UNIT III

CHAPTER 4

Algorithms Design Techniques & Greedy Algorithms

Syllabus

Algorithms Design Techniques : Introduction, Classification, Classification by Implementation Method, Classification by Design Method

Greedy Algorithms : Introduction, Greedy Strategy, Elements of Greedy Algorithms, Advantages and Disadvantages of Greedy Method, Greedy Applications, Understanding Greedy Technique.

Syllabus Topic : Algorithms Design Techniques - Introduction

4.1 Introduction

- In the previous chapters, we had seen many algorithms to solved different types of problem.
- Generally we need to look for the similarity of current problem to other problem for which we have solutions. This help for future reference.
- In this chapter, we will see different ways of classifying the algorithm and in subsequent chapters.
- It is covering Greedy, Divide and Conquer, Dynamic Programming.

Syllabus Topic : Classification

→ 4.1.1 Classification of Algorithms

Following are the ways of classifying algorithms.

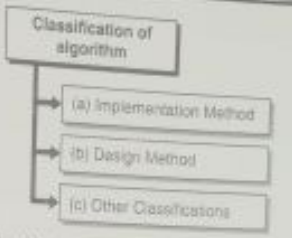

Fig. C4.1 : Classification of Algorithms

Syllabus Topic : Classification by Implementation Method

→ 4.1.1(a) Classification by Implementation Method

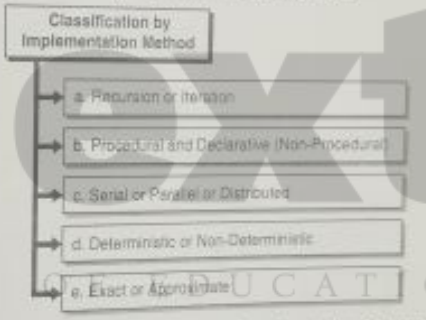

Fig. C4.2 : Classification by Implementation Method

→ a. Recursion or Iteration

- A recursion algorithm is the one that calls itself repeatedly until condition is true. It is common method used in programming language like C, C++, java etc. specifically use to calculate factorial of given number.
- Iterative algorithm use constructs like loops and sometimes data structure like Stack and Queue to solve the problem.
- Some problem like Tower of Hanoi is easily done by recursive method. Every recursive method has iterative method and vice versa.

→ b. Procedural and Declarative (Non-Procedural)

In non-procedural language like SQL, we only tell what to do but not tell how to do it. Where as in procedural language like C, PHP, Java etc. , we tell what to do and as well as how to do it.

c. Serial or Parallel or Distributed

- Computer executes instruction one after the other in serial order. These are called as serial algorithms.
- Computer executes some of instruction simultaneously using threads, this call as parallel algorithm.
- If the parallel algorithms are distributed on different machine then we call as Distributed algorithms.

d. Deterministic or Non-Deterministic

Deterministic algorithms solve the problem with a predefined process, where as non-deterministic algorithms guess the best solution at each step through the use of heuristics or random techniques.

e. Exact or Approximate

The algorithms for which we are able to find the optimal solutions are called Exact Algorithms. If we do not have optimal solution to problem, then we called it as approximate algorithms.

Syllabus Topic : Classifications by Design Method

4.1.1(b) Classifications by Design Method

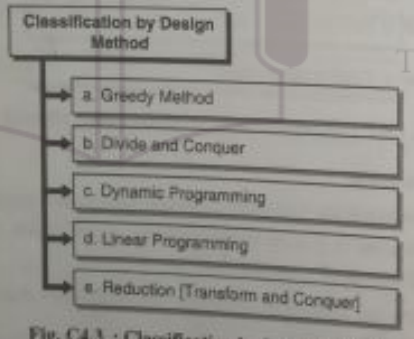

Fig. C4.3 : Classification by Design Method

a. Greedy Method

- Greedy algorithm works in stages. In each stage, a decision is made that good is good at that time. Without bothering about future consequences, this means that some 'local best' is chosen.
- It assumes that local best solution selection is best and also makes for global optimal solution.

b. Divide and Conquer

The divide and conquer method solves problem by:

1. **Divide :** Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
2. **Recursion :** Recursively solving these sub problems.
3. **Conquer :** Appropriately combining their answers.

