Module -2
Comparison of Algorithmic Design techniques (Time complexity analysis)

Problem formulation and Important recurrence relations with complexity analysis

Divide and Conquer: Problem Formulation

```
Divide_Conquer(P)
{
If (P is small) return solution(P);
Else
{
k=divide(P);
Return(combine(Divide_Conquer(P1),
Divide_Conquer(P2),....
Divide_Conquer(Pk)));
```

Divide and Conquer Algorithms

Maximum and Minimum

$$T(n) = \begin{cases} \{0 | if\ n = 1\} \\ \{1 | if\ n = 2\} \\ \left\{ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 1 \middle| if\ n > 2 \right\} \end{cases}$$

Time Complexity: O(n)

Best case, worst case, and average case : O(n)

Divide and Conquer Algorithms

Binary Search

$$T(n) = \begin{matrix} 0, if\ n = 0 \\ \{1 | if\ n = 1\} \\ \left\{T\left(\frac{n}{2}\right) + c \middle| if\ n > 2\right\} \end{matrix}$$

Time Complexity: O(log n)

Best case: O(1)

Divide and Conquer Algorithms

Merge Sort

$$T(n) = \begin{cases} 0, if\ n = 0 \\ \{c | if\ n = 1\} \\ \left\{ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c \middle| if\ n > 1 \right\} \end{cases}$$

Time Complexity: O(nlog n)

Best case, worst case, average case=O(nlogn)

Divide and Conquer Algorithms

Quick Sort

$$T(n) = \begin{matrix} 0, if\ n = 0 \\ \{c | if\ n = 1\} \\ \left\{T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n \middle| if\ n > 1\right\} \end{matrix}$$

Time Complexity: O(nlog n)

Best case= O(nlog n),

Worst case is $T(n) = T(n-k) + T(k-1) + cn$ =O(n^2),

Average case=O(nlogn)

Dynamic Programming

Longest Common Subsequence

$$Len(i,j)$$

$$\{0 | if\ i = 0\ or\ j = 0\}$$

$$= \{1 + Len(i-1, j-1); | if\ x[i] = y[j]\}$$

$$\{\max[Len(i-1,j), Len(i,j-1)] | if\ x[i] \neq y[j]\}$$

Time complexity:

By brute force= $O(2^m)$

By Dynamic programming = O(mn)

Space Complexity=O(mn): where m and n are length of two given sequences.

Dynamic Programming

0/1 Knapsack

$$\mathrm{K\_S}(m,n) = \left\{ \begin{array}{c|l} \{0 | if\ m = 0\ or\ n = 0\} & \\ \{K\_S(m,n-1) | if\ w[n] > m\} & \\ \max \begin{array}{c} K_{S(m-w[n],n-1)} + P[n] \\ K\_S(m,n-1) \end{array} & otherwise \end{array} \right\}$$

Time complexity=O(MN), if m value is large then it behaves like an exponential problem and NP Complete

Dynamic Programming

Travelling Sales man Problem (TSP)

Time complexity=$O(2^m.n^2)$,

It is NP-Hard problem.

Greedy Algorithmic Design

Huffman Coding

```
Optimal merge(int A[], int n)
{
initialise a priority queue, PQ, to contain n elements in A;
struct binary tree node*temp;
for(i=0;i<n;i++)
{
temp=struct(*)malloc(size of (binary tree node));
temp→left = Delete_min(PQ);
temp→right =Delete_min(PQ);
temp→data=(temp→left→ data)+(temp→right→data);
insert temp to PQ;
}
return PQ;
}
```

Greedy Algorithmic Design

- **Huffman coding Algorithm**
  - Time complexity: O(n logn)
  - The number of bits per message
    $= \sum (frequency\ of\ external\ node\ i) \times (number\ of\ bits\ required\ for\ q)$

Greedy Algorithmic Design

- **Fractional Knapsack Problem**
  - Arrange value/weight ratio in descending order.
  - Take one at a time in descending order and place in knapsack(bag) until the bag is full.
  - Time complexity : O(nlogn)+O(n) = O(nlogn).
- **Code**

```
for(i=1;i<=capacity;i++)
{a[i]=P_i/W_j;}
```

Module – 2
Algorithmic design techniques - Comparison

- Divide and Conquer
- Backtracking
- Dynamic Programming
- Greedy
- Branch and Bound

Contrast and Similarities

- Dynamic Programming Vs Greedy
- Dynamic Programming Vs Divide and Conquer
- Backtracking Vs Branch and Bound
- Compare: D&C, DP, and Greedy

Dynamic Programming versus Greedy Method

1. Dynamic Programming is used to obtain the ***optimal solution***.
2. In Dynamic Programming, we choose at each step, but the ***choice may depend on the solution to sub-problems***.
3. ***Less efficient*** as compared to a greedy approach
4. Example: 0/1 Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using ***Principle of Optimality***.

1. Greedy Method is also used to get the ***optimal solution***.
2. In a greedy Algorithm, we make whatever choice seems ***best at the moment and then solve the sub-problems arising after the choice is made***.
3. ***More efficient*** as compared to a DP.
4. Example: Fractional Knapsack
5. In Greedy Method, there is ***no such guarantee*** of getting Optimal Solution.

Contrast and Similarities

- Greedy and Dynamic Programming are methods for ***solving optimization problems***.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for some problems for which a ***brute force approach*** would be very slow.
- To use Dynamic Programming we need only show that the ***principle of optimality*** applies to the problem.

