Unit 2

2.1 Divide and Conquer

Divide and Conquer (D&C) is an algorithm design paradigm based on recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quick sort, merge sort), multiplying large numbers , finding the closest pair of points, and computing the discrete Fourier transform (FFTs).

Advantage of Divide and Conquer

- → Solving difficult problems
- → Algorithm efficiency
- → 3.4Memory access
- → 3.5Roundoff control

2.1.1 Sorting

Arranging in an ordered sequence is called "sorting". Sorting is a common operation in many applications, and efficient algorithms to perform it have been developed.

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order.

Sorting algorithms are usually divided into two classes:

a. Internal Sorting Algorithms (which assume that data is stored in an array in main memory)
b. External Sorting Algorithm (which assume that data is stored on disk or some other device that is best accessed sequentially)

2.1.1.1 Merge Sort

Merge sort is an efficient, general-purpose, comparison-based sorting algorithm.

To sort an array A [p , .., .., .., n]

- → Divide

Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

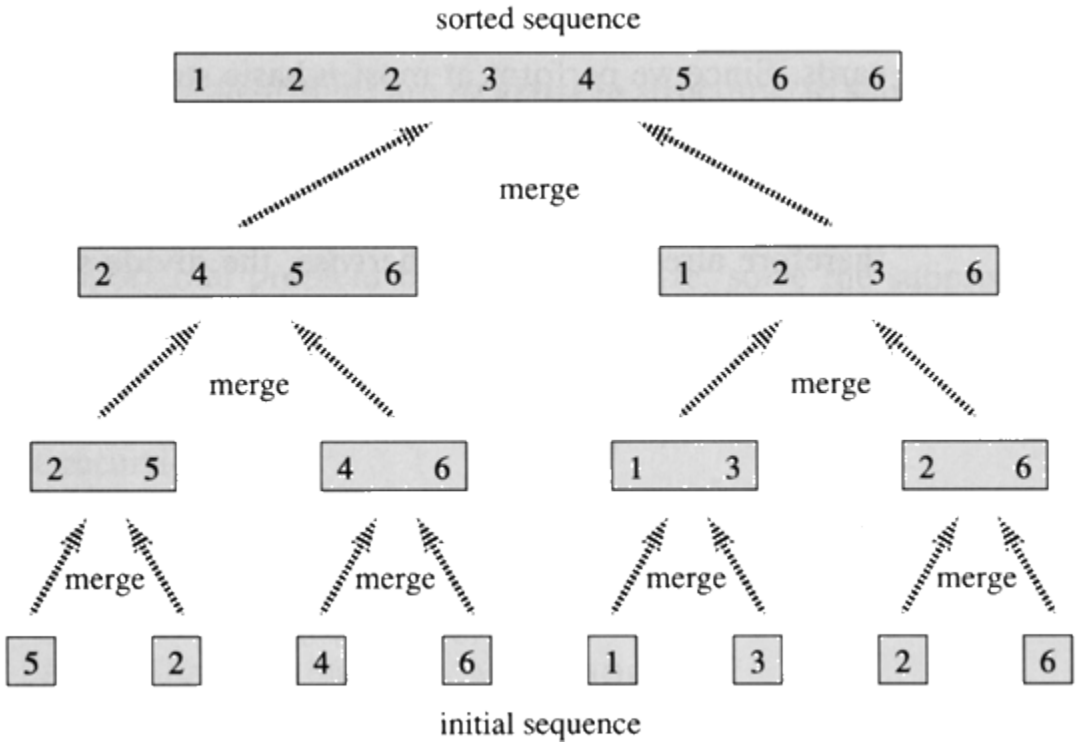
→ Conquer

Sort the subsequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do.

→ Combine

Merge the two sorted subsequences

Example:



Algorithm (Merge Sort):

Since we are dealing with sub problems, we state each sub problem as sorting a sub array $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through sub problems.

To sort the entire sequence A [1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, r)

1. IF $p < r$ // Check for base case
2. THEN q = FLOOR $[(p + r)/2]$ // Divide step
3. MERGE (A, p, q) // Conquer step.

4. MERGE (A, $q + 1$, r) // Conquer step.
5. MERGE (A, p, q, r) // Conquer step.

Time Complexity:	{
	T(n) = 1 if n=1
	T(n) = 2 T(n/2) + O(n) if n>1
	}
	= O(n log n)

Space Complexity	O(n)

Quick sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Q. How Quick sort is different from Merge Sort?

Like merge sort, quick sort uses divide-and-conquer, and so it's a recursive algorithm. The way that quick sort uses divide-and-conquer is a little different from how merge sort does. In merge sort, the divide step does hardly anything, and all the real work happens in the combine step. Quick sort is the opposite: all the real work happens in the divide step. In fact, the combine step in quick sort does absolutely nothing.

To sort an array

- Divide

  Partition the array A[l⋯r] into 2 sub arrays A[l..m] and A[m+1..r], such that each element of A[l..m] is smaller than or equal to each element in A[m+1..r]. Need to find index p to partition the array.

