GOVERNMENT COLLEGE OF ENGINEERING AND RESEARCH,
AVASARI (KHURD), DIST: PUNE.

Fourth Year of Computer Engineering (2019 Course)

410241: Design and Analysis of Algorithms Lab Manual

Assignment1) **Algorithms and Problem Solving**

a. **Towers of Hanoi**

Tower of Hanoi is a mathematical puzzle where we have three rods (**A**, **B**, and **C**) and **N** disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod **A**. The objective of the puzzle is to move the entire stack to another rod (here considered **C**), obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Examples:

Input*: 2*
Output: *Disk 1 moved from A to B*
Disk 2 moved from A to C
Disk 1 moved from B to C
Input: *3*
Output: *Disk 1 moved from A to C*
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C

Refer Code from: Java Code of TOH

Assignment1) **Algorithms and Problem Solving**

b. **GCD of Given Two Numbers**

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors.

Basic Euclidean Algorithm for GCD:
The algorithm is based on the below facts.

1) If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn’t change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
2) Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find the remainder 0.

Refer Java Code : GCD of Euclidian AlgorithmsGCD.java

Assignment2)**Analysis of Algorithms and Complexity Theory**

c. Analysis of iterative and recursive algorithm

When to Use Recursion?

Sometimes we encounter a problem that is too complex to solve directly. In most cases, we try to break such problems down into smaller pieces. Then, we can find a way to solve these smaller pieces and then build up a solution to the entire problem. This is the entire idea behind recursion.

Recursion is when a function calls itself directly or indirectly, and the function calling itself is called a recursive function. It is mainly used when the solution to a bigger problem can be expressed in terms of smaller problems.

To terminate the recursion, we use some conditions so that the function knows when not to make any further recursive call to itself and return; otherwise, it will lead to infinite recursion once the function is called. Such conditions are called base conditions in recursion.

For example, let's define steps for Leo in our example above to get the clue he needs:

- **Step 1:** If you have found the clue, stop; else, go to step 2.
- **Step 2:** Find a target to enter their dream for finding clues. Go to step 3.
- **Step 3:** Use the machine to enter their dream. Go to step 1.

We can see the recursive solution for finding the clue can be written in three steps. The base case here is if we have already found the clue, then we simply stop the recursion. Else, we enter some other target's dream and redo the entire algorithm.

Wondering how many times the function can call itself?

The local variables and parameters being passed by value are newly created each time the function calls itself. They are stored in stack memory whenever a recursive call is made and are deallocated when the execution of the current function call is complete. The state of the functions is also stored in stack memory. Thus, each recursive call consumes some amount of memory on the stack. In case of infinite recursion or when recursion depth is large enough to exhaust the memory on the stack, it will cause a stack overflow error.

When to Use Iteration?

Iteration is when we execute a set of instructions repeatedly until the condition controlling the loop becomes false. It includes initialization, comparison, executing statements inside the iteration, and updating the control variable.

In iteration, it is necessary to have the right controlling condition; else, the program may go in an infinite loop.

For example, let’s write an iterative program to help Leo find his clue:

- **Step 1:** Create an initial list of people who are targets.
- **Step 2:** While the list is not empty, get a target from the list and enter his dream. Go to step 3.
- **Step 3:** If you do not get the clue, go to step 2; else, go to step 4.
- **Step 4:** Congratulations! You found the clue!

Here, the while statement in Step 2 is the controlling statement, which will decide when the loop will end.

Key Differences Between Recursion and Iteration

Recursion and iteration are both different ways to execute a set of instructions repeatedly. The main difference between these two is that in recursion, we use function calls to execute the statements repeatedly inside the function body, while in iteration, we use loops like “for” and “while” to do the same.

Iteration is faster and more space-efficient than recursion. So why do we even need recursion? The reason is simple — it's easier to code a recursive approach for a given problem. Try doing inorder tree traversal using recursion and iteration both.

Example

Let’s look at a simple factorial program implemented using recursion and iteration.

Recursion

For Factorial we can write it as the below formula :

Factorial (n) = n * Factorial (n-1) , for all n>=1 ,

Factorial (0) = 1   , Base Case

So we can write this in a recursive way where if we reach n = 0, then that’s our base case; else, we make the recursive call for n-1.

Explanation:

- Here, the fact function uses recursion to calculate the factorial of a given number.
- We can write factorial(n) as n*factorial(n-1), which is the required recursive relation.

