RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	19CSE214A
Course Name	Design & Analysis of Algorithms
Programme	B. Tech
Department	Computer Science & Engineering
Faculty	Faculty of Engineering Technology

Name of the Student	SUBHENDU MAJI
Reg. No	18ETCS002121
Semester/Year	4th / 2020
Course Leader/s	Vaishali R Kulkarni

Declaration Sheet			
Student Name	SUBHENDU MAJI		
Reg. No	18ETCS002121		
Programme	B. Tech	Semester/Year	4th / 2020
Course Code	19CSE214A		
Course Title	Design and Analysis of Algorithms		
Course Date		To	
Course Leader	Vaishali R Kulkarni		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Assignment			
Reg.No.	18ETCS002121	Name of Student	SUBHENDU MAJI

Sections	Marking Scheme		Marks		
			Max Marks	First Examiner Marks	Moderator
Part A					
	A 1.1	Introduction	2		
	A 1.2	Set of case studies	2		
	A 1.5	Conclusion	1		
		Part-A Max Marks	05		
Part B					
	B 1.1	Pseudocode for each approach	3		
	B 1.2	Discussion on efficiency of algorithms	4		
	B 1.3	Results of three approaches using C programming	3		
		B.1 Max Marks	10		
	B 2.1	Pseudocode for each approach	3		
	B 2.2	Discussion on efficiency of algorithms.	3		
	B 2.3	Comparison of time/ space complexity for each algorithm.	4		
		B.2 Max Marks	10		
		Total Assignment Marks	25		

Course Marks Tabulation

Component-1 (B) Assignment	First Examiner	Remarks	Moderator	Remarks
A				
B.1				
B.2				
Marks (out of 25)				

Signature of First Examiner

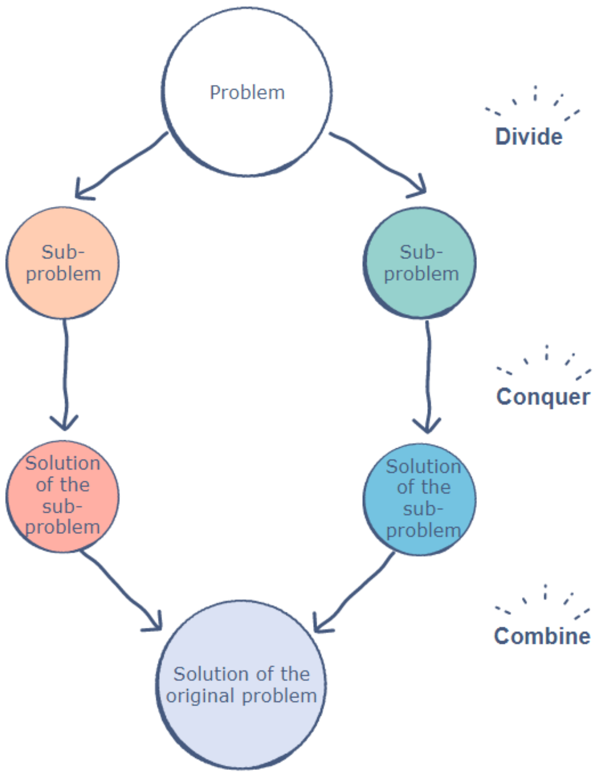
Signature of Moderator

Solution to Question No. 1:

A1.1 Introduction

Divide and conquer (DC) is one of the most important algorithmic techniques and can be used to solve a variety of computational problems.
Divide and conquer is an algorithm design paradigm based on multi-branched recursion.

A divide and conquer algorithm work by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.


Problem
Divide
Sub-problem
Sub-problem
Conquer
Solution of the sub-problem
Solution of the sub-problem
Combine
Solution of the original problem

Strategy:

1. Divide input into partitions of almost equal size.
2. Recursively solve the subproblems defined by the partition.
3. Combine the solutions to the subproblems into a single answer and pass it back as the answer to the recursion call

Base Case:
When the instance I of the problem P is sufficiently small, return the answer P(I) directly, or resort to a different, usually simpler, algorithm that is well suited for small instances.

Inductive Step:

1. **Divide** I into some number of smaller instances of the same problem P.
2. **Recurse** on each of the smaller instances to obtain their answers.
3. **Combine** the answers to produce an answer for the original instance I.

Divide-and-Conquer has several nice properties.

Firstly, it very closely follows the structure of an inductive proof, and therefore most often leads to rather simple proofs of correctness. As in induction, one first needs to prove the base case is correct. Then one can assume by strong (or structural) induction that the recursive solutions are correct, and needs to show that given correct solutions to each smaller instance the combined solution is a correct answer.