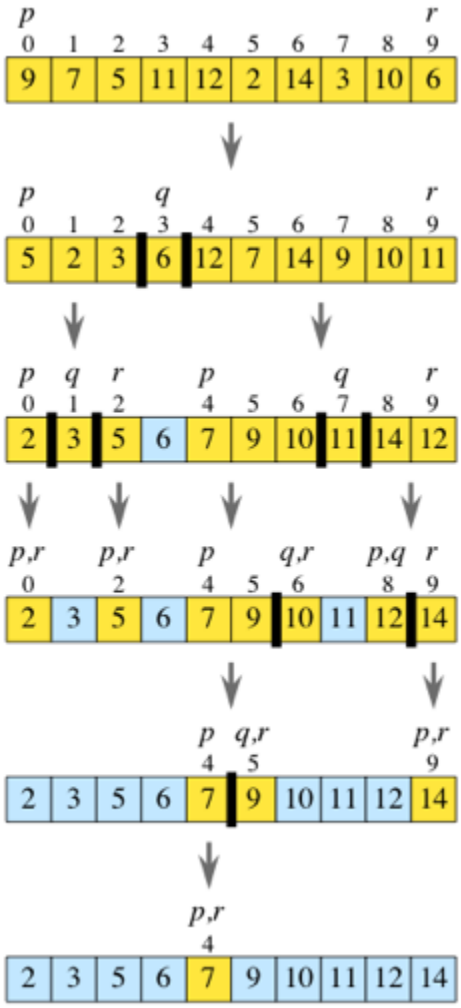
- Conquer

  Recursively sort A[p..q] and A[q+1..r] using Quick sort

- Combine

  No additional work is required to combine them.

Here is how the entire quick sort algorithm unfolds. Array locations in blue have been pivots in previous recursive calls, and so the values in these locations will not be examined or moved again:



Algorithm for Quick Sort

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

Time Complexity	Best Case {(T (n/2) + O (n) } = O (n log n)
	Worst Case { T(n) = T(n-1) + O(1)} O (n^2)
	Average Case: O(n log n)
Space Complexity	O(log n)

Class Work

Q. Sort the following array using Quick Sort.

{5 3 2 6 4 1 3 7}

Randomized Quick Sort:

The algorithm is called randomized if its behavior depends on input as well as random value generated by random number generator. Randomized algorithm is designed in such a way that no particular input can produce worst-case behavior of an algorithm.

Algorithm:

→ Pick a pivot element uniformly at random from the array

→ Split array into sub arrays: those smaller than pivot, those larger than pivot, and the pivot itself.

→ Recursively sort the sub arrays, and concatenate them.

Time Complexity	n log n (Average Case)

Assignment 2

What is Heap Sort? Write its algorithm with Example. State its time and space complexity.

Searching

A search algorithm is an algorithm that retrieves information stored within some data structure. Data structures can include linked lists, arrays, search trees, hash tables, or various other storage methods. The appropriate search algorithm often depends on the data structure being searched.

Search algorithms can be classified based on their mechanism of searching.

a. Sequential Search

Simply search for the given element left to right and return the index of the element, if found. Otherwise return Not Found".

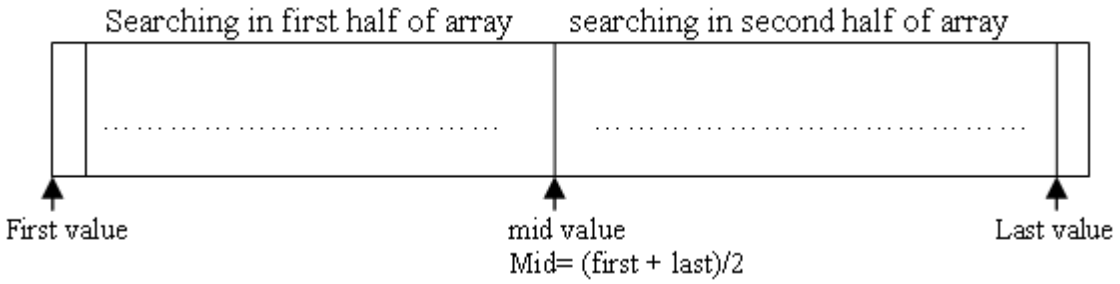
Pseudo Code:

```
for(i=0;i<n;i++)
{
        if(A[i] == key)
        return I;
}
```

Time Complexity	O(n)

b. Binary Search

Binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful or the remaining half is empty.



How Binary Search Works?

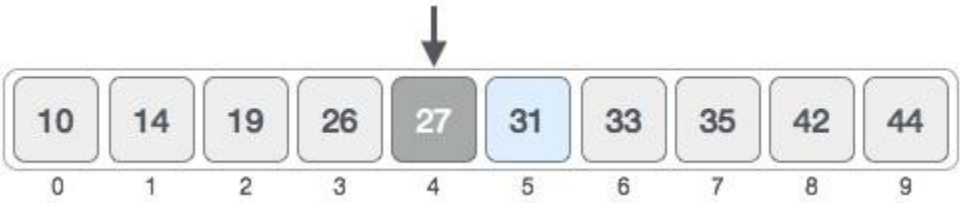
For a binary search to work, it is mandatory for the target array to be sorted.

