

Lecture 11 - Dynamic Programming algorithms

Introduction

The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. It was invented by mathematician named Richard Bellman in 1950s. The DP is closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. The DP differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these sub-problems.

The steps of Dynamic Programming technique are:

- **Dividing the problem into sub-problems:** The main problem is divided into smaller sub-problems. The solution of the main problem is expressed in terms of the solution for the smaller sub-problems.
- **Storing the sub solutions in a table:** The solution for each sub-problem is stored in a table so that it can be referred many times whenever required.
- **Bottom-up computation:**

The DP technique starts with the smallest problem instance and develops the solution to sub instances of longer size and finally obtains the solution of the original problem instance.

The strategy can be used when the process of obtaining a solution of a problem can be viewed as a sequence of decisions. The problems of this type can be solved by taking an optimal sequence of decisions. An optimal sequence of decisions is found by taking one decision at a time and never making an erroneous decision. In Dynamic Programming, an optimal sequence of decisions is arrived at by using the principle of optimality. *The principle of optimality states that whatever be the initial state and decision, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.*

A fundamental difference between the greedy strategy and dynamic programming is that in the greedy strategy only one decision sequence is generated, whereas in the dynamic programming, a number of them may be generated. Dynamic programming technique guarantees the optimal solution for a problem whereas greedy method never gives such guarantee.

Lecture 12 - Matrix Chain Multiplication

Let, we have three matrices A_1 , A_2 and A_3 , with order (10×100) , (100×5) and (5×50) respectively. Then the three matrices can be multiplied in two ways.

- (i) First, multiplying A_2 and A_3 , then multiplying A_1 with the resultant matrix i.e. $A_1(A_2 A_3)$.
- (ii) First, multiplying A_1 and A_2 , and then multiplying the resultant matrix with A_3 i.e. $(A_1 A_2) A_3$.

The number of scalar multiplications required in case 1 is $100 * 5 * 50 + 10 * 100 * 50 = 25000 + 50,000 = 75,000$ and the number of scalar multiplications required in case 2 is $10 * 100 * 5 + 10 * 5 * 50 = 5000 + 2500 = 7500$

To find the best possible way to calculate the product, we could simply parenthesize the expression in every possible fashion and count each time how many scalar multiplications are required. Thus the matrix chain multiplication problem can be stated as *"find the optimal parenthesisation of a chain of matrices to be multiplied such that the number of scalar multiplications is minimized"*.

Lecture 13 - Elements of Dynamic Programming

Dynamic Programming Approach for Matrix Chain Multiplication

Let us consider a chain of n matrices A_1, A_2, \dots, A_n , where the matrix A_i has dimensions $P[i-1] \times P[i]$. Let the parenthesisation at k results two sub chains A_1, \dots, A_k and A_{k+1}, \dots, A_n . These two sub chains must each be optimal for A_1, \dots, A_n to be optimal. The cost of matrix chain (A_1, \dots, A_n) is calculated as $\text{cost}(A_1, \dots, A_k) + \text{cost}(A_{k+1}, \dots, A_n) + \text{cost of multiplying two resultant matrices together i.e.}$

$$\text{cost}(A_1, \dots, A_n) = \text{cost}(A_1, \dots, A_k) + \text{cost}(A_{k+1}, \dots, A_n) + \text{cost of multiplying two resultant matrices together.}$$

Here, the cost represents the number of scalar multiplications. The sub chain (A_1, \dots, A_k) has a dimension $P[0] \times P[k]$ and the sub chain (A_{k+1}, \dots, A_n) has a dimension $P[k] \times P[n]$. The number of scalar multiplications required to multiply two resultant matrices is $P[0] \times P[k] \times P[n]$.

Let $m[i, j]$ be the minimum number of scalar multiplications required to multiply the matrix chain (A_i, \dots, A_j) . Then

- (i) $m[i, j] = 0$ if $i = j$
- (ii) $m[i, j] = \text{minimum number of scalar multiplications required to multiply } (A_i, \dots, A_k) + \text{minimum number of scalar multiplications required to multiply } (A_{k+1}, \dots, A_n) + \text{cost of multiplying two resultant matrices i.e.}$

$$m[i, j] = m[i, k] + m[k, j] + P[i-1] \times P[k] \times P[j]$$

However, we don't know the value of k , for which $m[i, j]$ is minimum. Therefore, we have to try all $j - i$ possibilities.

$$\begin{cases} 0 & \text{if } i = j \end{cases}$$

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k, j] + P[i-1] \times P[k] \times P[j] \}$$

Otherwise

Therefore, the minimum number of scalar multiplications required to multiply n matrices A_1, A_2, \dots, A_n is

$$m[1, n] = \min_{1 \leq k < n} \{ m[1, k] + m[k, n] + P[0] \times P[k] \times P[n] \}$$

The dynamic programming approach for matrix chain multiplication is presented in Algorithm

Algorithm MATRIX-CHAIN-MULTIPLICATION (P)