A second nice property is that divide-and-conquer can lead to quite efficient solutions to a problem. However, to be efficient one needs to be sure that the divide and combine steps are efficient, and that they do not create to many sub instances. This brings us to the third nice property, which is that the work and span for divide-and-conquer algorithms can be expressed as a form of mathematical equations called recurrences. Often these recurrences can be solved without too much difficulty making analyzing the work and span of many divide-and-conquer algorithms reasonably straightforward.

Finally, divide-and-conquer is a naturally parallel algorithmic technique. Most often we can solve the sub instances in parallel. This can lead to significant amount of parallelism since at each level of can create more instances to solve in parallel. Even if we only divide our instance into two sub instances, each of those sub instances will themselves generate two more sub-instances, and this repeat.

Generalized Divide & Conquer:

```
Input: I (an input to the given problem)
Output: J (a solution to problem for input I)

Algorithm Divide-and-Conquer (I, J)
    if I ∈ Known-Answers then
        return J (a solution to I)
    else
        Divide I into m >= 2 disjoint subproblems (I1, · · · , Im
)
        for k = 1...m do
            Divide-and-Conquer(Ik , Jk )
        Combine solutions (J1, · · · , Jm) into solution J
        return J
end
```

A1.2 Set of case studies

We divide the problem up to solve many smaller problems. It is just like recursion. We need to know when to stop.
A divide and conquer algorithm try to break a problem down into as many little chunks as possible since it is easier to solve with little chunks. It typically does this with recursion.

Examples of divide and conquer include binary search, Quick sort, Merge sort, Fibonacci number calculations, Tower of Hanoi etc.

- **Binary Search** is a searching algorithm. In each step, the algorithm compares the input element (x) with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs to the left side of the middle element, else it recurs to the right side of the middle element.

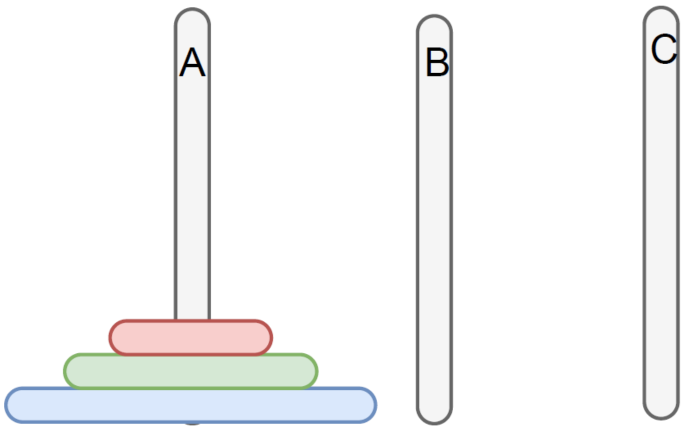
- **Quicksort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

- **Merge Sort** is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves. The time complexity of this algorithm is $O(nLogn)$, be it best case, average case or worst case. Its time complexity can be easily understood from the recurrence equates to: $T(n) = 2T(n/2) + n$.

- **Closest Pair of Points** The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solve the problem in $O(nLogn)$ time.

- **Towers of Hanoi**

  The Towers of Hanoi is a mathematical problem which consists of 3 pegs and 3 and a number of discs. In this instance, 3 discs.



Each disc is a different size. We want to move all discs to peg C so that the largest is on the bottom, second largest on bottom — 1 etc.

- We can only move 1 disc at a time.
- A disc cannot be placed on top of other discs that are smaller than it.
- We want to use the smallest number of moves possible.
- If we have 1 disc, we only need to move it once.
- If we have 2 discs, we need to move it 3 times.

We need to store the smallest disc in a buffer peg (1 move), move the largest disc to peg C (2 moves) and move the buffer disc to peg C (3 moves).