The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

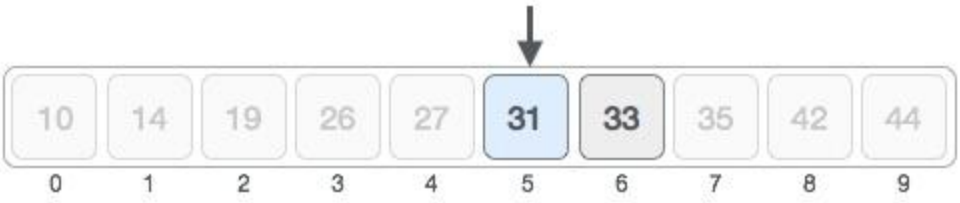
low = mid + 1

mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match; rather it is less than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Algorithm

Given an array A of n elements with values or records $A_0 \dots A_{n-1}$, sorted such that $A_0 \leq \dots \leq A_{n-1}$, and target value T, the following subroutine uses binary search to find the index of T in A.

1. Set L to 0 and R to $n - 1$.
2. If $L > R$, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor (the largest previous integer) of $(L + R)/2$.
4. If $A_m < T$, set L to $m + 1$ and go to step 2.
5. If $A_m > T$, set R to $m - 1$ and go to step 2.
6. Now $A_m = T$, the search is done; return m.

Pseudo code

```
BinarySearch(A,L,R, key)
{
  if(L= = r)
        {
              if(key = = A[L])
                    return L+1; //index starts from 0
              else
                    return 0;
        }
```

```
else
{
        m = (L + R) /2 ; //integer division
                if(key = = A[m]
                        return m+1;
                else if (key < A[m])
                        return BinarySearch(L, m-1, key) ;
                else
                        return BinarySearch(m+1, r, key) ;
        }
}
```

Time Complexity	T(n) = T(n/2) + Q(1)= O(log n)
In the best case output is obtained at one run i.e. O(1) time if the key is at middle. In the worst case the output is at the end of the array so running time is O(logn) time. In the average case also running time is O(logn). For unsuccessful search best, worst and average time complexity is O(logn).	

Median-Finding Problems and General Order Statistics

Median finding algorithms (also called linear-time selection algorithms) use a divide and conquer strategy to efficiently compute the i^{th} smallest number in a list of size n, where i is an integer between 1 and n.

i^{th} order statistic of a set of elements gives i^{th} largest(smallest) element. In general, i^{th} order statistic gives i^{th} smallest. Then minimum is first order statistic and the maximum is last order statistic. Similarly a median is given by i^{th} order statistic where i = (n+1)/2 for odd n and i = n/2 and n/2 + 1 for even n. This kind of problem commonly called selection problem.

Non linear general selection algorithm

We can construct a simple, but inefficient general algorithm for finding the k^{th} smallest or k^{th} largest item in a list, requiring O (k n) time, which is effective when k is small. To accomplish this, we simply find the most extreme value and move it to the beginning until we reach our desired index.

```
Select (A, k)
{
        for ( i=0; i<k; i++)
        {
                minindex = i;
                minvalue = A[i];
                        for (j=i+1; j<n; j++)
                        {
                                if (A[j] < minvalue)
                                {
                                        minindex = j;
                                        minvalue = A[j];
                                }
                        swap (A[i], A[minindex]);
                        }
        }
        return A[k];
}
```

Time Complexity	$O(n^2)$

Selection in expected linear time

This problem is solved by using the "divide and conquer" method. The main idea for this problem solving is to partition the element set as in Quick Sort where partition is randomized one.

```
RandSelect (A,L,R,i)
{
        if(L= =R )
        {
                return A[p];
        }
        p = RandPartition(A,L,R);
```

```
        k = p - L + 1;
        if(i <= k)
                return RandSelect(A,L,p-1,i);
        else
                return RandSelect(A,p+1,R,i - k);
}
```

Time Complexity	$O(n^2)$

Max and Min Finding

Here our problem is to find the minimum and maximum items in a set of n elements.

Divide and Conquer Algorithm for finding min-max:

Main idea behind the algorithm is: if the number of elements is 1 or 2 then max and min are obtained. Otherwise split problem into approximately equal part and solved recursively.

```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[i];
        }
        Else
        {
                Max = a[i];
                Min = a[j];
        }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
The procedure is initially invoked by the statement, MaxMin (1, n, x, y)
```

Time Complexity	recurrence relation is 0 n=1 T(n) = 1 n=2 T(n/2) + T(n/2) + 2 n>2 = O(n)

Matrix Multiplication

Given two A and B n-by-n matrices our aim is to find the product of A and B as C that is also n-by-n matrix.

$$\begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix} = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}$$

Where

C11= A11 x B11 + A12 x B21

C12= A11 x B12 + A12 x B22

C21= A21 x B11 + A22 x B21

C22= A21 x B12 + A22 x B22

We can find this by using the relation

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$$

```
MatrixMultiply(A,B)
{
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        for(k=0;k<n;k++)
                        {
                                C[i][j] = C[i][j]+ A[i][k]*B[k][j];
                        }
                }
        }
}
```

Using the above formula we need O (n) time to get C (i,j). There are n^2 elements in C hence the time required for matrix multiplication is O (n^3). We can improve the above complexity by using divide and conquer strategy.