// P is an array of length $n+1$ i.e. from $P[0]$ to $P[n]$. It is assumed that the matrix A_i has the dimension $P[i-1] \times P[i]$.

```
{  
    for( $i=1; i \leq n; i++$ )  
         $m[i,i] = 0$ ;  
  
    for( $l = 2; l \leq n; l++$ ){  
  
        for( $i = 1; i \leq n-(l-1); i++$ ){  
             $j = i + (l-1)$ ;  
  
             $m[i, j] = \infty$ ;  
  
            for( $k = i; k \leq j-1; k++$ )  
  
                 $q = m[i, k] + m[k+1, j] + P[i-1] P[k] P[j]$  ;  
  
                if ( $q < m[i, j]$ ){  
  
                     $m[i, j] = q$ ;  
  
                     $s[i, j] = k$ ;  
  
                }  
            }  
        }  
    }  
}  
  
Return  $m$  and  $s$ .  
}
```

Algorithm Matrix Chain multiplication algorithm.

Now let us discuss the procedure and pseudo code of the matrix chain multiplication. Suppose, we are given the number of matrices in the chain is n i.e. A_1, A_2, \dots, A_n and the dimension of matrix A_i is $P[i-1] \times P[i]$. The input to the matrix-chain-order algorithm is a sequence $P[n+1] = \{P[0], P[1], \dots, P[n]\}$. The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \dots, n$ in lines 2-3. Then, the algorithm computes $m[i, j]$ for $j-i = 1$ in the first step to the calculation of $m[i, j]$ for $j-i = n-1$ in the last step. In lines 3 – 11, the value of $m[i, j]$ is calculated for $j-i = 1$ to $j-i = n-1$ recursively. At each step of the calculation of $m[i, j]$, a calculation on $m[i, k]$ and $m[k+1, j]$ for $i \leq k < j$, are required, which are already calculated in the previous steps.

To find the optimal placement of parenthesis for matrix chain multiplication A_i, A_{i+1}, \dots, A_j , we should test the value of $i \leq k < j$ for which $m[i, j]$ is minimum. Then the matrix chain can be divided from $(A_1 \dots A_k)$

and $(A_{k+1} \dots A_j)$.

Let us consider matrices A_1, A_2, \dots, A_5 to illustrate MATRIX-CHAIN-MULTIPLICATION algorithm. The matrix chain order $P = \{P_0, P_1, P_2, P_3, P_4, P_5\} = \{5, 10, 3, 12, 5, 50\}$. The objective is to find the minimum number of scalar multiplications required to multiply the 5 matrices and also find the optimal sequence of multiplications.

The solution can be obtained by using a bottom up approach that means first we should calculate m_{ij} for $1 \leq i \leq 5$. Then m_{ij} is calculated for $j - i = 1$ to $j - i = 4$. We can fill the table shown in Fig. 7.4 to find the solution.

The value of m_{ij} for $1 \leq i \leq 5$ can be filled as 0 that means the elements in the first row can be assigned 0. Then

For $j - i = 1$

$$m_{12} = P_0 P_1 P_2 = 5 \times 10 \times 3 = 150$$

$$m_{23} = P_1 P_2 P_3 = 10 \times 3 \times 12 = 360$$

$$m_{34} = P_2 P_3 P_4 = 3 \times 12 \times 5 = 180$$

$$m_{45} = P_3 P_4 P_5 = 12 \times 5 \times 50 = 3000$$

For $j - i = 2$

$$\begin{aligned} m_{13} &= \min \{m_{11} + m_{23} + P_0 P_1 P_3, m_{12} + m_{33} + P_0 P_2 P_3\} \\ &= \min \{0 + 360 + 5 \times 10 \times 12, 150 + 0 + 5 \times 3 \times 12\} \\ &= \min \{360 + 600, 150 + 180\} = \min \{960, 330\} = 330 \\ m_{24} &= \min \{m_{22} + m_{34} + P_1 P_2 P_4, m_{23} + m_{44} + P_1 P_3 P_4\} \\ &= \min \{0 + 180 + 10 \times 3 \times 5, 360 + 0 + 10 \times 12 \times 5\} \\ &= \min \{180 + 150, 360 + 600\} = \min \{330, 960\} = 330 \\ m_{35} &= \min \{m_{33} + m_{45} + P_2 P_3 P_5, m_{34} + m_{55} + P_2 P_4 P_5\} \\ &= \min \{0 + 3000 + 3 \times 12 \times 50, 180 + 0 + 3 \times 5 \times 50\} \\ &= \min \{3000 + 1800 + 180 + 750\} = \min \{4800, 930\} = 930 \end{aligned}$$