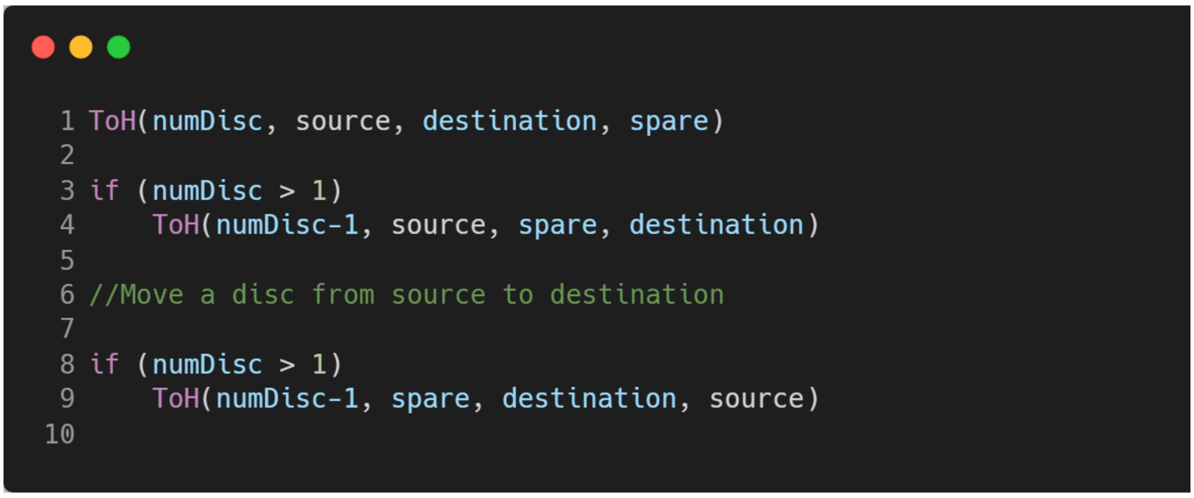
When we have 4 discs, we need to make 15 moves. 5 discs is 31.
These move numbers are powers of 2 minus 1.

To find out how many moves a Tower of Hanoi solution takes you calculate $(2^n) - 1$ where n is how many discs there are.

We can generalize this problem.

- If we have n discs: move n-1 from A to B recursively, move largest from A to C, move n-1 from B to C recursively.
- If there is an even number of pieces the first move is always into the middle. If there are an odd number of pieces the first move is always to the other end.

Algorithm:



```
ToH(numDisc, source, destination, spare)

if (numDisc > 1)
    ToH(numDisc-1, source, spare, destination)

//Move a disc from source to destination

if (numDisc > 1)
    ToH(numDisc-1, spare, destination, source)

```

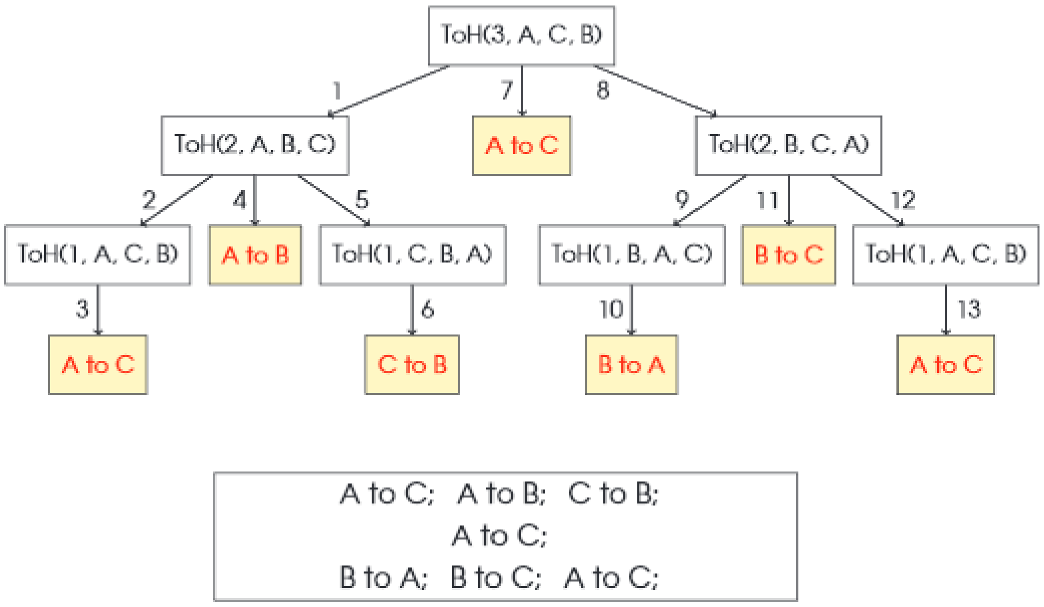


Figure 1 the execution tree for TOH algorithm

the execution tree for the above algorithm with $\boldsymbol{ToH}(\mathbf{3}, \boldsymbol{A}, \boldsymbol{C}, \boldsymbol{B})$.
Once we call that we 2 calls to:

- $\boldsymbol{ToH}(2, A, B, C)$
- $\boldsymbol{ToH}(2, B, C, A)$

Since 2 is more than 1, we move it down one more level again.