Strassens's Matrix Multiplication

Strassen showed that 2x2 matrix multiplications can be accomplished in 7 multiplication and 18 additions or subtractions.

The basic calculation is done for matrix of size 2 x 2.

$$\begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix} = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}$$

Where,

P1 = (A11+ A22) (B11+B22)

P2 = (A21 + A22) * B11

P3 = A11 * (B12 - B22)

P4 = A22 * (B21 - B11)

P5 = (A11 + A12) * B22

P6 = (A21 - A11) * (B11 + B12)

P7 = (A12 - A22) * (B21 + B22)

C11 = P1 + P4 - P5 + P7

C12 = P3 + P5

C21 = P2 + P4

C22 = P1 + P3 - P2 + P6

We can have recurrence relation for this algorithm as

$$T(n) = 7T(n/2) + O(n^2) = O(n^{2.81})$$

2.2 Greedy Paradigm

Greedy method is the simple straight forward way of algorithm design. The general class of problems solved by greedy approach is optimization problems. In this approach the input elements are exposed to some constraints to get feasible solution and the feasible solution that meets some objective function best among all the solutions is called optimal solution.

Greedy algorithms always makes optimal choice that is local to generate globally optimal solution however, it is not guaranteed that all greedy algorithms yield optimal solution.

We generally cannot tell whether the given optimization problem is solved by using greedy method or not, but most of the problems that can be solved using greedy approach have two parts:

Greedy choice property

Globally optimal solution can be obtained by making locally optimal choice and the choice at present cannot reflect possible choices at future.

Optimal substructure

Optimal substructure is exhibited by a problem if an optimal solution to the problem contains optimal solutions to the sub problems within it.

0/1 Knapsack Problem

Statement:

A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is w_i and it worth v_i. Any amount of item can be put into the bag i.e. x_i fraction of item can be collected, where $0<=x_i<=1$. Here the objective is to collect the items that maximize the total profit earned.

We can formally state this problem as, maximize $\sum_{i=1}^{n} x_i v_i$ Using the constraints $\sum_{i=1}^{n} x_i w_i \leq W$

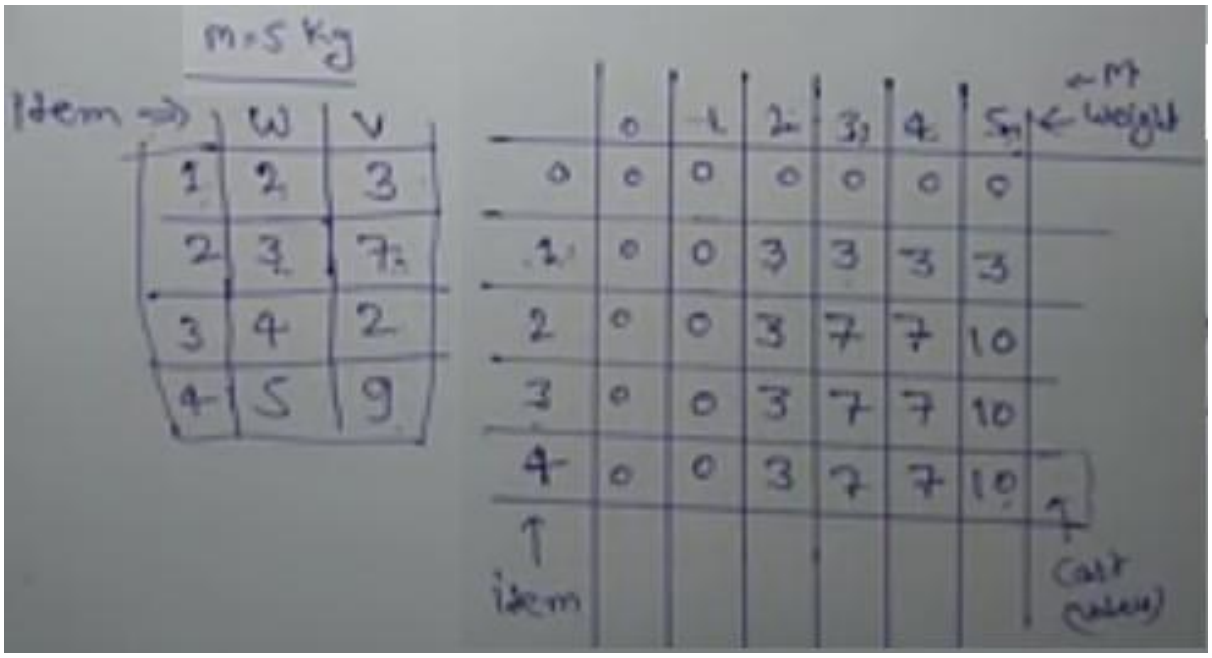
The algorithm takes as input maximum weight W, the number of items n, two arrays v[] for values of items and w[] for weight of items. Let us assume that the table c[i,w] is the value of solution for items 1 to i and maximum weight w. Then we can define recurrence relation for 0/1 knapsack problem as