For $j - i = 3$

$$\begin{aligned} m_{14} &= \min \{m_{11} + m_{24} + P_0 P_1 P_4, m_{12} + m_{34} + P_0 P_2 P_4, m_{13} + m_{44} + P_0 P_3 P_4\} \\ &= \min \{0 + 330 + 5 \times 10 \times 5, 150 + 180 + 5 \times 3 \times 5, 330 + 0 + 5 \times 12 \times 5\} \\ &= \min \{330 + 250, 150 + 180 + 75, 330 + 300\} \\ &= \min \{580, 405, 630\} = 405 \\ m_{25} &= \min \{m_{22} + m_{35} + P_1 P_2 P_5, m_{23} + m_{45} + P_1 P_3 P_5, m_{24} + m_{55} + P_1 P_4 P_5\} \\ &= \min \{0 + 930 + 10 \times 3 \times 50, 360 + 3000 + 10 \times 12 \times 50, 330 + 0 + 10 \times 5 \times 50\} \\ &= \min \{930 + 1500, 360 + 3000 + 6000, 330 + 2500\} \\ &= \min \{2430, 9360, 2830\} = 2430 \end{aligned}$$

For $j - i = 4$

$$\begin{aligned} m_{15} &= \min \{m_{11} + m_{25} + P_0 P_1 P_5, m_{12} + m_{35} + P_0 P_2 P_5, m_{13} + m_{45} + P_0 P_3 P_5, m_{14} + m_{55} + P_0 P_4 P_5\} \\ &= \min \{0 + 2430 + 5 \times 10 \times 50, 150 + 930 + 5 \times 3 \times 50, 330 + 3000 + 5 \times 12 \times 50, 405 + 0 + 5 \times 5 \times 50\} \end{aligned}$$

$$\begin{aligned}
&= \min \{2430+2500, 150+930+750, 330+3000+3000, 405+1250\} \\
&= \min \{4930, 1830, 6330, 1655\} = 1655
\end{aligned}$$

Hence, minimum number of scalar multiplications required to multiply the given five matrices in 1655.

To find the optimal parenthesization of $A_1 \dots A_5$, we find the value of k is 4 for which m_{15} is minimum. So the matrices can be splitted to $(A_1 \dots A_4) (A_5)$. Similarly, $(A_1 \dots A_4)$ can be splitted to $(A_1 A_2) (A_3 A_4)$ because for $k = 2$, m_{14} is minimum. No further splitting is required as the sub chains $(A_1 A_2)$ and $(A_3 A_4)$ has length 1. So the optimal paranthesization of $A_1 \dots A_5$ in $((A_1 A_2) (A_3 A_4)) (A_5)$.

Time complexity of multiplying a chain of n matrices

Let $T(n)$ be the time complexity of multiplying a chain of n matrices.

$$\begin{aligned}
T(n) &= \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(1) + \sum_{k=1}^{n-1} [T(k) + T(n-k) + \Theta(1)] & \text{if } n > 1 \end{cases} \\
\Rightarrow T(n) &= \Theta(1) + \sum_{k=1}^{n-1} [T(k) + T(n-k) + \Theta(1)] \quad \text{if } n > 1 \\
&= \Theta(1) + \Theta(n-1) + \sum_{k=1}^{n-1} [T(k) + T(n-k)] \\
\Rightarrow T(n) &= \Theta(n) + 2[T(1) + T(2) + \dots + T(n-1)] \quad \text{--- (7.1)}
\end{aligned}$$

Replacing n by $n-1$, we get

$$T(n-1) = \Theta(n-1) + 2[T(1) + T(2) + \dots + T(n-2)] \quad \text{--- (7.2)}$$

Subtracting equation 7.2 from equation 7.1, we have

$$\begin{aligned}
T(n) - T(n-1) &= \Theta(n) - \Theta(n-1) + 2T(n-1) \\
\Rightarrow T(n) &= \Theta(1) + 3T(n-1) \\
&= \Theta(1) + 3[\Theta(1) + 3T(n-2)] = \Theta(1) + 3\Theta(1) + 3^2 T(n-2) \\
&= \Theta(1) [1 + 3 + 3^2 + \dots + 3^{n-2}] + 3^{n-1} T(1) \\
&= \Theta(1) [1 + 3 + 3^2 + \dots + 3^{n-1}] \\
&= \frac{3^n - 1}{2} = O(2^n)
\end{aligned}$$

Lecture 14 - Longest Common Subsequence

Longest Common Subsequence

The longest common subsequence (LCS) problem can be formulated as follows “Given two sequences $X = \langle x_1, x_2, \dots, x_n \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ and the objective is to find the LCS $Z = \langle z_1, z_2, \dots, z_n \rangle$ that is common to x and y ”.

Given two sequences X and Y , we say Z is a common subsequence of X and Y if Z is a subsequence of both X and Y . For example, $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence. Similarly, there are many common subsequences in the two sequences X and Y . However, in the longest common subsequence problem, we wish to find a maximum length common subsequence of X and Y , that is $\langle B, C, B, A \rangle$ or $\langle B, D, A, B \rangle$. This section shows that the LCS problem can be solved efficiently using dynamic programming.

Theorem. 4. 1. (Optimal Structure of an LCS)

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences and let $Z = \langle z_1, z_2, \dots, z_n \rangle$ be any LCS of X and Y .

Case 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .

Case 2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .

Case 3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

Proof The proof of the theorem is presented below for all three cases.

