gradeup

Design Techniques Part-1

Design Technique Part-1
Content:
1. Properties of Algorithm
2. Characteristics of Algorithm
3. Algorithm Vs. Program
4. Introduction to Algorithm Design Technique
a. Divide – and – Conquer (D and C)
b. Greedy Approach
c. Dynamic Programming
d. Branch – and – Bound
e. Randomized Algorithms.
f. Backtracking Algorithms
5. Dynamic Programming
6. Matrix Chain Multiplication

WHY STUDY ALGORITHM?
As the speed of processors increases, presentation is often said to be less central than the other software quality characteristics (e.g., security, extensibility, re-usability, etc) however, large problem sizes are common in the area of computational science; which makes performance a very important factor. This is because longer calculation time, results slower, less through research, and higher cost of computation (if buying CPU hours from an external party). The examine of algorithms, gives us a language to express performance as a function of problem size.
AN ALGORITHM MUST HAVE THE FOLLOWING PROPERTIES
- **Finiteness.** It must complete after a finite number of instructions have been executed.
- **Absence of ambiguity.** Every step must be clearly defined, having only one explanation.
- **Definition of sequence.** Every step must have a unique defined preceding and succeeding step. The first and last step (Halt step) must be clearly noted.
- **Input/Output.** Number and types of required inputs and results must be specified.
- **Feasibility.** It must be possible to perform each instruction.

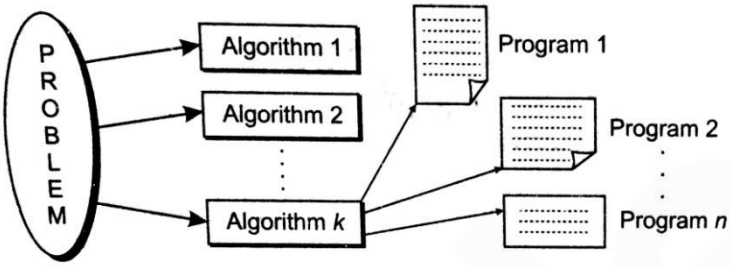
CHARACTERISTIS OF AN ALGORITHM
Every algorithm should have the following five characteristics features
1. Input
2. Output
3. Definiteness
4. Effectiveness
5. Termination

Characteristics (1) and (2) require that an algorithm produces one or more output and have zero or more input that are externally supplied.
According to characteristic (3) each operation must be definite meaning that it must be perfectly clear what should be done. Instructions such as "compute x/o" or "subtract 7 or 6 to x" are not permitted because it is not clear which of the two possibilities should be done or what the result is. That is definiteness means each instruction is clear and unambiguous. Characteristic (4) requires that each operation of effective that is each step must be such that it can be done by a person using pencil and paper in a finite amount of time. The (5) characteristic for algorithm is that it terminates after a finite number of operations. A related thought is that the time for closing should be reasonably short.

Therefore, It can be defined as a sequence of definite and productive instructions, which terminates with the production of correct output from the given input.
Showed little more correctly, It is a step-by-step formalization of a survey function to map input set onto an output set.

ALGORITHMS vs. PROGRAMS

We determine between an algorithm and a program. Program does not have to assure the finiteness condition. For example we can think of operating system that continues in a "wait" loop until more jobs are entered. Such a program does not conclude unless the system crashes. Our programs conclude when we use "algorithm" and "program" exchangeable.
Given a problem to resolve the design that produces an algorithm and the implementation phase then produces a program that expresses the designed algorithm. So, the physical expression of an algorithm in a programming language is called a program.



Example:
Problem Finding the largest number among n numbers.
Input The value of n and n numbers.
Output The largest value number.
Steps: 1. Let the value of the first be the largest value indicated by BIG
2. Let R denote the number of remaining number. R = n – 1
3. If R! = 0 then it is implied that the list is still not exhausted. Therefore look the next number
called NEW
4. Now R becomes R – 1
5. If NEW is greater than BIG then replace BIG by the value of NEW
6. Repeat steps 3 and 5 until R becomes zero (0).
7. Print BIG
8. Stop
End of Algorithm

ALGORITHM DESIGN TECHNIQUES

For a given problem, there are many ways to design algorithms for it, (e.g. Insertion sort is an incremental approach). The following is a list of famous design approaches.

- Divide – and – Conquer (D and C)
- Greedy Approach
- Dynamic Programming
- Branch – and – Bound
- Randomized Algorithms.
- Backtracking Algorithms

DIVIDE AND CONQUER

- Separate the problem into a set of sub problems.
- Solve every sub problem individually, recursively
- Combine the solutions of the sub problems (top level) into a solution of the whole original problems.

GREEDY APPROACH

They seek to improve a function by making choice (greedy criterion) which are the best locally but do not look at the global problems. The result is a better solution but not necessarily the best one. It does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

DYNAMIC PROGRAMMING

It is a technique for efficiently computing recurrences by storing partial results. It is a method of solving problems exhibiting the properties of overlapping sub-problems and optimal substructure that takes much less time then naïve methods.