$$C[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ C[i-1,w] & \text{if } wi > w \\ \text{Max}\{vi + C[i-1,w-wi], c[i-1,w] & \text{if } i>0 \text{ and } w>wi \end{cases}$$

```
DynaKnapsack(W,n,v,w)
{
        for(w=0; w<=W; w++)
                C[0,w] = 0;
        for(i=1; i<=n; i++)
                C[i,0] = 0;
        for(i=1; i<=n; i++)
        {
                for(w=1; w<=W;w++)
                {
                        if(w[i]<w)
                        {
                                if v[i] +C[i-1,w-w[i]] > C[i-1,w]
                                {
                                        C[i,w] = v[i] +C[i-1,w-w[i]];
                                }
                                else
                                {
                                        C[i,w] = C[i-1,w];
                                }
                        }
                        else
                        {
                                C[i,w] = C[i-1,w];
                        }
                }
        }
```

Analysis

For run time analysis examining the above algorithm the overall run time of the algorithm is O (nW).



Job Sequencing with Deadline

Problem - Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Example :

JobId	A	B	C	D	E
Deadline	2	1	2	1	3
Profit	100	19	27	25	15

Solution

Greedy Approach

1) Sort all jobs in decreasing order of profit.

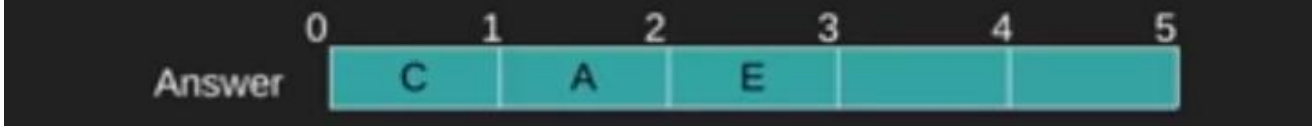
Sorted

JobId	A	C	D	B	E
Deadline	2	2	1	1	3
Profit	100	27	25	19	15

2) **Do following for remaining n-1 jobs**

a) If the current job can fit in the current result sequence without missing the deadline, add current job to the result, else ignore the current job.

JobId	A	C	D	B	E
Deadline	2	2	1	1	3
Profit	100	27	25	19	15



Analysis

- Assume the jobs are ordered such that p[1] >=p[2] >=…>=p[n]
- d[i]>=1, 1<=i<=n are the deadlines, n>=1.
- J[i] is the i^{th} job in the optimal solution, 1<=i<=k.

```
JobSequencing(int d[ ], int j[ ], int n)
{
        for(i=1;i<=n;i++)
        {
        //initially no jobs are selected
                J[i]=0;
        }
        for (int i=1; i<=n; i++)
        {
                d=d[i];
                for(k= d; k>=0; k--)
                        {
                                If (j[k]==0)
                                        {
                                                J[k] = i;
                                        }
                        }
        }
}
```

First for loop executes for O (n) times.

In case of second loop outer for loop executes O (n) times and inner for loop executes for at most O (n) times in the worst case. All other statements takes O (1) time. Hence total time for each of the iteration of the outer for loop is O (n) in worst case.

Thus time complexity= O (n) + O (n^2) = O (n^2)

Huffman Coding

Huffman coding is an algorithm for the lossless compression of files based on the frequency of occurrence of a symbol in the file that is being compressed. In any file, certain characters are used more than others. Using binary representation, the number of bits required to represent each character depends upon the number of characters that have to be represented. Using one bit we can represent two characters, i.e., 0 represents the first character and 1 represents the second character. Using two bits we can represent four characters, and so on. Unlike ASCII code, which is a fixed-length code using seven bits per character, Huffman compression is a variable-length coding system that assigns smaller codes for more frequently used characters and larger codes for less frequently used characters in order to reduce the size of files being compressed and transferred.

For example, in a file with the following data: 'XXXXXXYYYYZZ'. The frequency of "X" is 6, the frequency of "Y" is 4, and the frequency of "Z" is 2. If each character is represented using a fixed-length code of two bits, then the number of bits required to store this file would be 24, i.e., (2 x 6) + (2x 4) + (2x 2) = 24. If the above data were compressed using Huffman compression, the more frequently occurring numbers would be represented by smaller bits, such as: X by the code 0 (1 bit), Y by the code 10 (2 bits) and Z by the code 11 (2 bits), the size of the file becomes 18, i.e., (1x 6) + (2 x 4) + (2 x 2) = 18. In this example, more frequently occurring characters are assigned smaller codes, resulting in a smaller number of bits in the final compressed file.

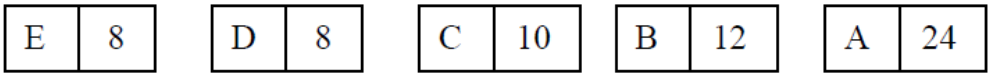
Huffman compression was named after its discoverer, David Huffman.