Case 1. If $x_m = y_n$ and we assume that $z_k \neq x_m$ or $z_k \neq y_n$ then $x_m = y_n$ can be added to Z at any index after k violating the assumption that Z_k is the longest common subsequence. Hence $z_k = x_m = y_n$. If we do not consider Z_{k-1} as LCS of X_{m-1} and Y_{n-1} , then there may exist another subsequence W whose length is more than $k-1$. Hence, after adding $x_m = y_n$ to the subsequence W increases the size of subsequence more than k , which again violates our assumption.

Hence, $Z_k = x_m = y_n$ and Z_{k-1} must be an LCS of X_{m-1} and Y_{n-1} .

Case 2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y_n . If there were a common subsequence W of X_{m-1} and Y with length greater than k than W would also be an LCS of X_m and Y_n violating our assumption that Z_k is an LCS of X_m and Y_n .

Case 3. The proof is symmetric to case-2.

Thus the LCS problem has an optimal structure.

Overlapping Sub-problems

From theorem 4.1, it is observed that either one or two cases are to be examined to find an LCS of X_m and Y_n . If $x_m = y_n$, then we must find an LCS of X_{m-1} and Y_{n-1} . If $x_m \neq y_n$, then we must find an LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1} . The LCS of X and Y is the longer of these two LCSs.

Let us define $c[m, n]$ to be the length of an LCS of the sequences X_m and Y_n . The optimal structure of the LCS problem gives the recursive formula

$$c[m, n] = \begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ c[m-1, n-1] + 1 & \text{if } x_m = y_n \end{cases} \dots\dots\dots (7.1)$$

$$\begin{cases} \max \{c[m-1, n], c[m, n-1]\} & \text{if } x_m \neq y_n \end{cases}$$

Generalizing equation 7.1, we can formulate

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max \{c[i-1, j], c[i, j-1]\} & \text{if } x_i \neq y_j \end{cases} \dots\dots\dots (7.2)$$

Algorithm LCS_LENGTH (X, Y)

```
{
     $m = \text{length}[X]$ 

     $n = \text{length}[Y]$ 

    for( $i = 1; i \leq m; i++$ )
         $c[i, 0] = 0;$ 

    for( $j = 0; j < n; j++$ )
         $c[0, j] = 0;$ 

    for( $i = 1; i < m; i++$ ){

        for( $j = 1; j \leq n; j++$ ){

            if( $x[i] = y[j]$ ){

                 $c[i, j] = 1 + c[i-1, j-1];$ 
                 $b[i, j] = \text{'\textasciitilde'};$ 
                ↖

            }

            else{

                if( $c[i-1, j] \geq c[i, j-1]$ )

                     $c[i, j] = c[i-1, j];$ 

                     $b[i, j] = \text{'\textasciitilde'};$ 

                else

                     $c[i, j] = c[i, j-1];$ 
                     $b[i, j] = \text{'\textasciitilde'};$ 

            }

        }

    }

    return  $c$  and  $b$  ;

}
```

Algorithm 7.3 Algorithm for finding Longest common subsequence .**Constructing an LCS**

The algorithm LCS_LENGTH returns c and b tables. The b table can be used to construct the LCS of X and Y

quickly.

Algorithm PRINT_LCS (b, X, i, j)

```
{
    if ( $i == 0 \mid j == 0$ )
        return;

    if ( $b[i, j] == \uparrow$ ) {
        PRINT_LCS ( $b, X, i-1, j-1$ )

        Print  $x_i$ 
    }

    else if ( $b[i, j] == \leftarrow$ )
        PRINT_LCS ( $b, X, i-1, j$ )

    else
        PRINT_LCS ( $b, X, i, j-1$ )
}
```

Algorithm 7.4 Algorithm to print the Longest common subsequence .

Let us consider two sequences $X = \langle C, R, O, S, S \rangle$ and $Y = \langle R, O, A, D, S \rangle$ and the objective is to find the LCS and its length. The c and b table are computed by using the algorithm LCS_LENGTH for X and Y that is shown in Fig. 7.5. The longest common subsequence of X and Y is $\langle R, O, S \rangle$ and the length of LCS is 3.

Lecture 15 - Greedy Algorithms

Greedy Method

Introduction

Let we are given a problem to sort the array $a = \{5, 3, 2, 9\}$. Someone says the array after sorting is $\{1, 3, 5, 7\}$. Can we consider the answer is correct? The answer is definitely “no” because the elements of the output set are not taken from the input set. Let someone says the array after sorting is $\{2, 5, 3, 9\}$. Can we admit the answer? The answer is again “no” because the output is not satisfying the objective function that is the first element must be less than the second, the second element must be less than the third and so on. Therefore, the solution is said to be a feasible solution if it satisfies the following constraints.

- (i) **Explicit constraints:** - The elements of the output set must be taken from the input set.
- (ii) **Implicit constraints:** - The objective function defined in the problem.

The best of all possible solutions is called the optimal solution. In other words we need to find the solution which has the optimal (maximum or minimum) value satisfying the given constraints.