- **Fibonacci Numbers**

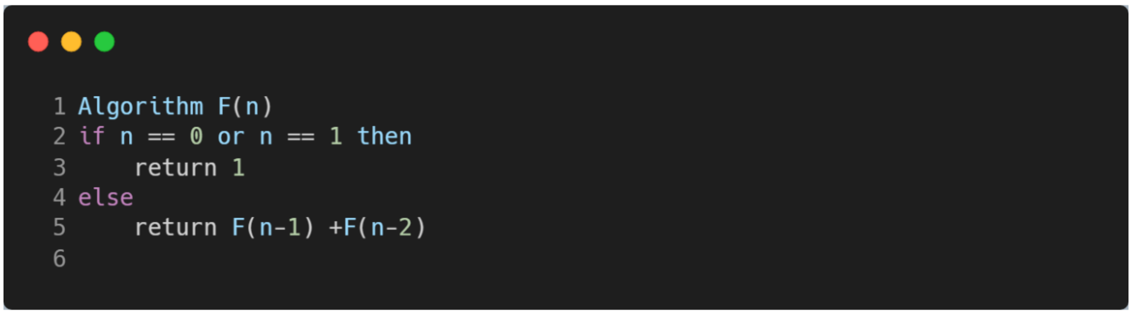
The Fibonacci numbers can be found in nature. They start at 1 and the next number is the current number + the previous number. Here it's 1 + 1 = 2. Then 2 + 1 = 3. 3 + 2 = 5 and so on.

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

This is the formal definition of the Fibonacci numbers. If n = 0 or n = 1, output 1. Else sum the previous numbers.

$\boldsymbol{n}$	0	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{F}(\boldsymbol{n})$	1	1	2	3	5	8	13	21	34	55	89

Figure 2 Execution table for F(n)



```
Algorithm F(n)
if n == 0 or n == 1 then
    return 1
else
    return F(n-1) +F(n-2)
```

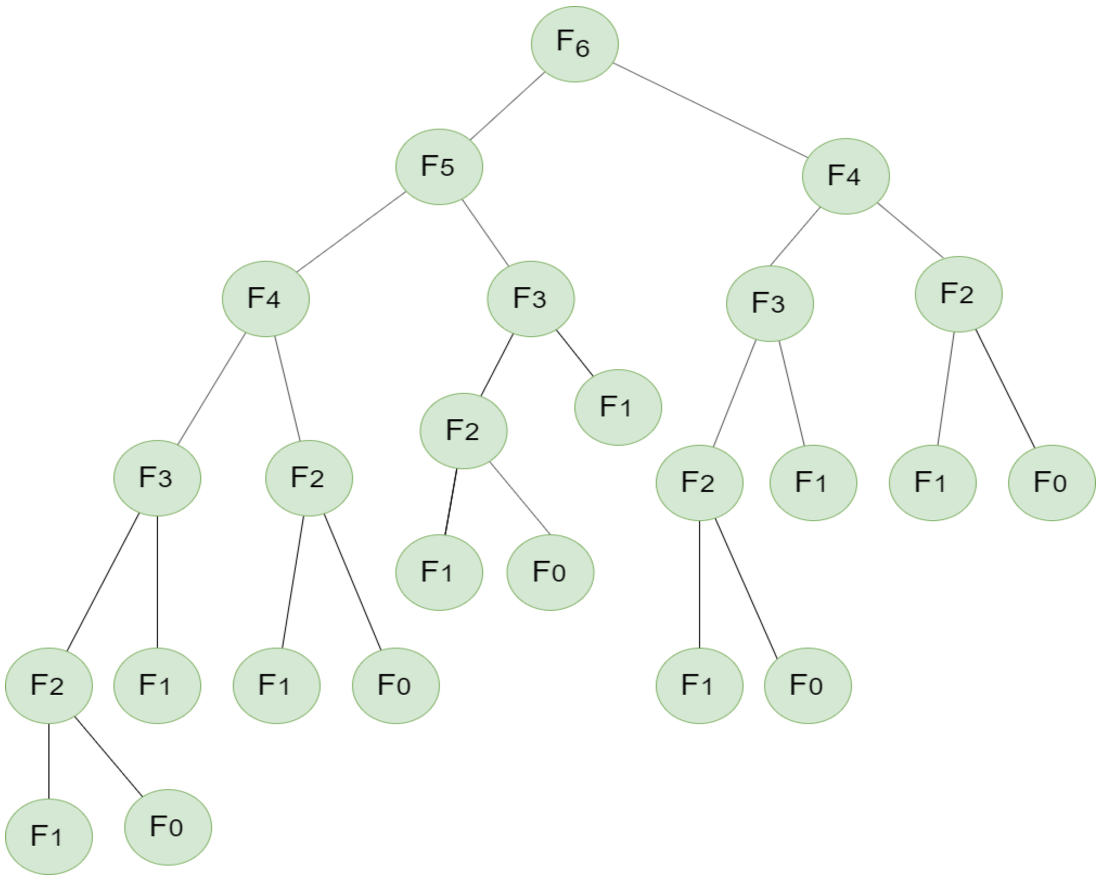


Figure 3 Execution of F(6)

Here, N is 6. N is more than 0 or 1, so we ignore that part.
We then calculate $F(5) + F(4)$.
$F(5)$ becomes $F(4) + F(2)$.

