R.PUSHPANATHAN.,MCA.,M.Phil.,MBA

DYNAMIC PROGRAMMING

What is Dynamic Programming

Dynamic programming is a method for solving optimization problems. It is algorithm technique to solve a complex and overlapping sub-problems. Compute the solutions to the sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later. Dynamic programming is more efficient then other algorithm methods like as Greedy method, Divide and Conquer method, Recursion method, etc….

Why ? the Dynamic programming needed

The real time many of problems are not solve using simple and traditional approach methods. like as coin change problem , knapsack problem, Fibonacci sequence generating , complex matrix multiplication….To solve using Iterative formula, tedious method , repetition again and again it become a more time consuming and foolish. some of the problem it should be necessary to divide a sub problems and compute its again and again to solve a such kind of problems and give the optimal solution , effective solution the Dynamic programming is needed…

Basic Features of Dynamic programming :-

Get all the possible solution and pick up best and optimal solution.

Work on principal of optimality.

Define sub-parts and solve them using recursively.

Less space complexity But more Time complexity.

Dynamic programming saves us from having to recompute previously calculated sub-solutions.

Difficult to understanding.

Let’s Discuss a matrix chain multiplication problem using Dynamic Programming :-

We are covered a many of the real world problems.In our day to day life when we do making coin change, robotics world, aircraft, mathematical problems like Fibonacci sequence, simple matrix multiplication of more then two matrices and its multiplication possibility is many more so in that get the best and optimal solution. NOW we can look about one problem that is MATRIX CHAIN MULTIPLICATION PROBLEM.

Suppose, We are given a sequence (chain) (A1, A2……An) of n matrices to be multiplied, and we wish to compute the product (A1A2…..An).We can evaluate the above expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is (A1, A2, A3, A4) then we can fully parenthesize the product (A1A2A3A4) in five distinct ways:

1:-(A1(A2(A3A4))) ,

2:-(A1((A2A3)A4)),

3:- ((A1A2)(A3A4)),

4:-((A1(A2A3))A4),

5:-(((A1A2)A3)A4) .

We can multiply two matrices A and B only if they are compatible. the number of columns of A must equal the number of rows of B. If A is a p x q matrix and B is a q x r matrix,the resulting matrix C is a p x r matrix. The time to compute C is dominated by the number of scalar multiplications is pqr. we shall express costs in terms of the number of scalar multiplications.For example, if we have three matrices (A1,A2,A3) and its cost is (10x100),(100x5),(5x500) respectively. so we can calculate the cost of scalar multiplication is 10*100*5=5000 if ((A1A2)A3), 10*5*500=25000 if (A1(A2A3)), and so on cost calculation. Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.that is here is minimum cost is 5000 for above example .So problem is we can perform a many time of cost multiplication and repeatedly the calculation is performing. so this general method is very time consuming and tedious.So we can apply dynamic programming for solve this kind of problem.

when we used the Dynamic programming technique we shall follow some steps.

- **Characterize the structure of an optimal solution.**
- **Recursively define the value of an optimal solution.**
- **Compute the value of an optimal solution.**
- **Construct an optimal solution from computed information.**

we have matrices of any of order. our goal is find optimal cost multiplication of matrices.when we solve the this kind of problem using DP step 2 we can get

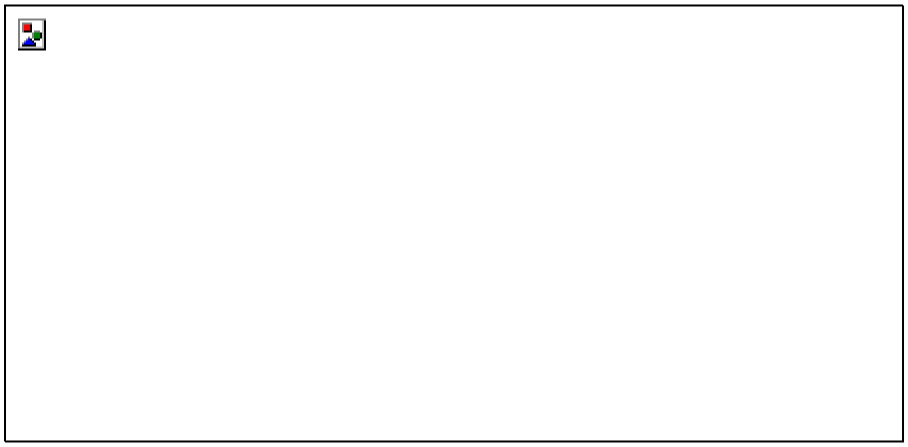
m[i , j] = min { m[i , k] + m[i+k , j] + pi-1*pk*pj } if $i < j$…. where p is dimension of matrix , $i \leq k < j$ …..

The basic algorithm of matrix chain multiplication:-

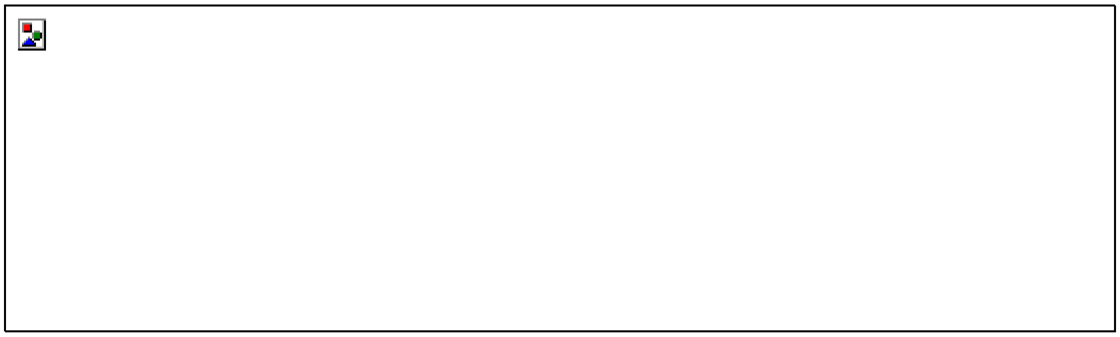
```
// Matrix A[i] has dimension dims[i-1] x dims[i] for i = 1..n

MatrixChainMultiplication(int dims[])

{

// length[dims] = n + 1

n = dims.length - 1;

// m[i,j] = Minimum number of scalar multiplications(i.e., cost)

// needed to compute the matrix A[i]A[i+1]...A[j] = A[i..j]
// The cost is zero when multiplying one matrix
for (i = 1; i <= n; i++)
    m[i, i] = 0;
for (len = 2; len <= n; len++){
    // Subsequence lengths
    for (i = 1; i <= n - len + 1; i++) {
        j = i + len - 1;
        m[i, j] = MAXINT;
        for (k = i; k <= j - 1; k++) {
            cost = m[i, k] + m[k+1, j] + dims[i-1]*dims[k]*dims[j];
            if (cost < m[i, j]) {
                m[i, j] = cost;
                s[i, j] = k;
                // Index of the subsequence split that achieved minimal cost
            }
        }
    }
}
}
```