The Greedy approach constructs the solution through a sequence of steps. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

Let us consider the problem of coin change. Suppose a greedy person has some 25p, 20p, 10p, 5paise coins. When someone asks him for some change then he wants to give the change with minimum number of coins. Now, let someone requests for a change of 45p then he first selects 25p. Then the remaining amount is 20p. Next, he selects the largest coin that is less than or equal to 20p i.e. 20p. The remaining 0p is paid by selecting a 0p coin. So the demand for 45p is paid by giving total 2 numbers of coins. This solution is an optimal solution. Now, let someone requests for a change of 40p then the Greedy approach first selects 25p coin, then a 10p coin and finally a 5p coin. However, the same could be paid with two 20p coins. So it is clear from this example that Greedy approach tries to find the optimal solution by selecting the elements one by one that are locally optimal. But Greedy method never gives the guarantee to find the optimal solution.

The choice of each step in a greedy approach is done based on the following:

- It must be feasible
- It must be locally optimal
- It must be unalterable

Lecture 16 - Activity Selection Problem

Activity Selection Problem

Suppose we have a set of activities $S = \{a_1, a_2, \dots, a_n\}$ that wish to use a common resource. The objective is to schedule the activities in such a way that maximum number of activities can be performed. Each activity a_i has a start time s_i and a finish time f_i , where $0 \leq s_i < f_i < \infty$. The activities a_i and a_j are said to be compatible if the intervals $[s_i, f_i]$ and $[s_j, f_j]$ do not overlap that means $s_i \geq f_j$ or $s_j \geq f_i$.

For example, let us consider the following set S of activities, which are sorted in monotonically increasing order of finish time.

i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

For this example, the subsets $\{a_3, a_9, a_{11}\}$, $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$ consist of mutually compatible activities. We have two largest subsets of mutually compatible activities.

Now, we can devise greedy algorithm that works in a top down fashion. We assume that the n input activities are ordered by monotonically increasing finish time or it can be sorted into this order in $O(n \log n)$ time. The greedy algorithm for activity selection problem is given below.

Algorithm ACTIVITY SELECTION (S, f)

```
{
    n = LENGTH (S) ; // n is the total number of activities //

    A = {a1} ; // A is the set of selected activities and initialized to a1//
    i = 1 ; // i represents the recently selected activity //
    for (j = 2 ; j <= n ; j++)

    {
        if (sj ≥ fi) {
            A = A ∪ {am} ;

            i = j ;
        }
    }
```

```

    }
    Return A ;
}

```

Algorithm 3. Algorithm of activity selection problem.

The algorithm takes the start and finish times of the activities, represented as arrays s and f , length (s) gives the total number of activities. The set A stores the selected activities. Since the activities are ordered with respect to their finish times the set A is initialized to contain just the first activity a_1 . The variable i stores the index of the recently selected activity. The for loop considers each activity and adds to the set A if it is mutually compatible with the previously selected activities. To see whether activity a_j is compatible with every activity assumingly in A , it needs to check whether the short time of a_j is greater or equal to the finish time of the recently selected activity a_i .

Lecture 17 - Elements of Greedy Strategy


Greedy strategy 1: In this case, the items are arranged by their profit values. Here the item with maximum profit is selected first. If the weight of the object is less than the remaining capacity of the knapsack then the object is selected full and the profit associated with the object is added to the total profit. Otherwise, a fraction of the object is selected so that the knapsack can be filled exactly. This process continues from selecting the highest profitable object to the lowest profitable object till the knapsack is exactly full.

Greedy strategy II: In this case, the items are arranged by fair weights. Here the item with minimum weight is selected first and the process continues like greedy strategy-I till the knapsack is exactly full.

Greedy strategy III: In this case, the items are arranged by profit/weight ratio and the item with maximum profit/weight ratio is selected first and the process continues like greedy strategy-I till the knapsack is exactly full.

Therefore, it is clear from the above strategies that the **Greedy method** generates optimal solution if we select the objects with respect to their profit to weight ratios that means the object with maximum profit to weight ratio will be selected first. Let there are n objects and the object i is associated with

profit p_i and weight w_i . Then we can say that if $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$, the solution


 $(x_1, x_2, x_3 \llcorner x_n)$ generated by greedy method is an optimal solution. The proof of the above statement is left as an exercise for the readers. The algorithm 6.1 describes the greedy method for finding the optimal solution for fractional knapsack problem.

Algorithm FKNAPSACK (p, w, x, n, M)

// $p[1:n]$ and $w[1:n]$ contains the profit and weight of n objects. Mis the maximum capacity of knapsack and $x[1:n]$ in the solution vector.//

 $\{$

```
for (i = 1; i <= n; i++)
```

```
x[i]=0; // initialize the solution to 0 //
```

```
cu = M // cu is the remaining capacity of the knapsack//
```

```
for (i =1; i<= n ; i ++){
```

```
if( $w[i] > cu$  )
```

```
        break;
    else{
```

$$x[i] = 1 ;$$

```
         $cu = cu - w[i] ;$   
  
    }  
}  
if(  $i \leq n$ ){  
  
     $x[i] = cu/w[i] ;$   
  
    return x;  
  
}
```

Lecture 18-19 - Fractional Knapsack Problem

Fractional Knapsack Problem

Let there are n number of objects and each object is having a weight and contribution to profit. The knapsack of capacity M is given. The objective is to fill the knapsack in such a way that profit shall be maximum. We allow a fraction of item to be added to the knapsack.