- n=0 is the base case, and we simply return 1 if it’s true.

  Factorial using Recursion: Refer Code Fact

Iteration

Iteration can be simply written by using the formula for factorial:

*Factorial (n) = 1 * 2 * 3 * ... * (n-2) * (n-1) * n*

Explanation:

- Here, we maintain a variable *answer* in which we will store the final result.
- We initialize the control variable *i = 2* and then multiply *answer* by it.
- After each iteration, *i* is incremented by 1 until it becomes greater than *n*.

Time Complexity

- There are O(N) recursive calls in our recursive approach, and each call uses O(1) operations. Thus, the time complexity of factorial using recursion is **O(N).**
- There are O(N) iterations of the loop in our iterative approach, so its time complexity is also O(N).
- Though both the programs’ theoretical time complexity is the same, a recursive program will take more time to execute due to the overhead of function calls, which is much higher than that of iteration.

Space Complexity

- In the recursive program, due to each recursive call, some memory gets allocated in the stack to store parameters and local variables. As there are O(N) recursive calls, the space complexity using recursion is **O(N).**
- No extra memory gets allocated in the iterative program, so its space complexity is O(1).

Strengths and Weaknesses of Recursion and Iteration

Iteration

Strengths:

- Iteration can be used to repeatedly execute a set of statements without the overhead of function calls and without using stack memory.
- Iteration is faster and more efficient than recursion.
- It's easier to optimize iterative codes, and they generally have polynomial time complexity.
- They are used to iterate over the elements present in data structures like an array, set, map, etc.
- If the iteration count is known, we can use *for* loops; else, we can use *while* loops, which terminate when the controlling condition becomes false.

Weaknesses:

- In loops, we can go only in one direction, i.e., we can’t go or transfer data from the current state to the previous state that has already been executed.
- It’s difficult to traverse trees/graphs using loops.
- Only limited information can be passed from one iteration to another, while in recursion, we can pass as many parameters as we need.

Recursion

Strengths:

- It’s easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - factorial(n) = n * factorial(n-1)

- Recursive codes are smaller and easier to understand.
- We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.
- It’s a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

- The simplicity of recursion comes at the cost of time and space efficiency.
- It is much slower than iteration due to the overhead of function calls and control shift from one function to another.
- It requires extra memory on the stack for each recursive call. This memory gets DE allocated when function execution is over.
- It is difficult to optimize a recursive code, and they generally have higher time complexity than iterative codes due to overlapping sub problems.

Assignment No3)**Greedy And Dynamic Programming algorithmic Striate**

d. Job Scheduling using Greedy Algorithm

Job scheduling is the problem of scheduling jobs out of a set of N jobs on a single processor which maximizes profit as much as possible. Consider N jobs, each taking unit time for execution. Each job is having some profit and deadline associated with it. Profit earned only if the job is completed on or before its deadline. Otherwise, we have to pay a profit as a penalty. Each job has deadline $d_i \geq 1$ and profit $p_i \geq 0$. At a time, only one job can be active on the processor.

The job is feasible only if it can be finished on or before its deadline. A feasible solution is a subset of N jobs such that each job can be completed on or before its deadline. An optimal solution is a solution with maximum profit. The simple and inefficient solution is to generate all subsets of the given set of jobs and find the feasible set that maximizes the profit. For N jobs, there exist 2^N schedules, so this brute force approach runs in $O(2^N)$ time.

However, the greedy approach produces an optimal result in fairly less time. As each job takes the same amount of time, we can think of the schedule S consisting of a sequence of job slots 1, 2, 3, …, N, where S(t) indicates job scheduled in slot t. Slot t has a span of (t – 1) to t. S(t) = 0 implies no job is scheduled in slot t.

Schedule S is an array of slots S(t), $S(t) \in \{1, 2, 3, \ldots, N\}$ for each $t \in \{1, 2, 3, \ldots, N\}$

Schedule S is *feasible* if,

- $S(t) = i$, then $t \leq d_i$ (Scheduled job must meet its deadline)
- Each job can be scheduled at max once.

Our goal is to find a feasible schedule S which maximizes the profit of scheduled job. The goal can be achieved as follow: Sort all jobs in decreasing order of profit. Start with the empty schedule, select one job at a time and if it is feasible then schedule it in the *latest possible slot*.