Example of Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.



Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



In Dynamic Programming, initialization of every method done by ‘0’.So we initialize it by ‘0’.It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

1. m (1,2) = m1 x m2

   = 4 x 10 x 10 x 3

= 4 x 10 x 3 = 120

2. m (2, 3) = m2 x m3

= 10 x 3 x 3 x 12

= 10 x 3 x 12 = 360

3. m (3, 4) = m3 x m4

= 3 x 12 x 12 x 20

= 3 x 12 x 20 = 720

4. m (4,5) = m4 x m5

= 12 x 20 x 20 x 7

= 12 x 20 x 7 = 1680

We initialize the diagonal element with equal i,j value with ‘0’.

After that second diagonal is sorted out and we get all the values corresponded to it

Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

M [1, 3] = M1 M2 M3

There are two cases by which we can solve this multiplication: (M1 x M2) + M3, M1+ (M2x M3)

After solving both cases we choose the case in which minimum output is there.

M [1, 3] =264

As Comparing both output 264 is minimum in both cases so we insert 264 in table and (M1 x M2) + M3 this combination is chosen for the output making.

M [2, 4] = M2 M3 M4

There are two cases by which we can solve this multiplication: (M2x M3)+M4, M2+(M3 x M4)

After solving both cases we choose the case in which minimum output is there.

M [2, 4] = 1320

As Comparing both output 1320 is minimum in both cases so we insert 1320 in table and M2+(M3 x M4) this combination is chosen for the output making.

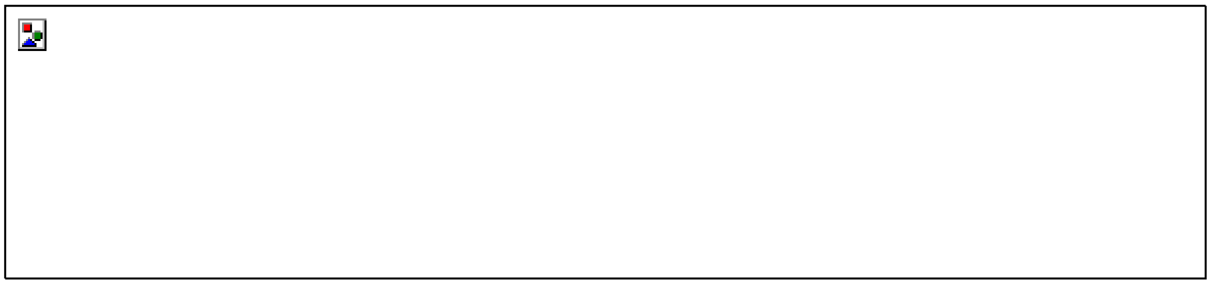
M [3, 5] = M3 M4 M5

There are two cases by which we can solve this multiplication: (M3 x M4) + M5, M3+ (M4xM5)

After solving both cases we choose the case in which minimum output is there.

M [3, 5] = 1140

As Comparing both output 1140 is minimum in both cases so we insert 1140 in table and (M3 x M4) + M5this combination is chosen for the output making.



Now Product of 4 matrices:

M [1, 4] = M1 M2 M3 M4

There are three cases by which we can solve this multiplication:

(M1 x M2 x M3) M4

M1 x(M2 x M3 x M4)

(M1 xM2) x (M3 x M4)

After solving these cases we choose the case in which minimum output is there

M [1, 4] =1080

As comparing the output of different cases then '1080' is minimum output, so we insert 1080 in the table and (M1 xM2) x (M3 x M4) combination is taken out in output making,

M [2, 5] = M2 M3 M4 M5

There are three cases by which we can solve this multiplication:

(M2 x M3 x M4)x M5

M2 x(M3 x M4 x M5)

(M2 x M3)x (M4 x M5)

After solving these cases we choose the case in which minimum output is there

M [2, 5] = 1350

As comparing the output of different cases then '1350' is minimum output, so we insert 1350 in the table and M2 x(M3 x M4xM5)combination is taken out in output making.

Now Product of 5 matrices:

M [1, 5] = M1 M2 M3 M4 M5

There are five cases by which we can solve this multiplication:

(M1 x M2 xM3 x M4)x M5

M1 x(M2 xM3 x M4 xM5)

(M1 x M2 xM3)x M4 xM5

M1 x M2x(M3 x M4 xM5)

After solving these cases we choose the case in which minimum output is there

M [1, 5] = 1344

As comparing the output of different cases then ‘1344’ is minimum output, so we insert 1344 in the table and M1 x M2 x(M3 x M4 x M5)combination is taken out in output making.

Final Output is:

Elements of Dynamic Programming

We have done an example of dynamic programming: the matrix chain multiply problem, but what can be said, in general, to guide us to choosing DP?