Eventually when we get down to the base cases (when F is 0 or 1) we end up with the number 1. We return the previous number to the upper level.

So $F(1) + F(0) = 2$ and then we get $F2 + F1 = 2 + 1 = 3$.

A1.3 Conclusion

The divide and conquer method indeed do have a much more efficient search pattern compared to that of the exhaustive search. However, it must be noted that the divide and conquer method does not find every single possible cycle. Contrary to the divide and conquer, the exhaustive method can successfully list out all the paths, but at a high time-consuming cost. Though reliable in the sense that given a smaller problem, the solution can be quickly searched and optimized. However, in many worldly problems, the graph size is often of large magnitude, and requires a faster algorithm. Given the massive size of some optimization problems, it is sometimes ideal to forgo optimal solutions and settle for a reasonable option calculated in a faster method.

In this respect, the divide and conquer method is a viable approach in solving NP-complete optimization problems, as it performs computational task faster than brute force exhaustive methods. In a larger graph scenario, the divide and conquer will be able to calculate a reasonable solution within a shorter period time and is more pragmatic compared to the exhaustive method.

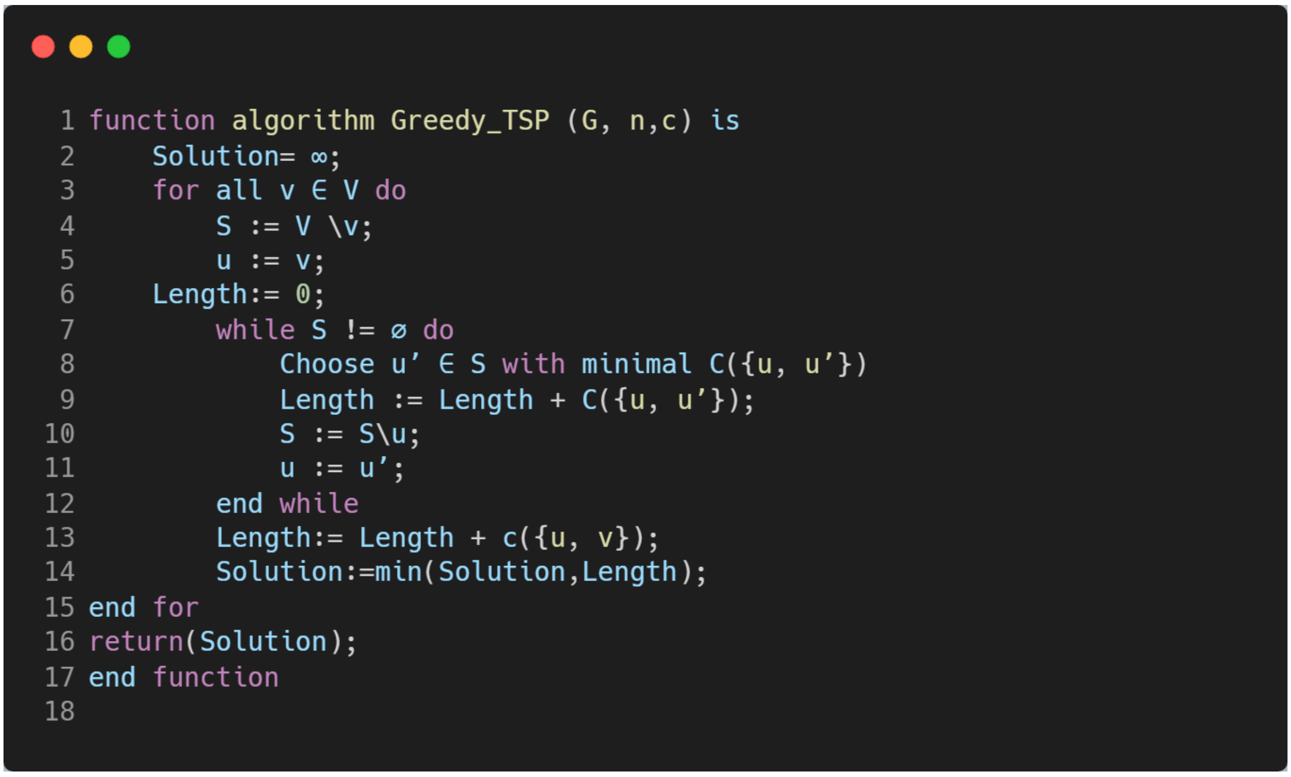
Solution to Question No. 2:

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?