Examples: merge sort and binary search algorithms.

c. Dynamic Programming

- Dynamic programming (DP) and memorization work together.
- The difference between DP and divide and conquer is that. In the case of the latter there is no dependency among the sub problems, where as in DP there will be an overlap of sub-problems.
- By using memorization [maintaining a table for already solved sub problems].

d. Linear Programming

- In linear programming, there are inequalities in terms of inputs and maximizing (or minimizing) some linear function of the inputs.
- Many problems (example: maximum flow for directed graphs) can be discussed using linear programming.

e. Reduction [Transform and Conquer]

- In this method we solve a difficult problem by transforming it into a known problem for which we have asymptotically optimal algorithms.
- In this method, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.
- For example, the selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list.

4.1.1(c) Other Classifications of Algorithm

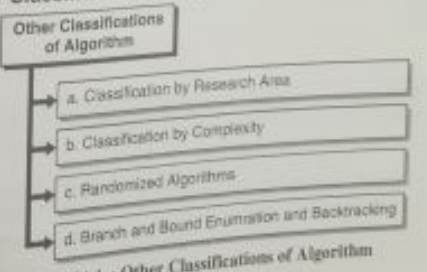

Fig. C4.4 : Other Classifications of Algorithm

➜ **a. Classification by Research Area**

- In computer science each field has its own problems and needs efficient algorithms.
- Examples: search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, geometric algorithms, combinatorial algorithms, machine learning, cryptography, parallel algorithms, data compression algorithms, parsing techniques, and more.

➜ **b. Classification by Complexity**

In this classification, algorithms are classified by the time they take to find a solution based on their input size. Some algorithms take linear time complexity ($O(n)$) and others take exponential time, and some never halt.

➜ **c. Randomized Algorithms**

A few algorithms make choices randomly. For some problems, the fastest solutions must involve randomness. Example: Quick Sort.

➜ **d. Branch and Bound Enumration and Backtracking**

These were used in Artificial Intelligence and we do not need to explore these fully. For the Backtracking method refer to the Recursion and Backtracking chapter.

Syllabus Topic : Greedy Algorithms : Introduction

4.2 Introduction of Greedy Algorithms

- A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.
- The Greedy technique is best suited for looking at the immediate situation.

4.2.1 Greedy Strategy

- A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the aim of finding a global optimum.
- In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that help to produce a global optimal solution in a reasonable time.
- For example, a greedy strategy for the travelling salesman problem (which is of a high computational complexity) is that: "At each stage visit an unvisited city nearest to the current city". This will need not find a best solution, but terminates in a reasonable number of steps; finding an optimal solution typically requires unreasonably many steps.

- That overall means Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions.

Syllabus Topic : Elements of Greedy Algorithms

4.2.2 Elements of Greedy Algorithms

- The greedy algorithm work on two properties.

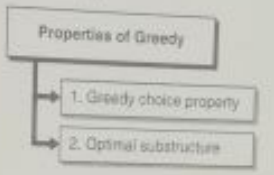

Fig. C4.5 : Properties of Greedy

➜ **1. Greedy choice property**

- We can make whatever choice seems best at the moment and then solve the subproblems that arise later. The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.
- In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution.
- After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

➜ **2. Optimal substructure**

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

Syllabus Topic : Advantages and Disadvantages of Greedy Method

4.2.3 Advantages and Disadvantages of Greedy Method

Advantages of Greedy Method

- The Greedy method is that it is straight forward, easy to understand and easy to code.

- Finding solution is quite easy with a greedy algorithm for a problem.
- Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer).
- Once we make a decision, we do not have to spend time re-examining the already computed values.

Disadvantages of Greedy Method

- The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues.
- Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science.
- In many cases there is no guaranteed that making locally optimal improvements in a locally optimal solution gives the optimal global solution.

Syllabus Topic : Greedy Applications

4.3 Greedy Applications

- **Sorting :** Selection sort, Topological sort
- **Priority Queues :** Heap sort
- Huffman coding compression algorithm
- Prim's and Kruskal's algorithms
- Shortest path in Weighted Graph {Dijkstra's}
- Coin change problem
- Fractional Knapsack problem
- Disjoint sets-UNION by size and UNION by height (or rank)
- Job scheduling algorithm
- Greedy techniques can be used as an approximation algorithm for complex problems

Syllabus Topic : Understanding Greedy Technique

4.4 Understanding Greedy Technique

To understand Greedy Algorithm lets go through with following examples.