Example

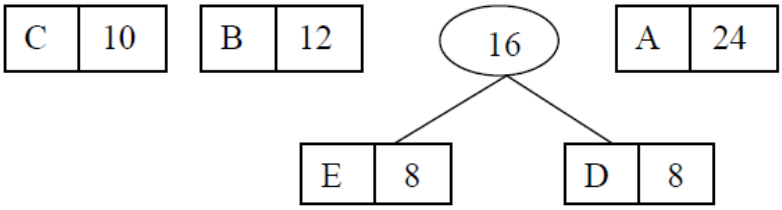
The following example bases on a data source using a set of five different symbols. The symbol's frequencies are:

Symbol	Frequency
A	24
B	12
C	10
D	8
E	8
	----> total 186 bit (with 3 bit per code word)

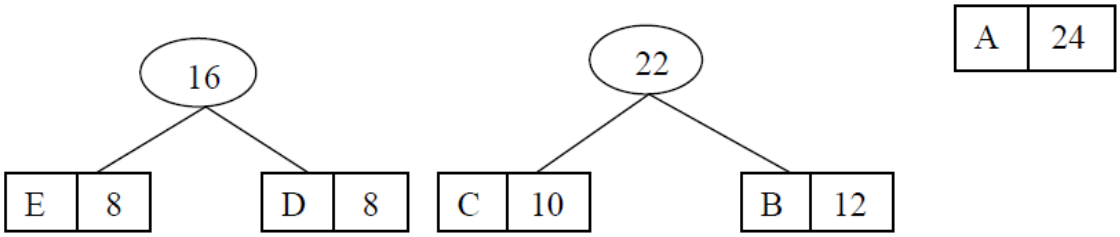
Step1:



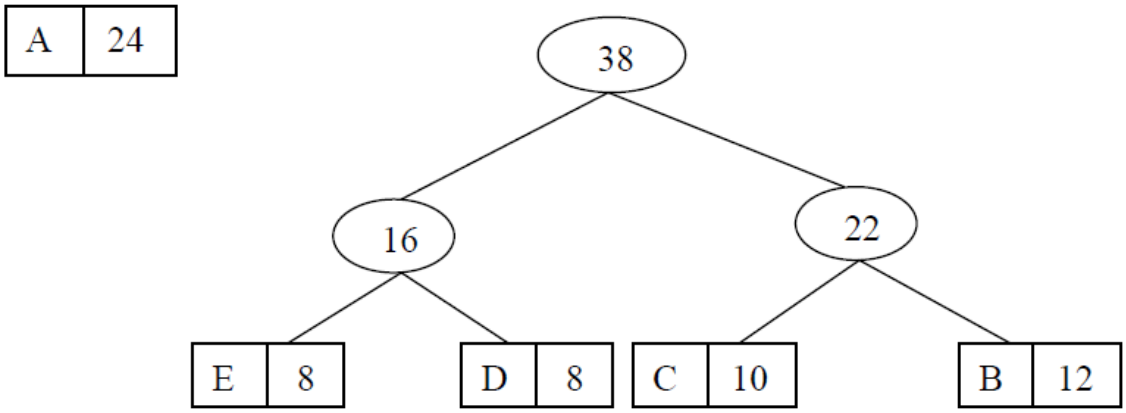
Step2:



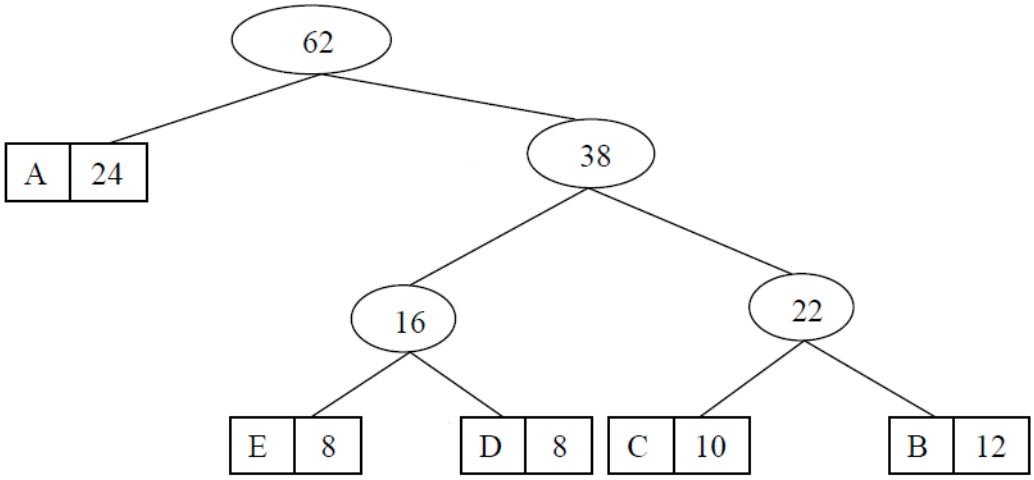
Step3:



Step4:



Step5:



Symbol	Frequency	Code	Code length	total Length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

Total length of message: 138 bit

Algorithm

A greedy algorithm can construct Huffman code that is optimal prefix codes. A tree corresponding to optimal codes is constructed in a bottom up manner starting from the |C| leaves and |C|-1 merging operations. Use priority queue Q to keep nodes ordered by frequency. Here the priority queue we considered is binary heap.

```
HuffmanAlgo(C)
{
        n = |C|;   Q = C;
        For(i=1; i<=n-1; i++)
        {
                z = Allocate-Node();
                x = Extract-Min(Q);
                y = Extract-Min(Q);
                left(z) = x;   right(z) = y;
                f(z) = f(x) + f(y);
                Insert(Q,z);
        }
```

Analysis

We can use BuildHeap(C) to create a priority queue that takes O(n) time. Inside the for loop the expensive operations can be done in O(logn) time. Since operations inside for loop executes for n-1 time total running time of Huffman algorithm is O(nlogn).