B1.1 Pseudocode for each approach

Greedy Algorithm:

Greedy algorithm is the simplest improvement algorithm. It starts with the departure Node 1. Then the algorithm calculates all the distances to other n-1 nodes. Go to the next closest node. Take the current node as the departing node, and select the next nearest node from the remaining n-2 nodes. The process continues until all the nodes are visited once and only once then back to
Node 1. When the algorithm is terminated, the sequence is returned as the best tour; and its associated OFV is the final solution. The advantage of this algorithm is its simplicity to understand and implement. Sometime, it may lead to a very good solution when the problem size is small. Also, because this algorithm does not entail any exchange of nodes, it saves considerably much computational time

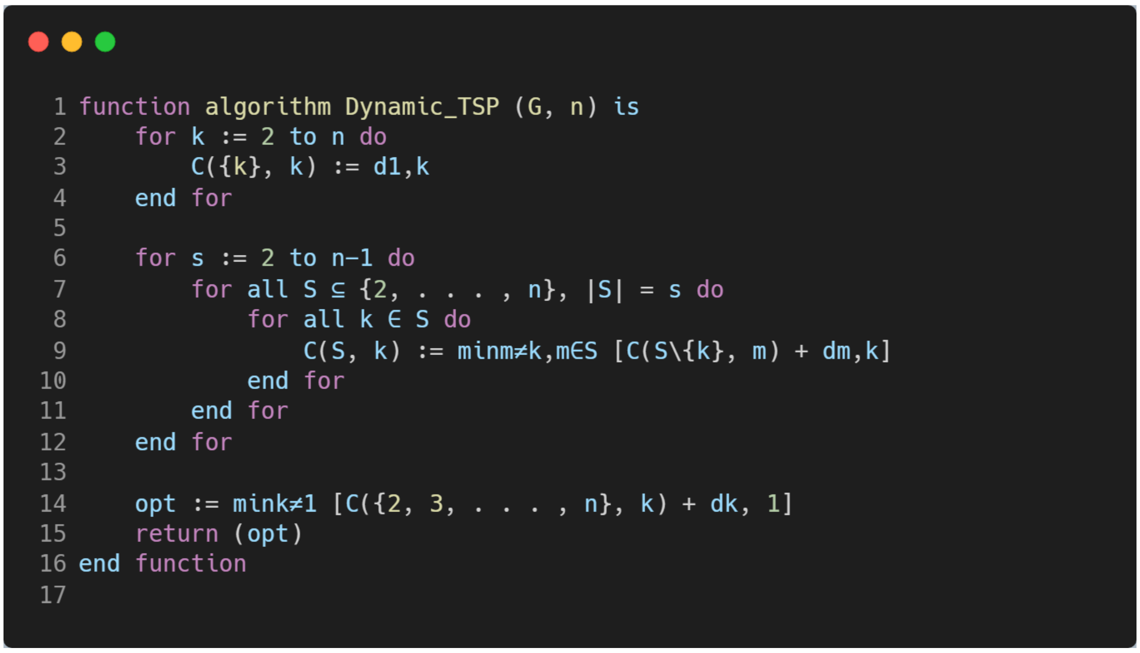


```
function algorithm Greedy_TSP (G, n,c) is
    Solution= ∞;
    for all v ∈ V do
        S := V \v;
        u := v;
    Length:= 0;
        while S != ∅ do
            Choose u’ ∈ S with minimal C({u, u’})
            Length := Length + C({u, u’});
            S := S\u;
            u := u’;
        end while
        Length:= Length + c({u, v});
        Solution:=min(Solution,Length);
end for
return(Solution);
end function
```

Dynamic Programing:

Number the cities 1, 2, . . . , N and assume we start at city 1, and the distance between city i and city j is $d_{i,j}$. Consider subsets $S \subseteq \{2, . . . , N\}$ of cities and, for $c \in S$, let $D(S, c)$ be the minimum distance, starting at city 1, visiting all cities in S and finishing at city c.
First phase: if $S = \{c\}$, then $D(S, c) = d_{1,c}$. Otherwise: $D(S, c) = \min_{x \in S-c} (D(S - c, x) + d_{x,c})$.