- **Optimal Substructure**
- **Overlapping Sub-problems**
- **Variant: Memoization**

Optimal Substructure:

OS holds if optimal solution contains within it optimal solutions to sub problems. In matrix-chain multiplication optimally doing A1, A2, A3, ...,An required A 1...k and A k+1 ...n to be optimal. It is often easy to show the optimal sub problem property as follows:

- **Split problem into sub-problems**
- **Sub-problems must be optimal, otherwise the optimal splitting would not have been optimal.**

There is usually a suitable "space" of sub-problems. Some spaces are more "natural" than others.

For matrix-chain multiply we chose sub-problems as sub chains. We could have chosen all arbitrary products, but that would have been much larger than necessary! DP based on that would have to solve too many sub-problems.

A general way to investigate optimal substructure of a problem in DP is to look at optimal sub-, sub-sub, etc. problems for structure. When we noticed that sub problems of A1, A2, A3, ...,An consisted of sub-chains, it made sense to use sub-chains of the form Ai, ..., Aj as the "natural" space for sub-problems.

Overlapping Sub-problems:

Space of sub-problems must be small: recursive solution re-solves the same sub-problem many times. Usually there are polynomially many sub-problems, and we revisit the same ones over and over again: overlapping sub-problems.

Recursive-Matrix-Chain (p, i, j) (inefficient recursive program)

```
if i = j
  then return 0
m[i,j]
for k i to j - 1
```

```
   do q Recursive-Matrix-Chain (p, i, k )  + Recursive-Matrix-Chain (p, k + 1, j) + pi-1 pk pj
         if q < m[i,j]
            thenm[i,j] q
return m[i,j]
```

With divide-and-conquer, each sub-problem is new, in DP, most of the sub-problems are old. Thus storing solutions to sub-problems with DP in a lookup table saves loads of time.

T (n) = running time to compute m[1,n] by the above recursive algorithm:

i = j and q < m[i,j] unit time are running time assumptions

T (1) 1

T (n) 1 +

n - 1

k = 1   (T (k) + T (n - k) + 1)

2

n - 1

i = 1   T (i) + n

Will show T (n) 2n-1 (= W (2n))

Induction n = 1, T (1) 21-1 = 1

Assume true for up to n:

T (n) 2

n - 1

i = 1 2i-1 + n

= 2 (1 + 21 + 22 + ... + 2n-2) + n

= 2 ((2n-1 -1) / (2 - 1)) + n

= 2 (2n-1 - 1) + n

= 2n - 2 + n 2n-1 (true when n 2 !!)

Recall: DP Q (n2) sub-problems solved. Rule of thumb: Whenever a recursive approach solves the same problem repeatedly this signals a good DP candidate.

Memoization:

What if we stored sub-problems and used the stored solutions in a recursive algorithm? This is like divide-and-conquer, top down, but should benefit like DP which is bottom-up. Memoized version maintains an entry in a table. One can use a fixed table or a hash table.

Memoized-Matrix-Chain (p)

```
n length[p] - 1
for i 1 to n
  do for j i to n
    do m[i,j] (initialize to "undefined" table entries)
return Lookup-Chain (p, 1, n)
```

```
Lookup-Chain (p, i, j)
if m[i,j] <   (see if we know or not)
   then return m[i,j]
if i = j
   then m[i,j] 0
   else for k i to j - 1
      do q Lookup-Chain (p, i, k) + Lookup-Chain (p, k + 1, j) + pi-1 pk pj
            if q < m[i,j]
               then m[i,j] q
return m[i,j]
```

Each call is either:

O (1) if m[i,j] was previously computed

O (n) if not

Each m[i,j] called many times, but initialized only once. O (n n2) Memoization turns W (2n) O (n3)!

Longest Common Subsequence

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence

Let us consider a sequence S = <s1, s2, s3, s4, …,sn>.

A sequence Z = <z1, z2, z3, z4, …,zm> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, X and Y are two sequences over a finite set of elements. We can say that Z is a common subsequence of X and Y, if Z is a subsequence of both X and Y.

Longest Common Subsequence

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics. It is also widely used by revision control systems, such as SVN and Git, for reconciling multiple changes made to a revision-controlled collection of files.

Naïve Method

Let X be a sequence of length m and Y a sequence of length n. Check for every subsequence of X whether it is a subsequence of Y, and return the longest common subsequence found.

There are 2m subsequences of X. Testing sequences whether or not it is a subsequence of Y takes O(n) time. Thus, the naïve algorithm would take O(n2m) time.

Dynamic Programming

Let X = < x1, x2, x3,…, xm > and Y = < y1, y2, y3,…, yn > be the sequences. To compute the length of an element the following algorithm is used.

In this procedure, table C[m, n] is computed in row major order and another table B[m,n] is computed to construct optimal solution.

Algorithm: LCS-Length-Table-Formulation (X, Y)

```
m := length(X)
n := length(Y)
for i = 1 to m do
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if xi = yj
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := ‘D’
    else
      if C[i -1, j] ≥ C[i, j -1]
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := ‘U’
      else
      C[i, j] := C[i, j - 1]
      B[i, j] := ‘L’
return C and B
```