2.3 Dynamic Programming

Dynamic Programming is a most powerful technique for designing algorithm for optimization problems. Dynamic programming problems are typically optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient.

The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose your problem into smaller sub problems. Express the solution of the original problem in terms of solutions for smaller problems.

Table-structure: Store the answers to the sub-problems in a table. This is done because sub problem solutions are reused many times.

Bottom-up computation: Combine solutions on smaller sub problems to solve larger sub problems.

The most important question in designing a DP solution to a problem is how to set up the sub problem structure. This is called the **formulation of the problem.** Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to be applicable.

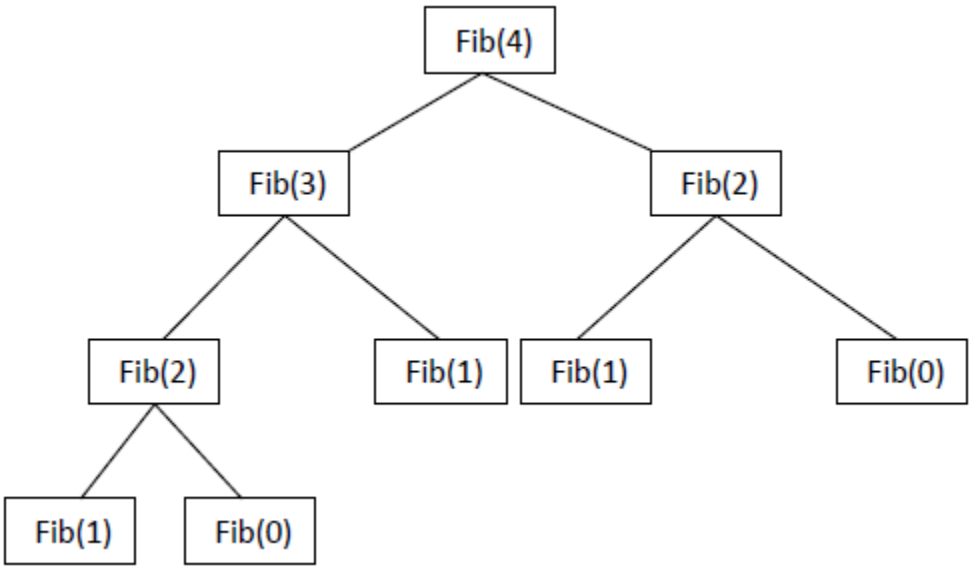
Optimal substructure: (Sometimes called the principle of optimality.) It states that for the global problem to be solved optimally, each sub problem should be solved optimally.

Polynomially many sub problems: An important aspect to the efficiency of DP is that the total number of sub problems to be solved should be at most a polynomial number.

Fibonacci numbers

In recursive version of an algorithm for finding Fibonacci number we can notice that for each calculation of the Fibonacci number of the larger number we have to calculate the Fibonacci number of the two previous numbers regardless of the computation of the Fibonacci number that has already be done. So there are many redundancies in calculating the Fibonacci number for a particular number.

Let's try to calculate the Fibonacci number of 4. The representation shown below shows the repetition in the calculation.



In the above tree we saw that calculations of fib (0) are done two times, fib (1) is done 3 times, fib (2) is done 2 times, and so on. So if we somehow eliminate those repetitions we will save the running time.

Algorithm:

```
DynaFibo(n)
{
        A[0] = 0, A[1]= 1;
        for(i = 2 ; i <=n ; i++)
        A[i] = A[i-2] +A[i-1] ;
        return A[n] ;
}
```

0/1 Knapsack Problem

……

Longest Common Subsequence Problem

Given two sequences $X = (x_1, x_2, \ldots\ldots\ldots\ldots, x_m)$ and $Z = (z_1; z_2; \ldots\ldots\ldots ; z_k)$, we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices $(i_1, i_2, \ldots\ldots\ldots i_k)$ $(1 \leq i_1 < i_2 < \ldots\ldots\ldots\ldots < i_k)$ such that $Z = (X_{i1}, X_{i2} \ldots\ldots\ldots X_{ik})$.

For example, let X = (ABRACADABRA) and let Z = (AADAA), then Z is a subsequence of X. Given two strings X and Y , the longest common subsequence of X and Y is a longest sequence Z that is a subsequence of both X and Y . For example, let X = (ABRACADABRA) and let Y = (YABBADABBAD). Then the longest common subsequence is Z = (ABADABA)

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences X = $(x1; : : : ; x_m)$ and $Y = (y, \ldots\ldots\ldots, y_n)$ determine a longest common subsequence.