Second phase: the minimum distance for a complete tour of all cities is
$M = \min_{c\in\{2,\dots,N\}} (D(\{2, . . . , N\}, c) + d_{c,1})$
A tour $n_1, . . ., n_N$ is of minimum distance just when it satisfies $M = D(\{2, . . . , N\}, n_N) + d_{n_N,1}$



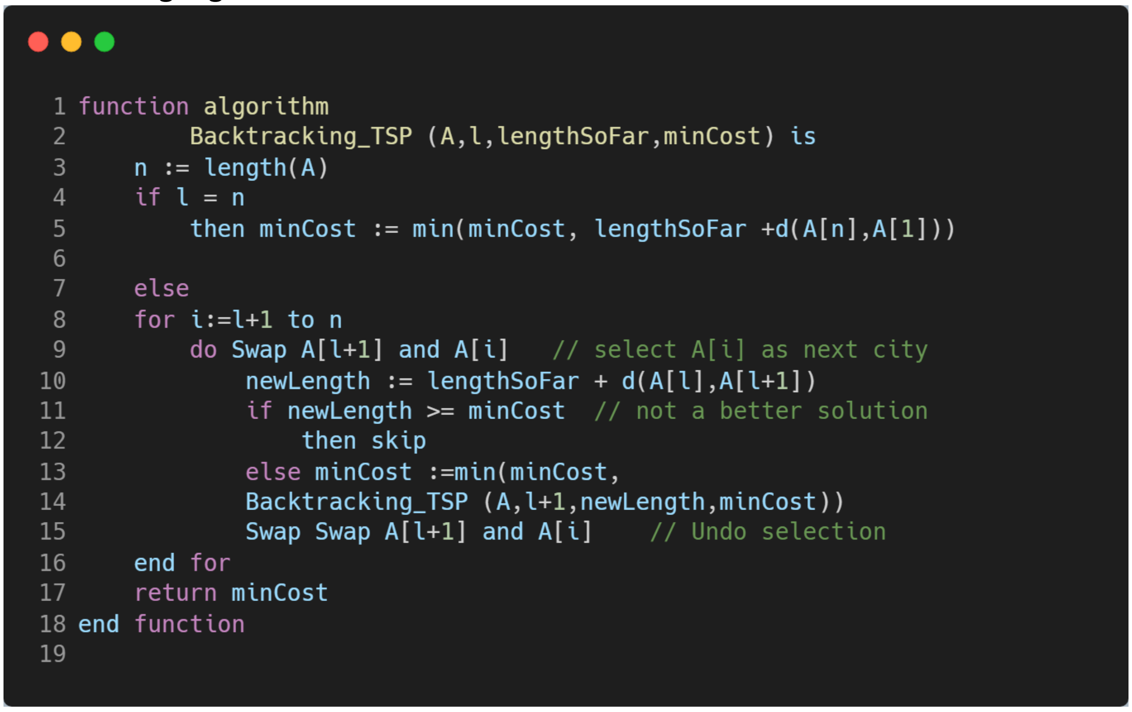
```
function algorithm Dynamic_TSP (G, n) is
    for k := 2 to n do
        C({k}, k) := d1,k
    end for

    for s := 2 to n−1 do
        for all S ⊆ {2, . . . , n}, |S| = s do
            for all k ∈ S do
                C(S, k) := minm≠k,m∈S [C(S\{k}, m) + dm,k]
            end for
        end for
    end for

    opt := mink≠1 [C({2, 3, . . . , n}, k) + dk, 1]
    return (opt)
end function

```

Backtracking Algorithm:



```
function algorithm
        Backtracking_TSP (A,l,lengthSoFar,minCost) is
    n := length(A)
    if l = n
        then minCost := min(minCost, lengthSoFar +d(A[n],A[1]))

    else
    for i:=l+1 to n
        do Swap A[l+1] and A[i]    // select A[i] as next city
            newLength := lengthSoFar + d(A[l],A[l+1])
            if newLength >= minCost  // not a better solution
                then skip
            else minCost :=min(minCost,
            Backtracking_TSP (A,l+1,newLength,minCost))
            Swap Swap A[l+1] and A[i]    // Undo selection
    end for
    return minCost
end function

```

B1.2 Discussion on efficiency of algorithms

Travelling Salesman problem is NP hard problem and doesn't have any solution in polynomial time.
In a **brute-force approach**, we need to generate all the (n-1)! permutations of cities and calculate cost of every permutation and keep track of minimum cost permutation, this has a time complexity of **O(n!)** which is worse than exponential time.

This can be improved by a using the backtracking approach, pruning: as soon as we see a city in a tour that we know will never yield an optimal solution, we can save time by skipping this tour and heading onto the next one. In the **worst case** the time complexity is the **same as brute-force O(n!)**, as we may never get a chance to prune a node. The performance in this approach depends on the different instances, so it performs really well but some it's as bad as brute-force.

In the dynamic programing approach, we get the **Recursive Equation**

T (i , s) = min ((i , j) + T (j , S – { j })) ; S!= Ø ; j € S ;

S is set that contains non visited vertices= (i, 1) ; S=Ø,
This is base condition for this recursive equation.
Here, T (i, S) means We are travelling from a vertex "i" and have to visit set of non-visited vertices "S" and have to go back to vertex 1 (let we started from vertex 1).
(i, j) means cost of path from node i to node j
If we observe the first recursive equation from a node we are finding cost to all other nodes (i,j) and from that node to remaining using recursion (T (j , {S-j}))
But it is not guarantee that every vertex is connected to other vertex then we take that cost as infinity. After that we are taking minimum among all so the path which is not connected get infinity in calculation and won't be consider.
If S is empty that means we visited all nodes, we take distance from that last visited node to node 1 (first node). Because after visiting all he has to go back to initial node.