Algorithm: Print-LCS (B, X, i, j)

```
if i = 0 and j = 0
  return
if B[i, j] = ‘D’
  Print-LCS(B, X, i-1, j-1)
  Print(xi)
else if B[i, j] = ‘U’
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of X and Y.

Analysis

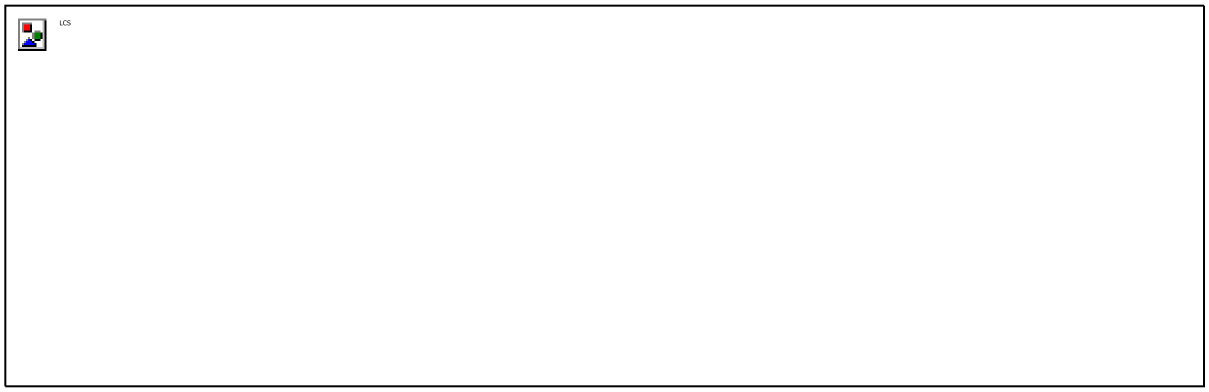
To populate the table, the outer for loop iterates m times and the inner for loop iterates n times. Hence, the complexity of the algorithm is O(m, n), where m and n are the length of two strings.

Example

In this example, we have two strings X = BACDB and Y = BDCB to find the longest common subsequence.

Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).

In table B, instead of ‘D’, ‘L’ and ‘U’, we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



Greedy Algorithms

Greedy Algorithms works step-by-step, and always chooses the steps which provide immediate profit/benefit. It chooses the "locally optimal solution", without thinking about future consequences. Greedy algorithms may not always lead to the optimal global solution, because it does not consider the entire data. The choice made by the greedy approach does not consider the future data and choices. In some cases making a decision that looks right at that moment gives the best solution (Greedy), but in other cases it doesn't. The Greedy technique is best suited for looking at the immediate situation.

All greedy algorithms follow a basic structure:

```
getOptimal(Item, arr[], int n)
 1) Initialize empty result : result = {}
 2) While (All items are not considered)
     // We make a greedy choice to select
     // an item.
     i = SelectAnItem()
     // If i is feasible, add i to the
     // result
     if (feasible(i))
       result = result U i
 3) return result
```

Why to choose Greedy Approach-

The greedy approach has a few tradeoffs, which may make it suitable for optimization. One prominent reason is to achieve the most feasible solution immediately. In the activity selection problem (Explained below), if more activities can be done before finishing the current activity, these activities can be performed within the same time. Another reason is to divide a problem recursively based on a condition, with no need to combine all the solutions. In the activity selection problem, the “recursive division” step is achieved by scanning a list of items only once and considering certain activities.

Greedy choice property: This property says that the globally optimal solution can be obtained by making a locally optimal solution (Greedy). The choice made by a Greedy algorithm may depend on earlier choices but not on the future. It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.

Optimal substructure: A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems. That means we can solve subproblems and build up the solutions to solve larger problems.

Characteristics of Greedy approach

- There is an ordered list of resources(profit, cost, value, etc.)
- Maximum of all the resources(max profit, max value, etc.) are taken.
- For example, in fractional knapsack problem, the maximum value/weight is taken first according to available capacity.

Applications of Greedy Algorithms

- Finding an optimal solution (Activity selection, Fractional Knapsack, Job Sequencing, Huffman Coding).
- Finding close to the optimal solution for NP-Hard problems like TSP.

Advantages and Disadvantages of Greedy Approach

Advantages

- Greedy approach is easy to implement.
- Typically have less time complexities.
- Greedy algorithms can be used for optimization purposes or finding close to optimization in case of NP Hard problems.

Disadvantages

- The local optimal solution may not always be global optimal.

Elements of the greedy strategy

Elements of the greedy strategy: A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seems best at the moment is chosen. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in activity-selection problem to develop a greedy algorithm was a bit more involved than is typical. We went through the following steps:

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
- Show that all but one of the subproblems induced by having made the greedy choice are empty.
- Develop a recursive algorithm that implements the greedy strategy.
- Convert the recursive algorithm to an iterative algorithm.

In going through these steps, we saw in great detail the dynamic-programming underpinnings of a greedy algorithm. In practice, however, we usually streamline the above steps when designing a greedy algorithm. We develop our substructure with an eye toward making a greedy choice that leaves just one subproblem to solve optimally. For example, in the activity-selection problem, we first defined the subproblems Sij, where both i and j varied. We then found that if we always made the greedy choice, we could restrict the subproblems to be of the form Si.n 1.

Alternatively, we could have fashioned our optimal substructure with a greedy choice in mind. That is, we could have dropped the second subscript and defined subproblems of the form Si = {ak ∈ S : fi ≤ sk}. Then, we could have proven that a greedy choice (the first activity am to finish in Si), combined with an optimal solution to the remaining set Sm of compatible activities, yields an optimal solution to Si. More generally, we design greedy algorithms according to the following sequence of steps:

Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.

Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

We shall use this more direct process in later sections of this chapter. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

How can one tell if a greedy algorithm will solve a particular optimization problem? There is no way in general, but the greedy-choice property and optimal sub-structure are the two key ingredients. If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the greedy-choice property: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, we must prove that a greedy choice at each step yields a globally optimal solution, and this is where cleverness may be required. It then shows that the solution can be modified to use the greedy choice, resulting in one similar but smaller subproblem.

The greedy-choice property often gains us some efficiency in making our choice in a subproblem. For example, in the activity-selection problem, assuming that we had already sorted the activities in monotonically increasing order of finish times, we needed to examine each activity just once. It is frequently the case that by preprocessing the input or by using an appropriate data structure (often a priority queue), we can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. As an example of optimal substructure, recall how we demonstrated in activity-selection problemthat if an optimal solution to subproblem Sij includes an activity ak, then it must also contain optimal solutions to the subproblems Sik and Skj. Given this optimal substructure, we argued that if we knew which activity to use as ak, we could construct an optimal solution to Sij by selecting ak along with all activities in optimal solutions to the subproblems Sik and Skj. Based on this observation of optimal substructure, we were able to devise the recurrence (16.3) that described the value of an optimal solution.

We usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, we have the luxury of assuming that we arrived at a subproblem by having made the greedy choice in the original problem. All we really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Activity Selection Problem

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

What is Activity Selection Problem?

Let's consider that you have n activities with their start and finish times, the objective is to find solution set having maximum number of non-conflicting activities that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some points to note here:

It might not be possible to complete all the activities, since their timings can collapse.

Two activities, say i and j, are said to be non-conflicting if si >= fj or sj >= fi where si and sj denote the starting time of activities i and j respectively, and fi and fj refer to the finishing time of the activities i and j respectively.

Greedy approach can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

act[] array containing all the activities.

s[] array containing the starting time of all the activities.

f[] array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

sol[] array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array act[] and add it to sol[] array.

Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.

Step 5: Select the next activity in act[] array.

Step 6: Print the sol[] array.

Algorithm

The algorithm of Activity Selection is as follows:

```
Activity-Selection(Activity, start, finish)
    Sort Activity by finish times stored in finish
    Selected = {Activity[1]}
    n = Activity.length
    j = 1
    for i = 2 to n:
        if start[i] ≥ finish[j]:
            Selected = Selected U {Activity[i]}
            j = i
 return Selected
```

Complexity

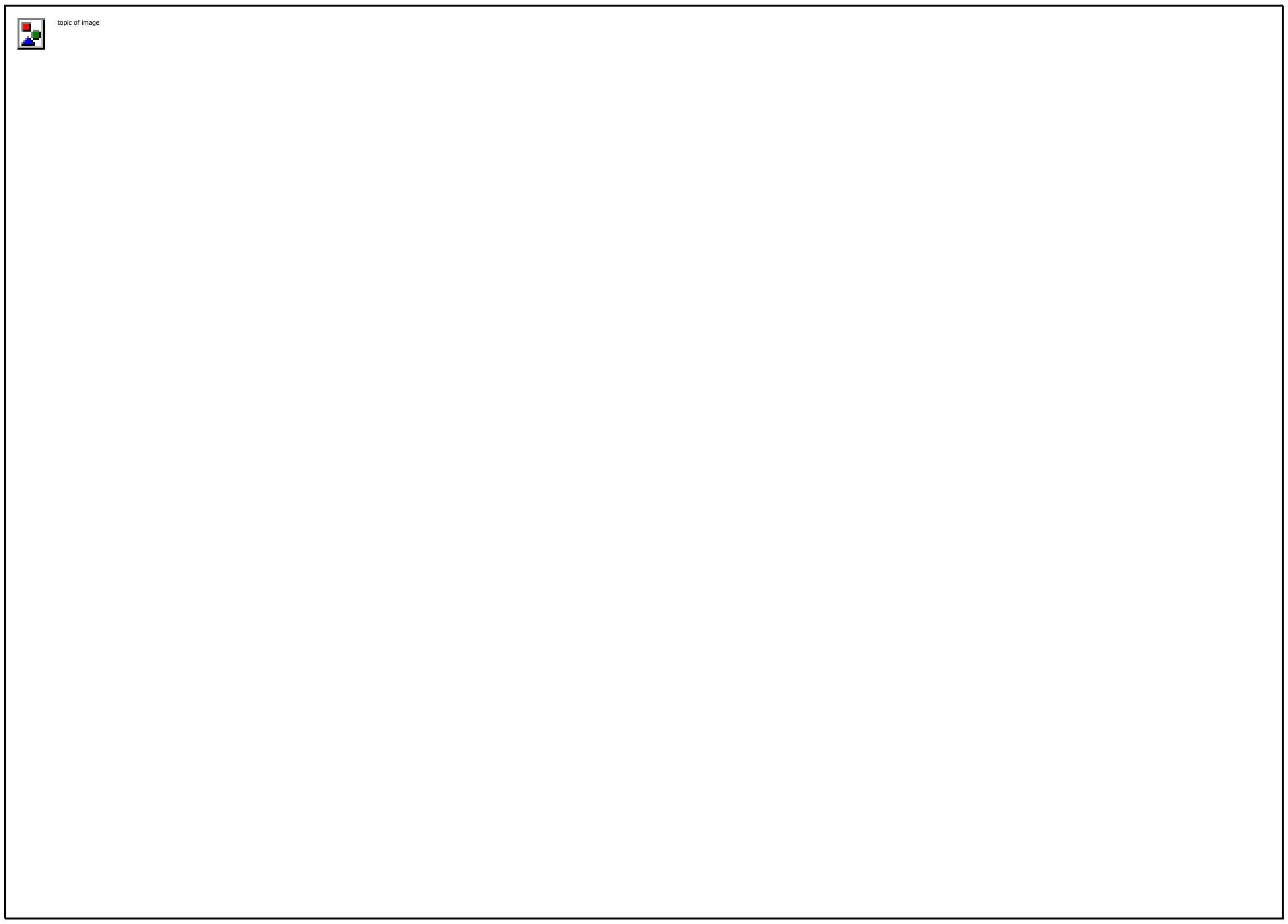
Time Complexity:

When activities are sorted by their finish time: O(N)

When activities are not sorted by their finish time, the time complexity is O(N log N) due to complexity of sorting

Example

topic of image



In this example, we take the start and finish time of activities as follows:

start = [1, 3, 2, 0, 5, 8, 11]

finish = [3, 4, 5, 7, 9, 10, 12]

Sorted by their finish time, the activity 0 gets selected. As the activity 1 has starting time which is equal to the finish time of activity 0, it gets selected. Activities 2 and 3 have smaller starting time than finish time of activity 1, so they get rejected. Based on similar comparisons, activities 4 and 6 also get selected, whereas activity 5 gets rejected. In this example, in all the activities 0, 1, 4 and 6 get selected, while others get rejected.

Applications

- Scheduling events in a room having multiple competing events
- Scheduling and manufacturing multiple products having their time of production on the same machine
- Scheduling meetings in one room
- Several use cases in Operations Research

Huffman Coding Algorithm

Every information in computer science is encoded as strings of 1s and 0s. The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous. This tutorial discusses about fixed-length and variable-length encoding along with Huffman Encoding which is the basis for all data encoding schemes

Encoding, in computers, can be defined as the process of transmitting or storing sequence of characters efficiently. Fixed-length and variable lengthare two types of encoding schemes, explained as follows-

Fixed-Length encoding

Every character is assigned a binary code using same number of bits. Thus, a string like "aabacdad" can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

Variable- Length encoding

