UNIT IV – DIVIDE & CONQUER, GREEDY AND DYNAMIC PROGRAMMING

Divide and conquer: Merge sort - Quick sort - Binary search. Greedy method: Knapsack problem- Job sequencing with deadlines - Minimum cost spanning tree – Single source shortest path. dynamic programming: All pair shortest path - Knapsack problem – Traveling salesman problem - Flow shop scheduling.

GREEDY METHOD

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property:

- At every step, we can make a choice that looks best at the moment, and we get the optimal solution to the complete problem.

Some popular Greedy Algorithms are Fractional Knapsack, Dijkstra's algorithm, Kruskal's algorithm, Huffman coding and Prim's Algorithm. The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems. For example, the Traveling Salesman Problem is an NP-Hard problem. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. However, it's important to note that not all problems are suitable for greedy algorithms. They work best when the problem exhibits the following properties:

Greedy Choice Property: The optimal solution can be constructed by making the best local choice at each step.

Optimal Substructure: The optimal solution to the problem contains the optimal solutions to its subproblems.

Characteristics of Greedy Algorithm

Here are the characteristics of a greedy algorithm:

- Greedy algorithms are simple and easy to implement.
- They are efficient in terms of time complexity, often providing quick solutions. Greedy Algorithms are typically preferred over Dynamic Programming for the problems where both are applied. For example, Jump Game problem and Single Source Shortest Path Problem (Dijkstra is preferred over Bellman Ford where we do not have negative weights)..
- These algorithms do not reconsider previous choices, as they make decisions based on current information without looking ahead.

FRACTIONAL KNAPSACK PROBLEM

Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit in the

knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

An efficient solution is to use the Greedy approach. The basic idea of the greedy approach is to calculate the ratio profit/weight for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it). This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

Algorithm:

Follow the given steps to solve the problem using the above approach:

- Calculate the ratio (profit/weight) for each item.
- Sort all the items in decreasing order of the ratio.
- Initialize res = 0, curr_cap = given_cap.
- Do the following for every item i in the sorted order:
- If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
- Else add the current item as much as we can and break out of the loop.
- Return res.

Illustration:

```
#include <stdio.h>
int n = 5;
int p[10] = {3, 3, 2, 5, 1};
int w[10] = {10, 15, 10, 12, 8};
int W = 10;
int main(){
  int cur_w;
  float tot_v;
  int i, maxi;
  int used[10];
  for (i = 0; i < n; ++i)
    used[i] = 0;
  cur_w = W;
  while (cur_w > 0) {
    maxi = -1;
    for (i = 0; i < n; ++i)
      if ((used[i] == 0) &&
          ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
```

```
            maxi = i;
        used[maxi] = 1;
        cur_w -= p[maxi];
        tot_v += w[maxi];
        if (cur_w >= 0)
            printf("Added object %d (%d, %d) completely in the bag. Space left: %d.\n",
maxi + 1, w[maxi], p[maxi], cur_w);
        else {
            printf("Added %d%% (%d, %d) of object %d in the bag.\n", (int)((1 +
(float)cur_w/p[maxi]) * 100), w[maxi], p[maxi], maxi + 1);
          tot_v -= w[maxi];
          tot_v += (1 + (float)cur_w/p[maxi]) * w[maxi];
        }
      }
      printf("Filled the bag with objects worth %.2f.\n", tot_v);
      return 0;
}
```

Example: [Follow class notes]
Time Complexity: O(N * logN)
Auxiliary Space: O(N)

JOB SEQUENCING WITH DEADLINES

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 a single unit of time, so the minimum possible deadline for any job is 1. Maximize the total profit if only one job can be scheduled at a time.

Greedily choose the jobs with maximum profit first, by sorting the jobs in decreasing order of their profit. This would help to maximize the total profit as choosing the job with maximum profit for every time slot will eventually maximize the total profit

Follow the given steps to solve the problem:

- Sort all jobs in decreasing order of profit.
- Iterate on jobs in decreasing order of profit.For each job , do the following :
- Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in this slot and mark this slot filled.
- If no such i exists, then ignore the job.

```
typedef struct Job {
   char id; // Job Id
```

```
    int dead; // Deadline of job
    int profit; // Profit if job is over before or on
                // deadline
} Job;
int compare(const void* a, const void* b)
{
    Job* temp1 = (Job*)a;
    Job* temp2 = (Job*)b;
    return (temp2->profit - temp1->profit);
}
int min(int num1, int num2)
{
    return (num1 > num2) ? num2 : num1;
}
void printJobScheduling(Job arr[], int n)
{
    qsort(arr, n, sizeof(Job), compare);
    int result[n];
    bool slot[n];
    for (int i = 0; i < n; i++)
        slot[i] = false;
    for (int i = 0; i < n; i++) {
        for (int j = min(n, arr[i].dead) - 1; j >= 0; j--)
          {
            if (slot[j] == false) {
              result[j] = i;
              slot[j] = true;
              break;
            }
        }
    }
    for (int i = 0; i < n; i++)
        if (slot[i])
            printf("%c ", arr[result[i]].id);
}
int main()
{
```

```
    Job arr[] = { { 'a', 2, 100 },
                  { 'b', 1, 19 },
                  { 'c', 2, 27 },
                  { 'd', 1, 25 },
                  { 'e', 3, 15 } };
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Following is maximum profit sequence of jobs \n");
    printJobScheduling(arr, n);
    return 0;
}
```

Example: [Follow class notes]
Time Complexity: O(N2)
Auxiliary Space: O(N)

DYNAMIC PROGRAMMING

Dynamic Programming is an algorithmic technique used in computer science and mathematics to solve complex problems by breaking them down into smaller overlapping subproblems. The core idea behind DP is to store solutions to subproblems so that each is solved only once.