Algorithm for Job Scheduling

Algorithm for job scheduling is described below:

```
Algorithm JOB_SCHEDULING( J, D, P )
// Description : Schedule the jobs using the greedy approach which maximizes the profit

// Input :
J: Array of N jobs
D: Array of the deadline for each job
P: Array of profit associated with each job

// Output : Set of scheduled job which gives maximum profit

Sort all jobs in J in decreasing order of profit
S ← Φ       // S is set of scheduled jobs, initially it is empty
SP ← 0      // Sum is the profit earned

for i ← 1 to N do
   if Job J[i] is feasible then
      Schedule the job in the latest possible free slot meeting its deadline.
      S ← S ∪ J[i]
      SP ← SP + P[i]
   end
end
```

Complexity Analysis of Job Scheduling

Simple greedy algorithm spends most of the time looking for the latest slot a job can use. On average, N jobs search N/2 slots. This would take $O(N^2)$ time.
However, with the use of set data structure (find and union), the algorithm runs nearly in O(N) time.

Examples Problem: Solve the following job scheduling with deadlines problem using the greedy method. Number of jobs N = 4. Profits associated with Jobs : $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$. Deadlines associated with jobs $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Solution:
Sort all jobs in descending order of profit.

So, P = (100, 27, 15, 10), J = (J_1, J_4, J_3, J_2) and D = (2, 1, 2, 1). We shall select one by one job from the list of sorted jobs, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one. Initially, Profit of scheduled jobs, SP = 0

Iteration 1:
Deadline for job J_1 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2. Solution set S= {J_1}, and Profit SP = {100}
Iteration 2:
Deadline for job J_4 is 1. Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.
Solution set S = {J_1,J_4}, and Profit SP = {100, 27}
Iteration 3:
Job J_3 is not feasible because first two slots are already occupied and if we schedule J_3 any time later t = 2, it cannot be finished before its deadline 2. So job J_3 is discarded,
Solution set S = {J_1,J_4}, and Profit SP = {100, 27}
Iteration 4:
Job J_2 is not feasible because first two slots are already occupied and if we schedule J_2 any time later t = 2, it cannot be finished before its deadline 1. So job J_2 is discarded,
Solution set S = {J_1,J_4}, and Profit SP = {100, 27}
With the greedy approach, we will be able to schedule two jobs {J_1, J_4}, which gives a profit of 100 + 27 = 127 units.

Assignment No 4) **Backtracking and Branch-n-Bound**

e. Knapsack problem,

The **knapsack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Refer Code from Knapsack

f. Travelling Salesman Problem

TSP Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **$(n - 1)!$** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph $G = (V, E)$, where V is a set of cities and E is a set of weighted edges. An edge $e(u, v)$ represents that vertices u and v are connected. Distance between vertex u and v is $d(u, v)$, which should be non-negative.

Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities $S \in \{1, 2, 3, \dots , n\}$ that includes **1**, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in **S** exactly once, starting at 1 and ending at j.

When $|S| > 1$, we define $C(S, 1) = \propto$ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at 1 and end at **j**. We should select the next city in such a way that

C(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠jC(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠j

Algorithm: Traveling-Salesman-Problem

```
C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
    C (S, 1) = ∞
  for all j Є S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis

There are at the most 2n.n2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2n.n2)O(2n.n2).

Refer Code from TSP

Assignment No 5) **Amortized Analysis**

g. Sorting algorithm

Merge Sort Algorithm

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Refer Code of Merge sort

QuickSort

Like Merge Sort, **QuickSort** is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Refer Code for Quick Sort

Assignment No 6) **Multithreaded and Distributed Algorithms**

Multiplication of Matrix using threads

Multiplication of matrix does take time surely. Time complexity of matrix multiplication is O(n^3) using normal matrix multiplication. And **Strassen algorithm** improves it and its time complexity is O(n^(2.8074)).
But, Is there any way to improve the performance of matrix multiplication using the normal method.

Multi-threading can be done to improve it. In multi-threading, instead of utilizing a single core of your processor, we utilizes all or more core to solve the problem. We create different threads, each thread evaluating some part of matrix multiplication.
Depending upon the number of cores your processor has, you can create the number of threads required. Although you can create as many threads as you need, a better way is to create each thread for one core.

In second approach,we create a separate thread for each element in resultant matrix. Using **pthread_exit()** we return computed value from each thread which is collected by **pthread_join()**. This approach does not make use of any global variables.

Refer code for Multithreaded matrix multiplication