As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text. Thus, for a given string like "aabacdad", frequency of characters 'a', 'b', 'c' and 'd' is 4,1,1 and 2 respectively. Since 'a' occurs more frequently than 'b', 'c' and 'd', it uses least number of bits, followed by 'd', 'b' and 'c'. Suppose we randomly assign binary codes to each character as follows-

a 0 b 011 c 111 d 11

Thus, the string "aabacdad" gets encoded to **00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),** using fewer number of bits compared to fixed-length encoding scheme.

But the real problem lies with the decoding phase. If we try and decode the string **00011011111011**, it will be quite ambiguous since, it can be decoded to the multiple strings, few of which are-

aaadacdad (0 | 0 | 0 | 11 | 0 | 111 | 11 | 0 | 11) aaadbcad (0 | 0 | 0 | 11 | 011 | 111 | 0 | 11) aabbcb (0 | 0 | 011 | 011 | 111 | 011)

… and so on

To prevent such ambiguities during decoding, the encoding phase should satisfy the "prefix rule" which states that no binary code should be a prefix of another code. This will produce uniquely decodable codes. The above codes for 'a', 'b', 'c' and 'd' do not follow prefix rule since the binary code for a, i.e. 0, is a prefix of binary code for b i.e 011, resulting in ambiguous decodable codes.

Lets reconsider assigning the binary codes to characters 'a', 'b', 'c' and 'd'.

a 0 b 11 c 101 d 100

Using the above codes, string "aabacdad" gets encoded to **001101011000100 (0 | 0 | 11 | 0 | 101 | 100 | 0 | 100)**. Now, we can decode it back to string "aabacdad".

Problem Statement-

Input:

Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

Output:

Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

Huffman Encoding-

Huffman Encoding can be used for finding solution to the given problem statement.

- Developed by David Huffman in 1951, this technique is the basis for all data compression and encoding schemes
- It is a famous algorithm used for lossless data encoding
- It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

The major steps involved in Huffman coding are-

Step I

Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