Mathematically, we can write

$$\text{maximize } \sum_{i=1}^n p_i x_i$$

Subject to

$$\sum_{i=1}^n w_i x_i \leq M$$

$$1 \leq i \leq n \text{ and } 0 \leq x_i \leq 1.$$

Where p_i and w_i are the profit and weight of i^{th} object and x_i is the fraction of i^{th} object to be selected.

For example

Given $n = 3$, $(p_1, p_2, p_3) = \{25, 24, 15\}$

$(w_1, w_2, w_3) = \{18, 15, 10\}$ $M = 20$

Solution

Some of the feasible solutions are shown in the following table.

<i>Solution No</i>	x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	1	2/15	0	20	28.2
2	0	2/3	1	20	31.0
3	0	1	1/2	20	31.5

These solutions are obtained by different greedy strategies.

Lecture 20 - Huffman Codes

Huffman Coding

Each character is represented in 8 bits when characters are coded using standard codes such as ASCII. It can be seen that the characters coded using standard codes have fixed-length code word representation. In this fixed-length coding system the total code length is more. For example, let we have six characters (a, b, c, d, e, f) and their frequency of occurrence in a message is {45, 13, 12, 16, 9, 5}. In fixed-length coding system we can use three characters to represent each code. Then the total code length of the message is $(45+13+12+16+9+5) \times 3 = 100 \times 3 = 300$.

Let us encode the characters with variable-length coding system. In this coding system, the character with higher frequency of occurrence is assigned fewer bits for representation while the characters having lower frequency of occurrence are assigned more bits for representation. The variable length code for the characters are shown in the following table. The total code length in variable length coding system is $1 \times 45 + 3 \times 12 + 3 \times 16 + 4 \times 9 + 4 \times 5 = 224$. Hence fixed length code requires 300 bits while variable code requires only 224 bits.

a	b	c	d	e	f
0	101	100	111	1101	1100

Prefix (Free) Codes

We have seen that using variable-length code word we minimize the overall encoded string length. But the question arises whether we can decode the string. If *a* is encoded 1 instead of 0 then the encoded string "111" can be decoded as "d" or "aaa". It can be seen that we get ambiguous string. The key point to remove this ambiguity is to use prefix codes. Prefix codes are the code in which there is no codeword that is a prefix of other codeword.

The representation of "decoding process" is binary tree whose leaves are characters. We interpret the binary codeword for a character as path from the root to that character, where

⇒ "0" means "go to the left child"

⇒ "1" means "go to the right child"

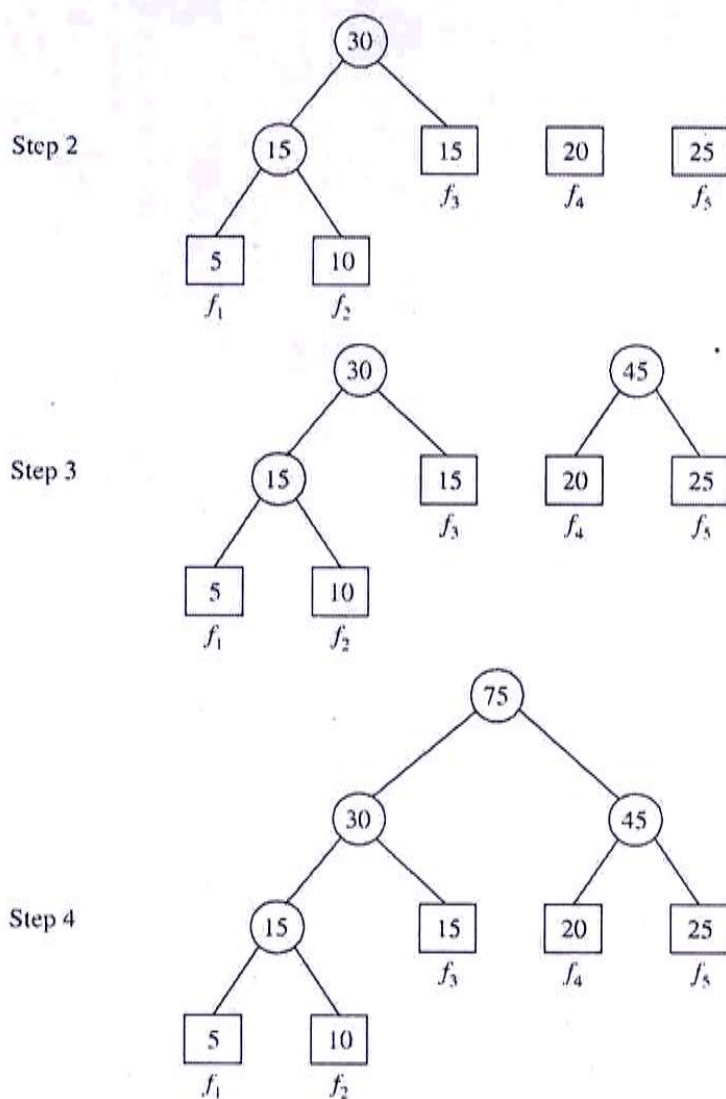
Greedy Algorithm for Huffman Code:

