *L* +1 times.
- The body of the first loop gets execute *n*−1 times, with *l* moving from 2 to *n*.

So, this is the following sum:

$$\begin{aligned}\sum_{l=2}^{n} 2(l-1)(n-l+1) &= \sum_{l=1}^{n-1} 2l(n-l) \\ &= 2n\sum_{l=1}^{n-1} l - 2\sum_{l=1}^{n-1} l^2 \\ &= 2n\frac{(n-1)n}{2} - 2\frac{(n-1)((n-1)+1)(2(n-1)+1)}{6} \\ &= n^3 - n^2 - \frac{n(n-1)(2n-1)}{3} \\ &= \frac{1}{3}\left(3n^3 - 3n^2 - 2n^3 + n^2 + 2n^2 - n\right) \\ &= \frac{1}{2}\left(n^3 - n\right) \\ &= \frac{n^3 - n}{3}\end{aligned}$$

## Part 14.3-2

Memoization is a technique where the expensive function calls are cached so it can re-used when the same input is called again. Memoization is effective is dynamic programming where the subproblems are occurring again. But it is not suitable for the merge sort algorithm.

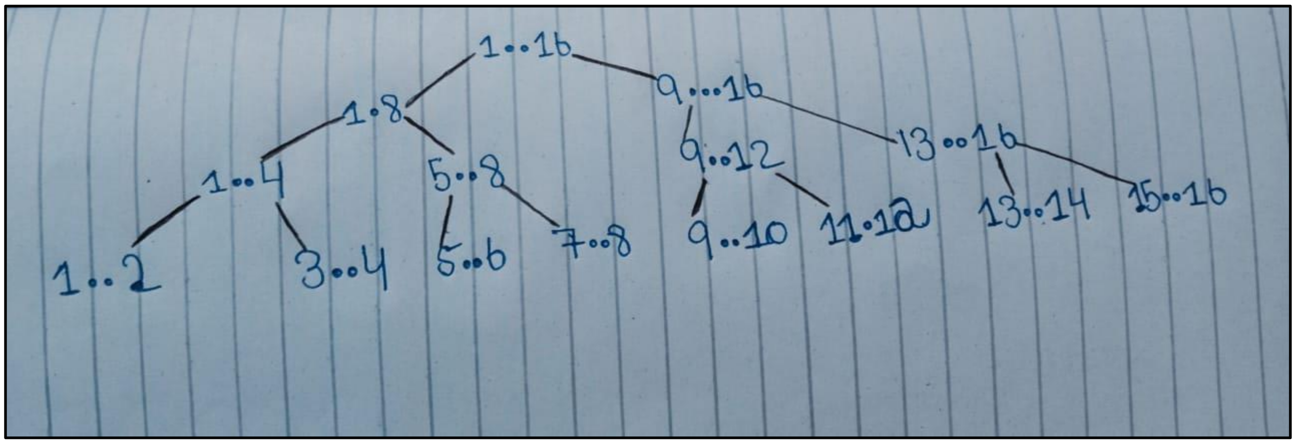

*Figure 1: Merge Sort Recursion Tree*

Merge sort is dividing the tree into more sub-problems and then into more sub-problems, and at each level merge sort algorithm is handling different level of problem. there are no redundant subproblems within the same run of the algorithm, there is no opportunity for the memorization to store and reuse the results.

## Part 14.3-3

- Yes, this problem does exhibit optimal substructure property.

Say the highest level parameterization splits the matrix chain into two sub chains. The parameterization within each sub-chain must be such that they maximize the number of scalar multiplications involved for each sub chain.

- Example

Let us suppose that the first sub-chain could be parenthesized in another way that increases the multiplications for that sub-chain. Then we could obtain a higher number of

total multiplications for the entire chain by parenthesizing the first chain in this other way. **This is so because the parameterization within one sub-chain does not affect the other chain, neither does it affect the cost of the eventual multiplication of the two sub-chains.** The final multiplication cost itself depends on the matrix dimensions which are constant for a particular split point regardless of how each sub-chain is internally parenthesized.

So, it must be the case that when choosing from the various split points possible, the best split point can be obtained by considering, for every split point, the cost of multiplying two sub-chains and the cost of each sub-chain optimally parenthesized internally.

## Part 14.4-3

```
LCS-LENGTH(X, Y)
    m ← length[X]
    n ← length[Y]
    for i ← 1 to m do
        for j ← 1 to n do
            c[i,j] ← -1
        end for
    end for
    return LOOKUP-LENGTH(X,Y,m,n)
```

```
LOOKUP-LENGTH(X,Y,i,j)
    if c[i,j] > -1 then
        return c[i,j]
    end if
    if i = 0 or j = 0 then
        c[i,j] ← 0
    else
        if X[i] = Y[j] then
            c[i,j] ← LOOKUP-LENGTH(X,Y,i-1,j-1)+1
        else
            c[i,j] ← max(LOOKUP-LENGTH(X,Y,i,j-1),LOOKUP-LENGTH(X,Y,i-1,j))
        end if
    end if
    return c[i,j]
```