To solve DP problems, we first write a recursive solution in a way that there are overlapping subproblems in the recursion tree (the recursive function is called with the same parameters multiple times).

To make sure that a recursive value is computed only once (to improve time taken by algorithm), we store results of the recursive calls.

There are two ways to store the results, one is top down (or memoization) and other is bottom up (or tabulation).

All pair shortest path

The Floyd-Warshall algorithm, named after its creators Robert Floyd and Stephen Warshall, is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is

negative). It follows a Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Floyd Warshall Algorithm:

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
- For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
    - k is not an intermediate vertex in the shortest path from i to j. We keep the value of dist[i][j] as it is.
    - k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j], if dist[i][j] > dist[i][k] + dist[k][j]

Pseudo-Code of Floyd Warshall Algorithm :

```
For k = 0 to n – 1
        For i = 0 to n – 1
                For j = 0 to n – 1
                        Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])
```

where i = source Node, j = Destination Node, k = Intermediate Node

Example: [Follow class notes**]**

Time Complexity: O(V3), where V is the number of vertices in the graph and we run three nested loops each of size V

Auxiliary Space: O(V2), to create a 2-D matrix in order to store the shortest distance for each pair of nodes.

KNAPSACK PROBLEM

Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible. The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Follow the below steps to solve the problem:

The maximum value obtained from ‘N’ items is the max of the following two values.

- Case 1 (include the Nth item): Value of the Nth item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (W-weight of the Nth item).
- Case 2 (exclude the Nth item): Maximum value obtained by N-1 items and W weight.
- If the weight of the ‘Nth‘ item is greater than ‘W’, then the Nth item cannot be included and Case 2 is the only possibility.

Illustration:

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
   if (n == 0 || W == 0)
      return 0;
   if (wt[n - 1] > W)
      return knapSack(W, wt, val, n - 1);
   else
      return max(
         val[n - 1]
            + knapSack(W - wt[n - 1], wt, val, n - 1),
         knapSack(W, wt, val, n - 1));
}
int main()
{
   int profit[] = { 60, 100, 120 };
   int weight[] = { 10, 20, 30 };
   int W = 50;
   int n = sizeof(profit) / sizeof(profit[0]);
   printf("%d", knapSack(W, weight, profit, n));
   return 0;
}
```

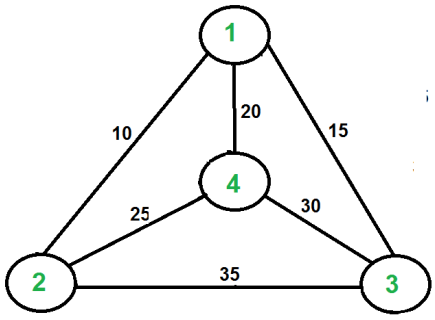
Example: [Follow class notes**]**

Time Complexity: O(N * W). As redundant calculations of states are avoided.

Auxiliary Space: O(N * W) + O(N). The use of a 2D array data structure for storing intermediate states and O(N) auxiliary stack space(ASS) has been used for recursion stack

TRAVELING SALESMAN PROBLEM

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP hard problem. There is no polynomial-time know solution for this problem. The following are different solutions for the traveling salesman problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate the cost of every permutation and keep track of the minimum cost permutation.
4) Return the permutation with minimum cost.

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as the starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values.

Time Complexity: O(n2*2n) where O(n* 2n) are the maximum number of unique subproblems/states and O(n) for transition (through for loop as in code) in every state.

Auxiliary Space: O(n*2n), where n is the number of Nodes/Cities here.

FLOW-SHOP SCHEDULING

Flow-shop scheduling is an optimization problem in computer science and operations research. It is a variant of optimal job scheduling. In a general job-scheduling problem, we are given n jobs J1, J2, ..., Jn of varying processing times, which need to be scheduled on m machines with varying processing power, while trying to minimize the makespan – the total length of the schedule (that is, when all the jobs have finished processing). In the specific variant known as flow-shop scheduling, each job contains exactly m operations. The i-th operation of the job must be executed on the i-th machine. No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified.

There are m machines and n jobs. Each job contains exactly m operations. The i-th operation of the job must be executed on the i-th machine. No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified.

Operations within one job must be performed in the specified order. The first operation gets executed on the first machine, then (as the first operation is finished) the second operation on the second machine, and so on until the m-th operation. Jobs can be executed in any order, however. Problem definition implies that this job order is exactly the same for each machine. The problem is to determine the optimal such arrangement, i.e. the one with the shortest possible total job execution makespan.

Algorithm:

```
Algorithm JOHNSON_FLOWSHOP(T, Q)
Q = Φ
for j = 1 to n do
   t = minimum machine time scanning in booth columns
   if t occurs in column 1 then
      Add Job j to the first empty slot of Q
   else
      Add Job j  to last empty slot of Q
   end
   Remove processed job from consideration
end
return Q
```

Example: [Follow class notes]

Two-machine case: The two-machine permutation flow-shop case can be solved in O(nlogn) time

---