BRANCH – AND – BOUND

It is given a problem, which cannot be skip, has to be divided into at least two new restricted sub problems. They are methods for global optimization in non-convex problems. They can be shown, in the worst case they need effort that grows expanding with problem size, but in some cases we are lucky, and the methods converge with must less effort.

RANDOMIZED ALGORITHMS

It is defined as an algorithm that allows to access a source freely, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

BACKTRACKING ALGORITHMS

It try every possibility until they search the right one. It is a first search of the set of attainable solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries to next alterative. When the choices are drained, the search returns to the previous choice point and try the next alternative there. It there are no choices points, the search fails.

ALGORITHM CLASSIFICATION

It use a similar problem-solving approach can be grouped together. This classification is neither exhaustive nor disjoint. Algorithm types we will consider include:

- Recursive algorithms
- Backtracking algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Branch and bound algorithms
- Randomized algorithms.

RECURSIVE ALGORITHM

- Solves the base cases directly
- Recurs with a simpler sub problem
- Does some extra work to convert the solution to the simpler sub problem into a solution to the given problem
- Example: To count the number of elements in a list

if a value occurs test a list.

BACKTRACKING ALGORITHMS

They are based on a depth-first recursive search. A backtracking algorithms:

- solution has been found if test is to be seen and returns it; otherwise
- For every choice that can be made at point,

1. Make that choice
2. Recur
3. If the recursion returns a solution, return it

- If no choice remain, return failure.

Example, To color a map with no more than four colors:

Color (Country n):

If all countries have been colored (n > number of countries) return success ; otherwise,
For each color c four colours,
If country n is not adjacent to country that has been colored c
Color country n with color c
Recursively color country n + 1
If successful, return success
If loop exists, return failure

DIVIDE AND CONQUER

It consists of two parts:

1. Divide problem into smaller sub problems of the same type, and solve these sub problems recursively.
2. Combine the solutions to the sub problems into a solution to the original problem.

An algorithm is only called "divide and conquer" if it contains at least two recursive calls. Example, Quick sort, Merge sort.

DYNAMIC PROGRAMMING ALGORITHMS

It remembers past results and uses them to find new results. It is generally used for optimization problems. In this multiple solutions exist, need to find the "best" one. It "requires substructure" and "overlapping sub problems". Optimal substructure means optimal solution. Contains solutions to sub problems. Overlapping sub problems means solutions to sub problems can be stored and reused in a bottom-up fashion.

It differs from Divide and Conquer, where sub problems need not overlap.

GREEDY ALGORITHM

An optimization problem is one in which we want to find, not just a solution, but the best solution. A "greedy algorithm" sometimes works well for optimization problems. It works in phases:

At each phase:

- We take the best we can get right not, without regard for future consequences.
- We hope that by choosing a local optimum at each step, we will and up at a global optimum

BRANCH AND BOUND ALGORITHMS

Branch and bound algorithms are generally used for optimization problem. As the algorithms progresses, a tree of sub-problems is formed. The original problem is considered the "root problems".

A process is used to build an upper and lower bound for a given problem.

- At each node, apply the bounding methods.
- If the bound match, it is deemed a feasible solution to that particular sub problem.
- If bounds do not match, partition the problem represented by that node, and make the two sub problems into children nodes.
- Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed.

BRUTE FORCEF ALGORITHMS

It simply tries all possibilities until a satisfactory solution is found. Such an algorithm can be:

- **Optimizing.** Find the best solution. This may require finding all solution, or if value for the best solution is known, it may stop when any best solution is found. Example: searching the path for a travelling salesman.
- **Satisfying.** Stop a solution is found that is good enough. Example, finding a travelling salesman path that is within 10% of optimal

RANDOMIZED ALGORITHMS

It uses a random number at least once during the computation to make a decision.

- Example, In Quick sort, using a random number of choose a pivot.
- Example, Trying to factor a large number by choosing random numbers as possible divisors.

DYNAMIC PROGRAMMING

INTRODUCTION

It is a problem solving technique that, like Divide and Conquer, solves problems by dividing them into sub problems. It is used when the sub problems are not independent, e.g. when they share the same sub problems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

It solves each sub problem once and stores the result in a table so that it can be v5555rapidly retrieved if needed again. It is often used in Optimized Problems: A problem with many possible solutions for which we want to find an optimal (the best) solution. (there may be more than 1 optimal solution). It is an approach developed to solve sequential, or multi-stage, decision problems; hence, the name "dynamic" programming. But, as we shall see, this approach is equally applicable for decision problems where sequential property is induced solely for computational convenience.

It can be through of as being the reserve of recursion. Recursion is a top-down mechanism – we take a problem, split it up, and solve the smaller problems that are created. It is a bottom-up mechanism – we solve all possible small problems and then combine them to obtain solution for bigger problems.