4.4.1 Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹1, ₹ 5, ₹10 and ₹ 20 (Yes, We've ₹ 20 coins : D) and we are asked to count ₹ 36 then the greedy procedure will be –

Select one ₹ 20 coin, the remaining count is 16

- Then select one ₹ 10 coin, the remaining count is 6
- Then select one ₹ 5 coin, the remaining count is 1
- And finally, the selection of one ₹ 1 coins solves the problem

36 – 20 = 16 (20)
16 – 10 = 6 (20) (10)
6 – 5 = 1 (20) (10) (5)
1 – 1 = 16 (20) (10) (5) (1)

Fig. 4.4.1

- Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.
- For the currency system, where we have coins of ₹ 1, ₹ 6, ₹ 10 and ₹ 20 value, counting coins for value 36 will be absolutely optimum but for count like 32, it may use more coins than necessary.
- For example, the greedy approach will use 20 + 10 + 1 + 1, total 4 coins. Whereas the same problem could be solved by using only 3 coins (20 + 6 + 6)

4.4.2 Finding Largest Sum

Fig. 4.4.2

- If we've a goal of reaching the largest-sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step, and will not reach the best solution, which contains 99.

- Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

4.4.3 Huffman Coding Algorithm

- Given a set of n characters from Let alphabet 'A' each character c € A/and their associated frequency freq(c).Find a binary code for each character c € A, such that $\sum_{C \in A} freq(c)[binarycode(c)]$ is minimum, where /binarycode(c)/ represents the length of binary code of character c.
- That means the sum of the lengths of all character codes should be minimum [the sum of each character's frequency multiplied by the number of bits in the representation).
- The basic idea behind the Huffman coding algorithm is lo use fewer bits for more frequently occurring characters. The Huffman coding algorithm compresses the storage of data using variable length codes. We know that each character takes 8 bits for representation. But in general, we do not use all of them.
- Also, we use some characters more frequently than others. When reading a file, the system generally reads 8 bits at a time to read a single character. But this coding scheme is inefficient.
- The reason for this is that some characters are more frequently used than other characters. Let's say that the character 'e' is used 10 times more frequency than the character 'q'. It would then be advantageous for us to instead use a 7 bit code for e and a 9 bit code for q because that could reduce our overall message length.
- On average, using Huffman coding on standard files can reduce them anywhere from 10% to 30% depending on the character frequencies. The idea behind the character coding is to give longer binary codes for less frequent characters and groups of characters.
- Also, the character coding is constructed in such a way that no two character codes are prefixes of each other.

Example

- Let's assume that after scanning a file we find the following character frequencies.

Character	Frequency
a	12
b	2
c	7
D	13
e	14
F	85

- Given this, create a binary tree for each character that also stores the frequency with which it occurs (as shown below).

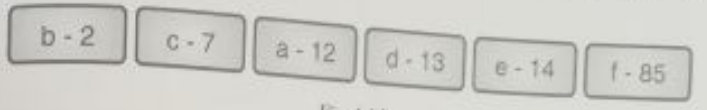

Fig. 4.4.3

- The algorithm works as follows: In the list, find the two binary trees that store minimum frequencies at their nodes.
- Connect these two nodes at a newly created common node that will store no character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like this:

Fig. 4.4.4

- Repeat this process until only one tree is left.

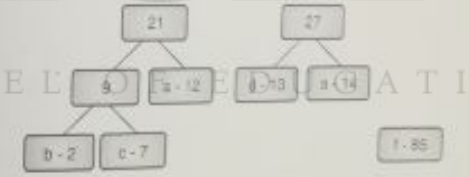

Fig. 4.4.5

Fig. 4.4.6

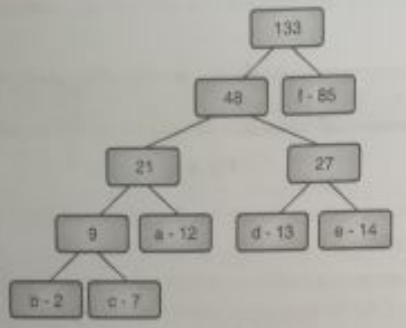

Fig. 4.4.7

- Once the tree is built, each leaf node corresponds to a letter with n code. To determine the code for a particular node, traverse from the root to the leaf node.
- For each move to the left, append a '0' to the code, and for each move to the right, append a 1. As a result, for the above generated tree, we get the following codes.