According to Huffman algorithm, a bottom up tree is built starting from the leaves. Initially, there are *n* singleton trees in the forest, as each tree is a leaf. The greedy strategy first finds two trees having minimum frequency of occurrences. Then these two trees are merged in a single tree where the frequency of this tree is the total sum of two merged trees. The whole process is repeated until there is only one tree in the forest.

Let us consider a set of characters $S = \{a, b, c, d, e, f\}$ with the following frequency of occurrences $P =$

$\{45, 13, 12, 16, 5, 9\}$. Initially, these six characters with their frequencies are considered six singleton trees in the forest. The step wise merging these trees to a single tree is shown in Fig. 6.3. The merging is done by selecting two trees with minimum frequencies till there is only one tree in the forest

a : 45	b : 13	c : 12	d : 16	e : 5	f : 9
--------	--------	--------	--------	-------	-------



Step wise merging of the singleton trees.

Now the left branch is assigned a code "0" and right branch is assigned a code "1". The decode tree after assigning the codes to the branches.

The binary codeword for a character is interpreted as path from the root to that character; Hence, the codes for the characters are as follows

$a = 0$

$b = 101$

$c = 100$

$d = 111$

$e = 1100$

$f = 1101$

Therefore, it is seen that no code is the prefix of other code. Suppose we have a code 01111001101. To decode the binary codeword for a character, we traverse the tree. The first character is 0 and the character at which the tree traversal terminates is *a*. Then, the next bit is 1 for which the tree is traversed right. Since it has not reached at the leaf node, the tree is next traversed right for the next bit 1. Similarly, the tree is traversed for all the bits of the code string. When the tree traversal terminates at a leaf node, the tree traversal again starts from the root for the next bit of the code string. The character string after decoding is “*adcf*”.

Algorithm HUFFMAN(n, S)

```
{
    //  $n$  is the number of symbols and  $S$  in the set of characters, for each character  $c \in S$ , the frequency of
    Occurrence in  $f(c)$  //

    Initialize the priority queue;

     $Q = S$ ; // Initialize the priority  $Q$  with the frequencies of all the characters of set  $S$  //

    for( $i = 1$  ;  $i \leq n-i$ ,  $i++$ ){

         $z = \text{CREAT\_NODE}()$ ;          // create a node pointed by  $z$ ; //

        // Delete the character with minimum frequency from the  $Q$  and store in node  $x$  //
         $x = \text{DELETE\_MIN}(Q)$ ;
        // Delete the character with next minimum frequency from the  $Q$  and store in node  $y$  //
         $y = \text{DELETE\_MIN}(Q)$ ;
         $z \rightarrow \text{left} = x$ ;          // Place  $x$  as the left child of  $z$  //

         $z \rightarrow \text{right} = y$ ;       // Place  $y$  as the right child of  $z$  //

        // The value of node  $z$  is the sum of values at node  $x$  and node  $y$  //

         $f(z) = f(x) + f(y)$ ;

        // insert  $z$  into the priority  $Q$  //

         $\text{INSERT}(Q, z)$ ;

    }

    Return  $\text{DELETE\_MIN}(Q)$ 
```

Lecture – 21 Disjoint Set Data Structure

In computing, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

It supports the following useful operations:

- *Find*: Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.
- *Union*: Join two subsets into a single subset.
- *Make Set*, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved.

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find*(x) returns the representative of the set that x belongs to, and *Union* takes two set representatives as its arguments.

Example :



Make Set creates 8 singletons.



After some operations of *Union*, some sets are grouped together.

Applications:

- partitioning of a set
- Boost Graph Library to implement its Incremental Connected Components functionality.
- Implementing Kruskal's algorithm to find the minimum spanning tree of a graph.
- Determine the connected components of an undirected graph.

CONNECTED-COMPONENTS(G)

1. **for** each vertex $v \in V[G]$
2. **do** MAKE-SET(v)
3. **for** each edge $(u, v) \in E[G]$
4. **do if** FIND-SET(u) \neq FIND-SET(v)
5. **then** UNION(u, v)

•

6. **for** each vertex $v \in V[G]$
7. **do** MAKE-SET(v)
8. **for** each edge $(u, v) \in E[G]$
9. **do** if FIND-SET(u) \neq FIND-SET(v)
10. **then** UNION(u, v)

SAME-COMPONENT (u, v)

1. **if** FIND-SET(u)=FIND-SET(v)
2. **then** return TRUE

else

Lecture 22 - Disjoint Set Operations, Linked list Representation

- A disjoint-set is a collection $\mathcal{C} = \{S_1, S_2, \dots, S_k\}$ of distinct dynamic sets.
- Each set is identified by a member of the set, called *representative*.