It works when a problem has the following characteristics:

- **Optimal Substructure:** If an optimal solution carry optimal sub solutions, then a problem exhibits optimal substructure.
- **Overlapping Sub problems:** When a recursive algorithm would visit the same sub problems repeatedly, then a problem has overlapping sub problems

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping sub problems, the new can improve on a recursive implementation by computing each sub problem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we really don't have anything to gain by using dynamic programming.

APPLICATIONS

- Knapsack problem
- Mathematical optimization problems
- Shortest path problems
- Matrix chain multiplication
- Longest common sequence (LCS)
- Control (Curise control, Robotics, Thermostates)
- Flight control (Balance factors that oppose one another, e.g., maximize accuracy, minimize time).
- Time Sharing: Schedule user and jobs to maximize CPU usage.
- Other types of scheduling.

MATRIX MULTIPLICATION

To multiple two matrices, multiply the rows in the first matrix by the column of the second matrix. In general:

If A = ⌊ ⌋ is a p × q matrix, B = ⌊ ⌋ is a q × r matrix and, C = ⌊ ⌋ is a p × r matrix

Then AB = C if $c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$

Given following matrices {A_1, A_2, A_3, …..,A_n} and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$A_1 \times A_2 \times A_3 \times \ldots \times A_n$

Matrix multiplication operation is associative is nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to parenthesize the above multiplication depending upon our need.

DYNAMIC PROGRAMMING APPROACH

THE STRUCTURE OF AN OPTIMAL PARENTHESIZATION

Let us adopt the notation $A_{i..j}$ for the matrix that results from evaluating the product $A_i\ A_{i+1} \ldots A_j$. An optimal parenthesization of the product $A_1\ A_2 \ldots A_n$. Splits the product between A_k and A_{k+1} for some integer k in the range $1 \le k \le n$ i.e. for some value of k, we first compute the matrices $A_{1..k}$ and $A_{k+1..n}$ and then multiply them together to produce the final product $A_{1..n}$. The cost of this optimal parenthesization is thus the cost of computing the matrix $A_{1..k}$ + the cost of computing $A_{k+1..n}$ + cost of multiplying them together.

Let m[I, j] be the minimized number of scalar multiplications needed to compute the matrix $A_{i..j}$, the cost of a cheapest way to compute $A_{1..n}$ would thus be m[1..n].

We can define m[i..j] recursively as follows:

If i = j the chain consists of just one matrix $A_{i..j} = A_i$ so no scalar multiplication are necessary to compute the product. Thus m[i, j] = 0 for i = 1, 2, 3, …, n.

To compute m[i, j], when i < j. Let us assume that the optimal parenthesization splits the product $A_i\ A_{i+1} \ldots A_j$ between A_k and A_{k+1} where $i \le k \le j$. then m[i, j] is equal to the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying them together. Since computing the matrix product $A_{i..k}$ and $A_{k+1..j}$ takes $p_{i-1}\ p_k\ p_j$ scalar multiplications, we obtain.

$M[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}\ p_k\ p_j$

There are only (j-1) possible values for 'k' namely k = i, i + 1, …. j-1. Since the optimal parenthesization must use one of these values for 'k', we need only check them all to find the best. So the minimum cost of parenthesizing the product $A_i\ A_{i+1} \ldots A_j$ becomes.

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

To construct an optimal solution, let us defined s[i, j] to be the value of 'k' at which we can split the product $A_i\ A_{i+1} \ldots A_j$ to obtain an optimal parenthesization i.e. s[i, j] = k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}\ p_k\ p_j$

Example: We are given the sequence [4, 10, 3, 12, 20, 7]. The matrices have sizes 4×10, 10×3, 3×12, 12×20, 20×7. We need to compute M[i, j], $0 \le i, j \le 5$. We know M[i, j] = 0 for all i.

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

We proceed, working away from the diagonal. We compute the optimal solution for products of 2 matrices.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Now products of 3 matrices.

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{cases} = \mathbf{264}$$

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases} = \mathbf{1320}$$

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases} = \mathbf{1140}$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 4 matrics.

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0 p_3 p_4 = 264 + 0 + 4 \cdot 12 \cdot 20 = 1224 \\ M[1,2] + M[3,4] + p_0 p_2 p_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 1080 \\ M[1,1] + M[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4 \cdot 10 \cdot 20 = 2120 \end{cases} = \mathbf{1080}$$

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{cases} = \mathbf{1350}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now product of 5 matrices

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{cases} = \mathbf{1344}$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

COMPARISON WITH DYNAMIC PROGRAMMING

Dynamic programming algorithm usually outperforms a top-down memorized algorithm by constant factor, because there is no over-head for recursion and fewer overheads for maintaining the table. In situations where not every subproblem is computed, memorization only solves those that are needed but dynamic programming solves all the subproblems.

In summary, the matrix-claim multiplication problem can be solved in $O(n^3)$ time by either a top-down, memorized algorithm or a bottom-up dynamic-programming algorithm.

gradeup SUPER

Gradeup UGC NET
Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now