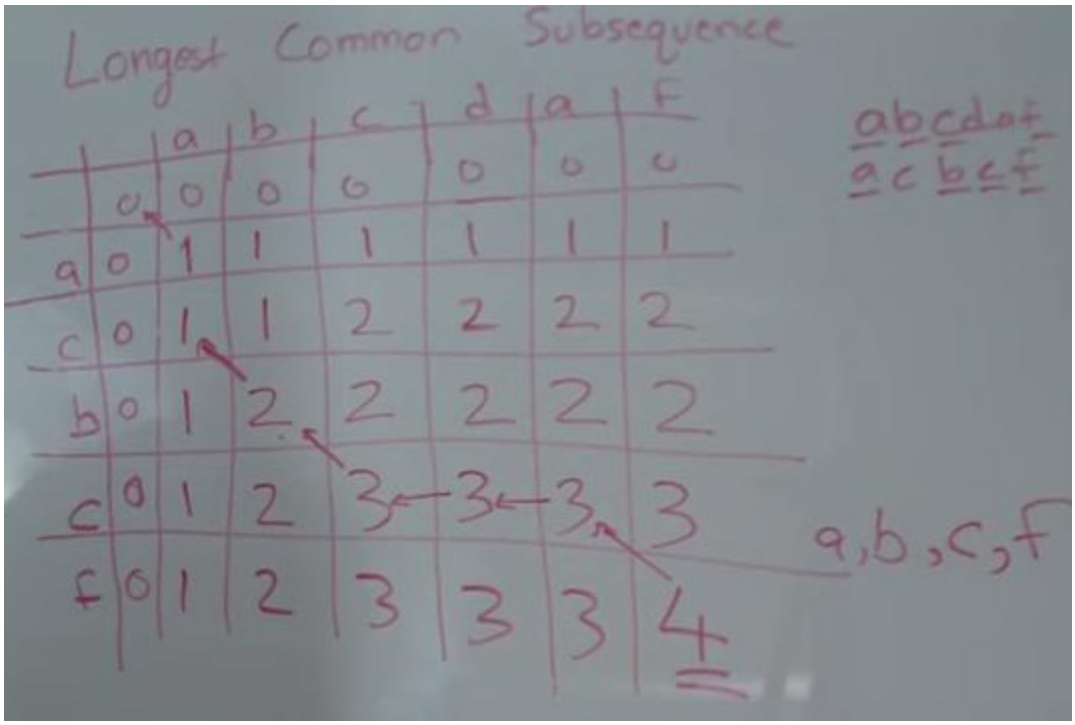
DP Formulation for LCS:

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ is called i^{th} prefix of X, here we have X_0 as empty sequence. Now in case of sequences X_i and Y_j:

If $x_i = y_j$ (i.e. last Character match) , we claim that the LCS must also contail charater x_i or y_j .

If $x_i \neq y_j$ (i.e Last Character do not match) , In this case x_i and y_j cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is not part of the LCS, or y_j is not part of the LCS (and possibly both are not part of the LCS). Let L[I,j] represents the length of LCS of sequences X_i and Y_j.

$$L[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ L[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{L[i-1,j], L[i,j-1]\} & \text{if } i>0 \text{ and } w>wi \end{cases}$$



Class work

Consider the character Sequences X=abbabba and Y=aaabba and find the LCS.

Assignment 3

Write the Pseudo code and analyze for LCS

Matrix Chain Multiplication

Chain Matrix Multiplication Problem: Given a sequence of matrices $A_1;A_2; : : : ; A_n$ and dimensions $p_0; p_1; : : : ; p_n$, where A_i is of dimension p_{i-1} x p_i, determine the order of multiplication that minimizes the number of operations.

Important Note: This algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

Although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A1 be 5 x 4, A2 be 4 x 6 and A3 be 6 x 2.

multCost[((A1A2)A3)] = (5 . 4 . 6) + (5 . 6 . 2) = 180

multCost[(A1(A2A3))] = (4 . 6 . 2) + (5 . 4 . 2) = 88

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

Let $A_{i...j}$ denote the result of multiplying matrices i through j. It is easy to see that $A_{i...j}$ is a p_{i-1} x p_j matrix. So for some k total cost is sum of cost of computing $A_{i...k}$, cost of computing $A_{k+1...j}$, and cost of multiplying $A_{i...k}$ and $A_{k+1...j}$.

Recursive definition of optimal solution: let m[j,j] denotes minimum number of scalar multiplications needed to compute $A_{i...j}$.

$$C[i,w] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j} \{m[I,k] + m[k+1,j] + p_{i-1}p_k p_j & \text{if } i<j \end{cases}$$

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one extra row and one
       extra column are allocated in m[][].  0th row and 0th
       column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar multiplications needed
       to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where
       dimension of A[i] is p[i-1] x p[i] */

    // cost is zero when multiplying one matrix.
    for (i=1; i<n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}
```

The above algorithm can be easily analyzed for running time as O (n^3), due to three nested loops. The space complexity is O (n^3).