- A Huffman tree, similar to a binary tree data structure, needs to be created having n leaf nodes and n-1 internal nodes
- Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
- Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
- Internal nodes, on the other hand, contain weight and links to two child nodes

Step II –

- Assigning the binary codes to each symbol by traversing Huffman tree
- Generally, bit ‘0’ represents the left child and bit ‘1’ represents the right child

Algorithm for creating the Huffman Tree-

Step 1- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step 2- Repeat Steps 3 to 5 while heap has more than one node

Step 3- Extract two nodes, say x and y, with minimum frequency from the heap

Step 4- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)

Step 5- Add z to min heap

Step 6- Last node in the heap is the root of Huffman tree

Let's try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Step A- Create leaf nodes for all the characters and add them to the min heap.

Step 1- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step B- Repeat the following steps till heap has more than one nodes

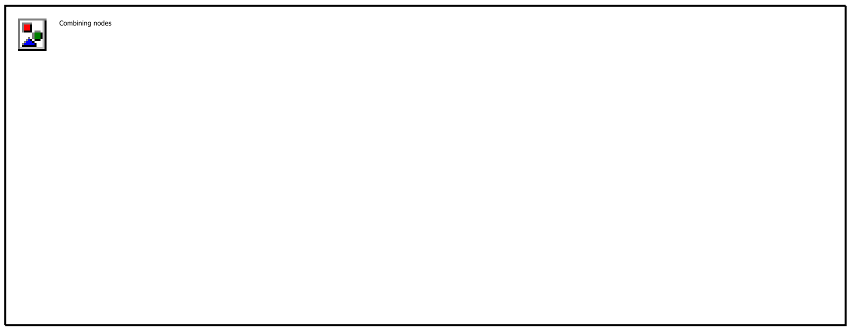
Step 3- Extract two nodes, say x and y, with minimum frequency from the heap

Step 4- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)

Step 5- Add z to min heap

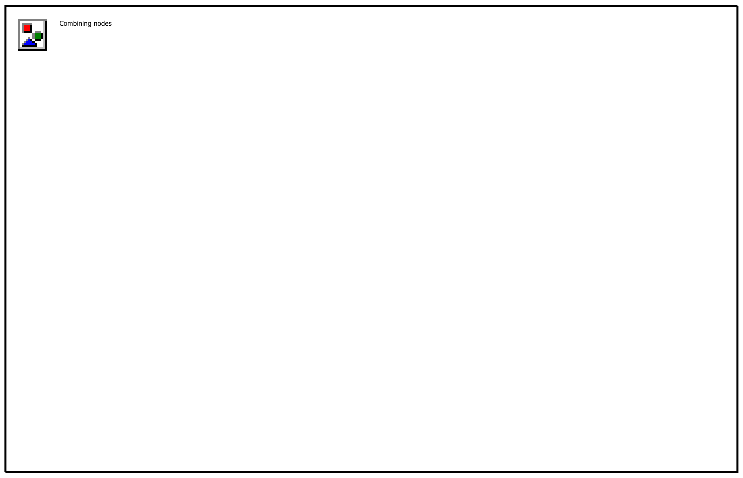
Extract and Combinenode u with an internal node having 4 as the frequency

Add the new internal node to priority queue-



Extract and Combine node awith an internal node having 8 as the frequency

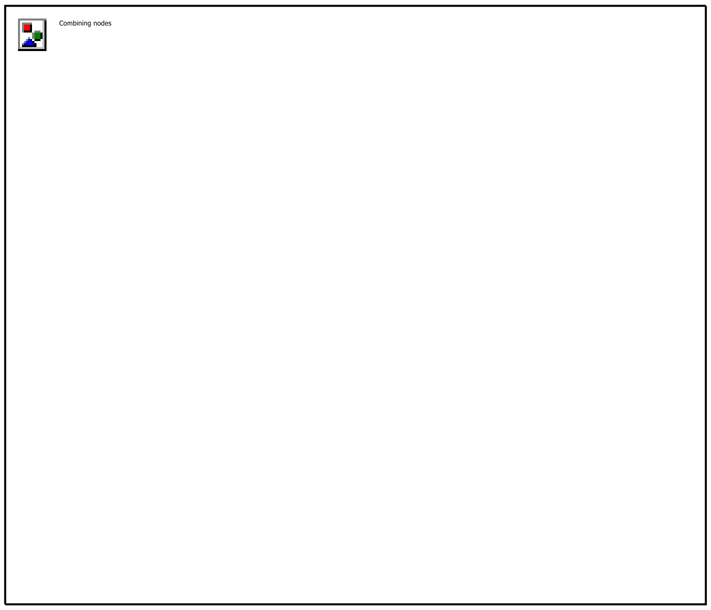
Add the new internal node to priority queue



internal node having 4 as frequency

Extract and Combine nodes i and s

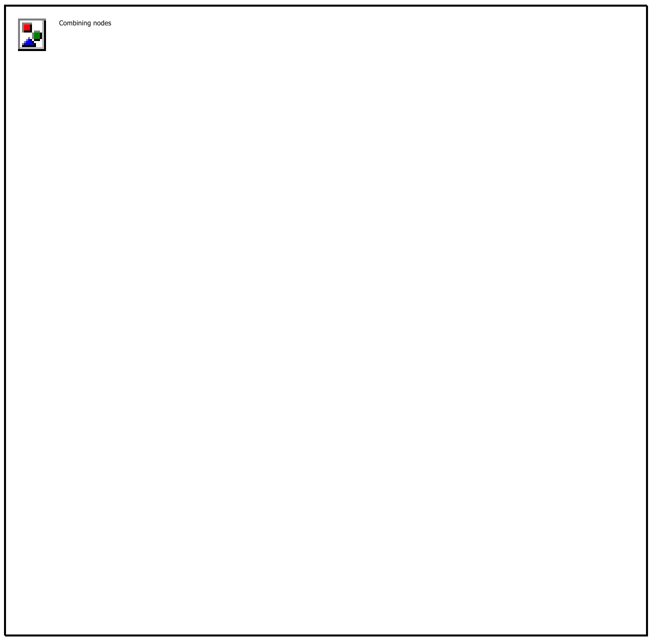
Add the new internal node to priority queue-



node having 4 as frequency

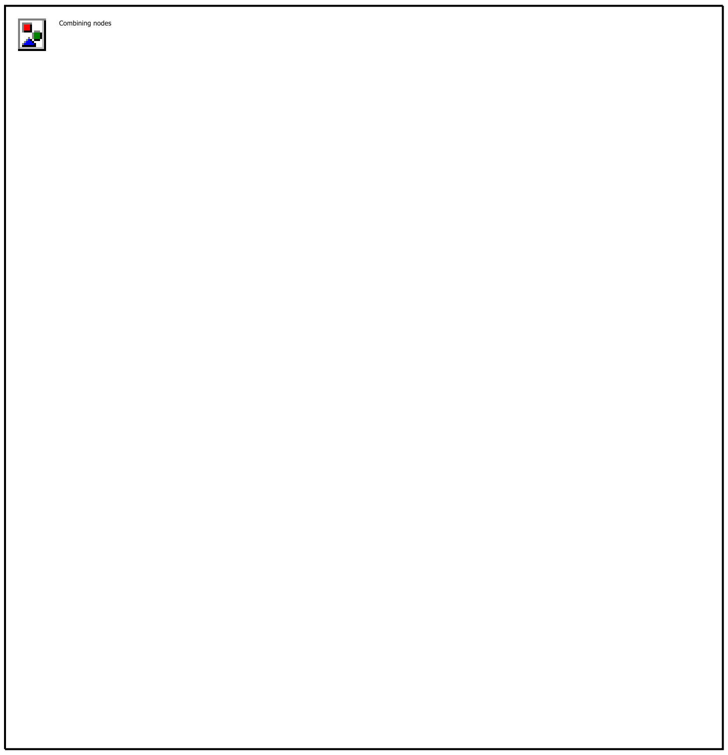
Extract and Combine nodes i and s

Add the new internal node to priority queue-



Extract and Combine node ewith an internal node having 18 as the frequency

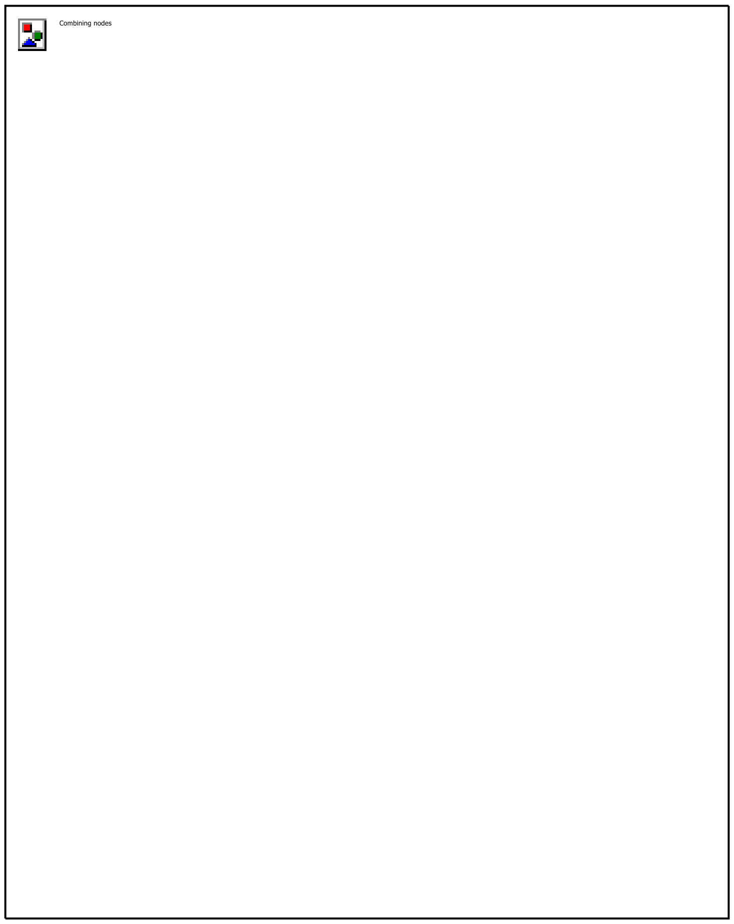
Add the new internal node to priority queue-



internal node having 18 as frequency