GREEDY METHOD versus DYNAMIC PROGRAMMING

Feasibility ***:*** In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution. In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .

Optimality : In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution. It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.

Recursion ***:*** A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.

Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.

GREEDY METHOD versus DYNAMIC PROGRAMMING

Memorization: It is more efficient in terms of memory as it never look back or revise previous choices It requires dp table for memorization and it increases it's memory complexity.

Time complexity: Greedy methods are generally faster. For example, Dijkstra's shortest path algorithm takes O(ELogV + VLogV) time.

Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.

Fashion : The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.

Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.

Example : Fractional knapsack .

0/1 knapsack problem

Divide and Conquer Versus Dynamic Programming

- 1. ***Divide*** - It first divides the problem into small chunks or sub-problems.
- 2. ***Conquer*** - It then solve those sub-problems recursively so as to obtain a separate result for each sub-problem.
- 3. ***Combine*** - It then combine the results of those sub-problems to arrive at a final result of the main problem.
- Some Divide and Conquer algorithms are Merge Sort, Binary Sort, etc.

- Dynamic Programming is similar to Divide and Conquer when it comes to dividing a large problem into sub-problems. But here, ***each sub-problem is solved only once. There is no recursion. The key in dynamic programming is remembering***. That is why we store the result of sub-problems in a table so that we don't have to compute the result of a same sub-problem again and again.

- Another difference between Dynamic Programming and Divide and Conquer approach is that -
- In Divide and Conquer, the sub-problems are independent of each other while
- in case of Dynamic Programming, the ***sub-problems are not independent*** of each other
- (***Solution of one sub-problem may be required to solve another sub-problem***).

DP and D&C

Difference between dynamic programming and Divide and conquer

a. Both techniques are based on dividing a problem's instance into smaller instances of the same problem.

b. Typically, divide-and-conquer divides an instance into smaller instances with no intersection whereas

c. dynamic programming deals with problems in which smaller instances overlap.

Consequently, divide-and-conquer algorithms ***do not explicitly store solutions to smaller instances*** and dynamic programming algorithms do.

Branch-and-Bound

1. It is used to solve optimization problem.
2. It may traverse the tree in any manner, Depth first Search (DFS) or Breadth First Search (BFS).
3. It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.
4. It completely searches the state space tree to get optimal solution.
5. It involves ***bounding function***.

Backtracking Versus Branch & Bound

- Backtracking is a general algorithm for finding all ***the solutions to some computational problems***,
- notably constraint satisfaction problems, that incrementally builds possible candidates to the solutions and
- abandons a candidate as soon as it determines that the candidate cannot possibly be completed to
- finally become a valid solution. ***It is an algorithmic-technique for solving problems recursively***
- by trying to build a solution incrementally, one piece at a time, ***removing those solutions that fail to satisfy the constraints of the problem at any point of time*** (by time, here, is referred to the time elapsed till reaching any level of the search tree).

- Branch and bound is an algorithm design paradigm for ***discrete and combinatoric optimisation problems, as well as mathematical optimisation.***
- A branch-and-bound algorithm consists of a systematic ***enumeration of candidate solutions.***
- That is, the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.
- The algorithm explores branches of this tree, which represent the subsets of the solution set. Before enumerating the candidate solutions of a branch, ***the branch is checked against upper and***
- ***lower estimated bounds on the optimal solution and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.***

Branch&Bound Vs Backtracking

- **Approach :** Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.
- **Traversal:** Backtracking traverses the state space tree by DFS(Depth First Search) manner.
- **Function:** Backtracking involves feasibility function.
- **Problems:** Backtracking is used for solving Decision Problem.
- **Searching:** In backtracking, the state space tree is searched until the solution is obtained.
- **Efficiency:** Backtracking is more efficient.
- **Applications:** Useful in solving N-Queen Problem, Sum of subset.

- **Approach:** Branch-and-Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution that the pre-solution leads to,
- **Traversal:** it abandons that pre-solution. It completely searches the state space tree to get optimal solution. Branch-and-Bound traverse the tree in any manner, DFS or BFS.
- **Function:** Branch-and-Bound involves a bounding function.
- **Problems:** Branch-and-Bound is used for solving Optimisation Problem.
- **Searching:** In Branch-and-Bound as the optimum solution may be present any where in the state space tree, so the tree need to be searched completely.
- **Efficiency:** Branch-and-Bound is less efficient.
- **Applications:** Useful in solving Knapsack Problem, Travelling Salesman Problem.

The Conceptual Difference

Divide-and-Conquer Techniques

Strategy: Break a small problem into smaller sub-problems. Smaller sub-problems will most likely be a scaled down version of the original problem. Solve the smaller problems. Combine the smaller solutions to make the complete solution!

Example: Consider a cake bar of big enough size. It is broken into smaller pieces and then have one at a time.

Greedy techniques

Strategy: A solution is made up of a number of steps. At each step do what is best for you at that time. (Note: It may or may not be the best choice going forward into the game… Like a local maxima)

Example: When you are playing Cricket. At each ball you would want to Hit a six to maximise your score!

Dynamic Programming techniques

Strategy: You can't just choose a local maxima always. So you would need an exhaustive search of the solution space. Instead of recomputing each step multiple times. You would calculate smaller solutions, save them in a table and look them up when recall is computationally cheaper than re-computation.

Example: Playing Chess!.. You can't just choose the best current move. You need to think of future possibilities and scenarios.