Character	Frequency
A	001
B	0000
C	0001
D	010
E	011
F	1

Calculating Bits Saved

- Now, let us see how many bits that Huffman coding algorithm is saving. All we need to do for this calculation is see how many bits are originally used to store the data and subtract from that the number of bits that are used.
- To store the data using the Huffman code.
- In the above example, since we have six characters, let's assume each character is stored with a three bit code.
- Since there are 133 such characters (multiply total frequencies by 3), the total number of bits used is 3 × 133 = 399. Using the Huffman coding frequencies we can calculate the new total number of bits used.

Letter	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

Thus, we saved 399 – 238 = 161 bits, or nearly 40% of the storage space.

```
From heapq import heappush, heappop, heapify
From collections import defaultdict
def HuffmanEncode(characterFrequency):
heap = [[freq,[sym,""]] for sym, freq in characterFrequency.items()]
heapify(heap)
while len(heap) >1:
lo = heappop(heap)
hi = heappop(heap)
for pair in lo[1:]
  pair[1] = '0' + pair[1]
for pair in hi[1:]
  pair[1] = '1' + pair[1]
heappush(heap,[lo[0]+hi[0]]+lo[1:]+hi[1:])
return sorted(heappop(heap)[1:],key=lambda p:[len(p[-1]), p])
inputText = "this is an example for Huffman encoding"
characterFrequency = defaultdict(int)
for character in inputText :
characterFrequency[character] + = 1
huffCodes = HuffmanEncode (characterFrequency)
print "Symbol \tFrequency\tHuffman code "
for p in huffCodes :
print %s\t\t\t%s\t\t\t%s"% (p[0], characterFrequency[p[0]],p[1])
```

Time Complexity: O(nlogn), since there will be one build heap, 2n - 2 delete mins, and n - 2 inserts, on a priority queue that never has more than n elements. Refer to the priority Queues chapter for details.

4.4.4 Problem

The following algorithm gives the optimal solution?

Algorithm : Merge the files in pairs. That means after the first step, the algorithm produces the n/2 intermediate files. For the next step, we need to consider these intermediate files and merge them in pairs and keep going.

Note : Sometimes this algorithm is called 2-way merging. Instead of two files at a time, if we merge K files at a time then we call is K -way merging.

- Solution: This algorithm will not produce the optimal solution and consider the previous example for a counter example. As per the above algorithm, we need to merge the first pair of files (10 and 5 size files), the second pair of files (100 and 50) and the third pair of files (20 and 15). As a result we gel the following list of files. {15, 150, 35}
- Similarly, merge the output in pairs and this step produces [below, the third element does not have a pair element, so keep it the same
- Finally, 165,35}, {185}
- The total cost of merging = Cost of all merging operations = 15 + 150 + 35 + 165 + 185 = 550. This is much more than 395 (of the previous problem). So, the given algorithm is not giving the best (optimal) solution.

Review Questions

Q. 1 Describe classifications of algorithm in detail. **(Refer sections 4.1.1, 4.1.1(a), 4.1.1(b), 4.1.1(c))**

Q. 2 Explain properties of greedy algorithms. **(Refer section 4.2.2)**

Q. 3 Write short note on advantages and disadvantages of greedy method. **(Refer section 4.2.3)**

Q. 4 Explain Huffman coding algorithm with example. **(Refer section 4.4.3)**

UNIT III

CHAPTER 5

Divide and Conquer Algorithms

Syllabus

Introduction, What is Divide and Conquer Strategy? Divide and Conquer Visualization, Understanding Divide and Conquer, Advantages of Divide and Conquer, Disadvantages of Divide and Conquer, Master Theorem, Divide and Conquer Applications

Syllabus Topic : Introduction

5.1 Introduction

In the Greedy chapter, we have seen that for many problems the Greedy strategy failed to provide optimalsolutions. From those problems, there are some that can be easily solved by using the Divide and Conquer(D & C) technique. Divide and Conquer is a popular technique for algorithm design.

Syllabus Topic : What is Divide and Conquer Strategy?

5.1.1 Divide and Conquer Strategy

- This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem.
- Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems.
- A divide-and-conquer algorithm can be described using three parts :
  1. Divide the problem into a number of subproblems that are smaller instances of the same problem.