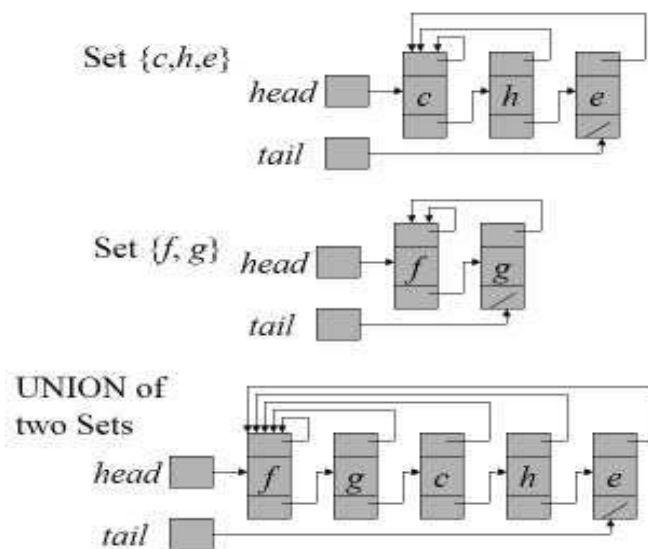
Disjoint set operations

- MAKE-SET(x): create a new set with only x . assume x is not already in some other set.
- UNION(x, y): combine the two sets containing x and y into one new set. A new representative is selected.
- FIND-SET(x): return the representative of the set containing x .

Linked list Representation

- Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.
- Example:
- MAKE-SET costs $O(1)$: just create a single element list.
- FIND-SET costs $O(1)$: just return back-to-representative pointer.

Linked-lists for two sets



UNION Implementation

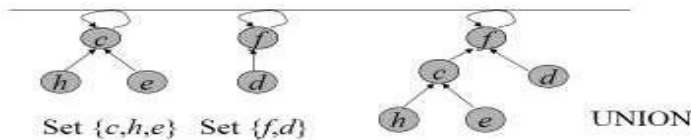
- A simple implementation: $\text{UNION}(x,y)$ just appends x to the end of y , updates all back-to-representative pointers in x to the head of y .
- Each UNION takes time linear in the x 's length.
- Suppose n MAKE-SET(x_i) operations ($O(1)$ each) followed by $n-1$ UNION
 - $\text{UNION}(x_1, x_2), O(1),$
 - $\text{UNION}(x_2, x_3), O(2),$
 -
 - $\text{UNION}(x_{n-1}, x_n), O(n-1)$
- The UNIONs cost $1+2+\dots+n-1=\Theta(n^2)$

So $2n-1$ operations cost $\Theta(n^2)$, average $\Theta(n)$ each

Lecture 23 - Disjoint Forests

Disjoint-set Implementation: Forests

- Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.

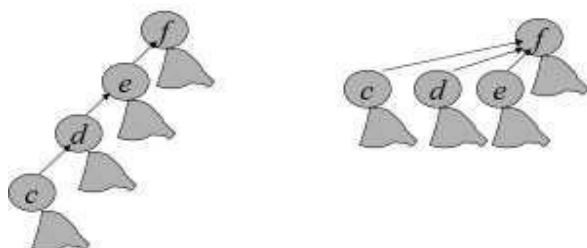


- Three operations
 - MAKE-SET(x): create a tree containing x . $O(1)$
 - FIND-SET(x): follow the chain of parent pointers until to the root. $O(\text{height of } x\text{'s tree})$
 - UNION(x, y): let the root of one tree point to the root of the other. $O(1)$
- It is possible that $n-1$ UNIONS results in a tree of height $n-1$. (just a linear chain of n nodes).
- So n FIND-SET operations will cost $O(n^2)$.

Union by Rank & Path Compression

- Union by Rank:** Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of sub tree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.
- Path Compression:** used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.

Path Compression



Algorithm for Disjoint-Set Forest

MAKE-SET(x) 1. $p[x] \leftarrow x$ 2. $rank[x] \leftarrow 0$	UNION(x, y) 1. LINK (FIND-SET (x), FIND-SET (y))	FIND-SET(x) 1. if $x \neq p[x]$ 2. then $p[x] \leftarrow \text{FIND-SET}(p[x])$ 3. return $p[x]$
	LINK(x, y) 1. if $rank[x] > rank[y]$ 2. then $p[y] \leftarrow x$ 3. else $p[x] \leftarrow y$ 4. if $rank[x] = rank[y]$ 5. then $rank[y]++$	

Worst case running time for m MAKE-SET, UNION, FIND-SET operations is:
 $O(m\alpha(n))$ where $\alpha(n) \leq 4$. So nearly linear in m .