Finally, Extract and Combine internal nodes having 25 and 33 as the frequency

Add the new internal node to priority queue-



combining internal nodes having 25 and 33 as frequency

Now, since we have only one node in the queue, the control will exit out of the loop

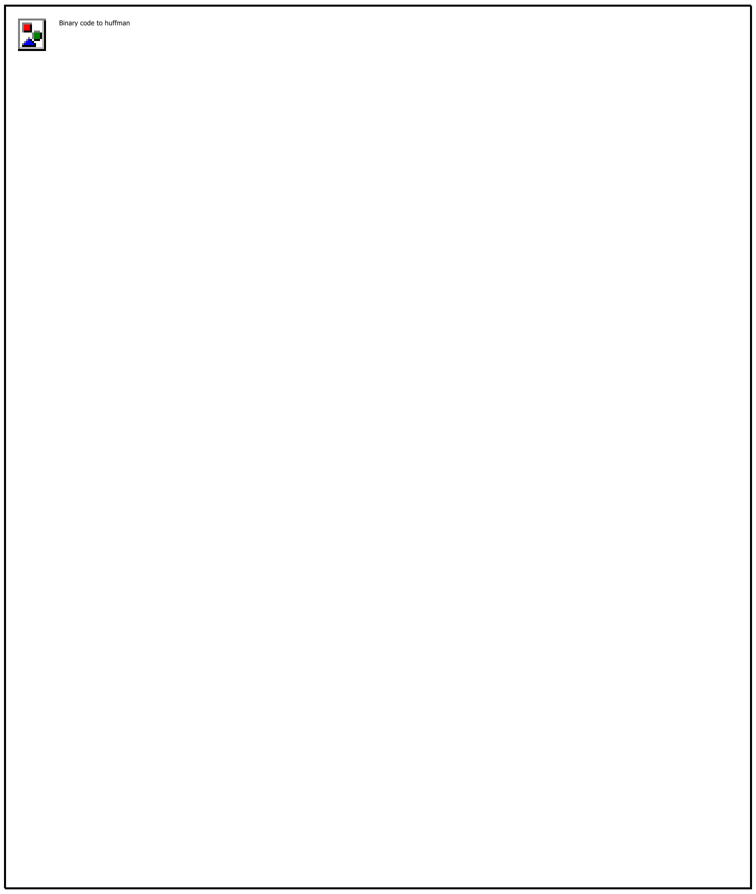
Step C- Since internal node with frequency 58 is the only node in the queue, it becomes the root of Huffman tree.

Step 6- Last node in the heap is the root of Huffman tree

Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to arraywhile traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length-



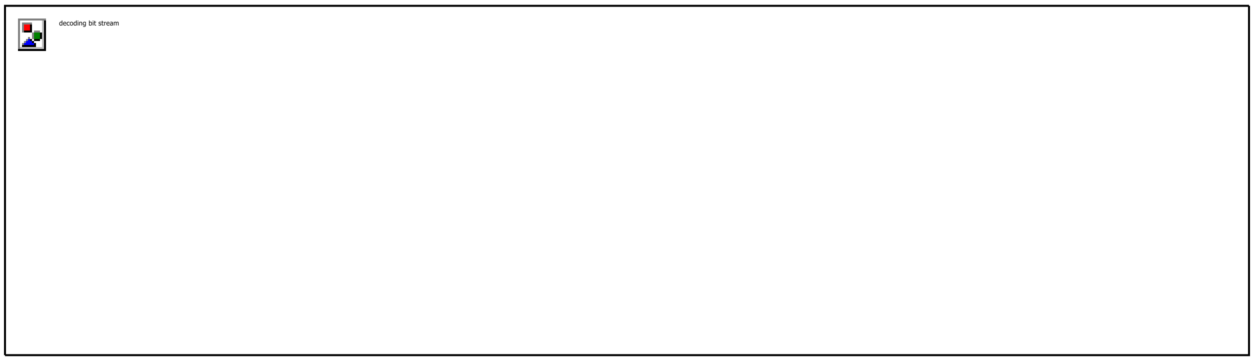
Characters	**Binary Codes**
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

Using the above binary codes-

Suppose the string “staeiout” needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to “0111010111100011011110011010” (01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010) at the sender side.

Once received at the receiver’s side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the string. A ‘1’ or ‘0’ in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

Thus for the above bit stream



Decoding the bit stream

On similar lines-

111 gets decoded to ‘a’

10 gets decoded to ‘e’

00 gets decoded to ‘i’

11011 gets decoded to ‘o’

1100 gets decoded to ‘u’

And finally, 11010 gets decoded to ‘t’, thus returning the string “staeiout” back

Time Complexity Analysis-

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).

Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

Advantages of Huffman Encoding-

This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length

It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string

The binary codes generated are prefix-free

Disadvantages of Huffman Encoding-

Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images.

Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.

Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output.

Real-life applications of Huffman Encoding-

Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.

Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes)

Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.