To find the **Time Complexity:**
Since we are solving this using Dynamic Programming, we know that Dynamic Programming approach contains sub-problems.
Here after reaching ith node finding remaining minimum distance to that ith node is a sub-problem.
If we solve recursive equation, we will get total $\boldsymbol{(n-1)\ 2(n-2)}$ sub-problems, which $is\ \boldsymbol{O\ (n2n)}$.
Each sub-problem will take **O (n)** time (finding path to remaining **(n-1)** nodes).
Therefore, total time complexity is $\boldsymbol{O\ (n2n)\ *\ O\ (n)\ =\ O\ (n22n)}$
Space complexity is also number of sub-problems which is $\boldsymbol{O\ (n2n)}$

The Greedy Approach is just a **Heuristic algorithm** or an **approximation algorithm**, which quickly gives a good solution if not the optimal solution. a greedy algorithm) lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. For N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path. However, there exist many specially arranged city distributions which

make the NN algorithm give the worst route. This is perhaps the simplest and most straight forward TSP heuristic.
The key to this algorithm is to always visit the nearest city, then return to the starting city when all the other cities are visited, because we have to find the city with the shortest path which isn't visited from a given city, this takes O(n) as there are n cities we need to repeat this for all of them, it takes $\boldsymbol{O}(\boldsymbol{n2})$ although this is very fast compared to both it doesn't give the optimal solution.

B1.3 Results of three approaches using C programming

Dynamic Programing:

Distance matrix:

$$C = \begin{pmatrix} 00 & 02 & 09 & 10 \\ 01 & 00 & 06 & 04 \\ 15 & 07 & 00 & 08 \\ 06 & 03 & 12 & 00 \end{pmatrix}$$

Functions description:

- *g(x, S)* - starting from 1, path min cost ends at vertex x, passing vertices in set S exactly once
- *c_{xy}* - edge cost ends at x from y
- *p(x, S)* - the second-to-last vertex to x from set S. Used for constructing the TSP path back at the end.

k = 0, null set:

Set Ø:

```
g(2, Ø) = c₂₁ = 1
g(3, Ø) = c₃₁ = 15
g(4, Ø) = c₄₁ = 6
```

k = 1, consider sets of 1 element:

Set {2}:

```
g(3,{2}) = c₃₂ + g(2, Ø ) = c₃₂ + c₂₁ = 7 + 1 = 8        p(3,{2}) = 2
g(4,{2}) = c₄₂ + g(2, Ø ) = c₄₂ + c₂₁ = 3 + 1 = 4        p(4,{2}) = 2
```

Set {3}:

```
g(2,{3}) = c₂₃ + g(3, Ø ) = c₂₃ + c₃₁ = 6 + 15 = 21      p(2,{3}) = 3
g(4,{3}) = c₄₃ + g(3, Ø ) = c₄₃ + c₃₁ = 12 + 15 = 27     p(4,{3}) = 3
```

Set {4}:

```
        g(2,{4}) = c24 + g(4, Ø ) = c24 + c41 = 4 + 6 = 10      p(2,{4}) = 4
        g(3,{4}) = c34 + g(4, Ø ) = c34 + c41 = 8 + 6 = 14      p(3,{4}) = 4
```

k = 2, consider sets of 2 elements:

Set {2,3}:

```
         g(4,{2,3}) = min {c42 + g(2,{3}), c43 + g(3,{2})} = min {3+21,
12+8}= min {24, 20}= 20
         p(4,{2,3}) = 3
```

Set {2,4}:

```
         g(3,{2,4}) = min {c32 + g(2,{4}), c34 + g(4,{2})} = min {7+10, 8+4}=
min {17, 12} = 12
         p(3,{2,4}) = 4
```

Set {3,4}:

```
          g(2,{3,4}) = min {c23 + g(3,{4}), c24 + g(4,{3})} = min {6+14,
4+27}= min {20, 31}= 20
          p(2,{3,4}) = 3
```

Length of an optimal tour:

```
 f = g(1,{2,3,4}) = min { c12 + g(2,{3,4}), c13 + g(3,{2,4}), c14 + g(4,{2,3})
}
                  = min {2 + 20, 9 + 12, 10 + 20} = min {22, 21, 30} = 21
```

Successor of node 1: p(1,{2,3,4}) = 3

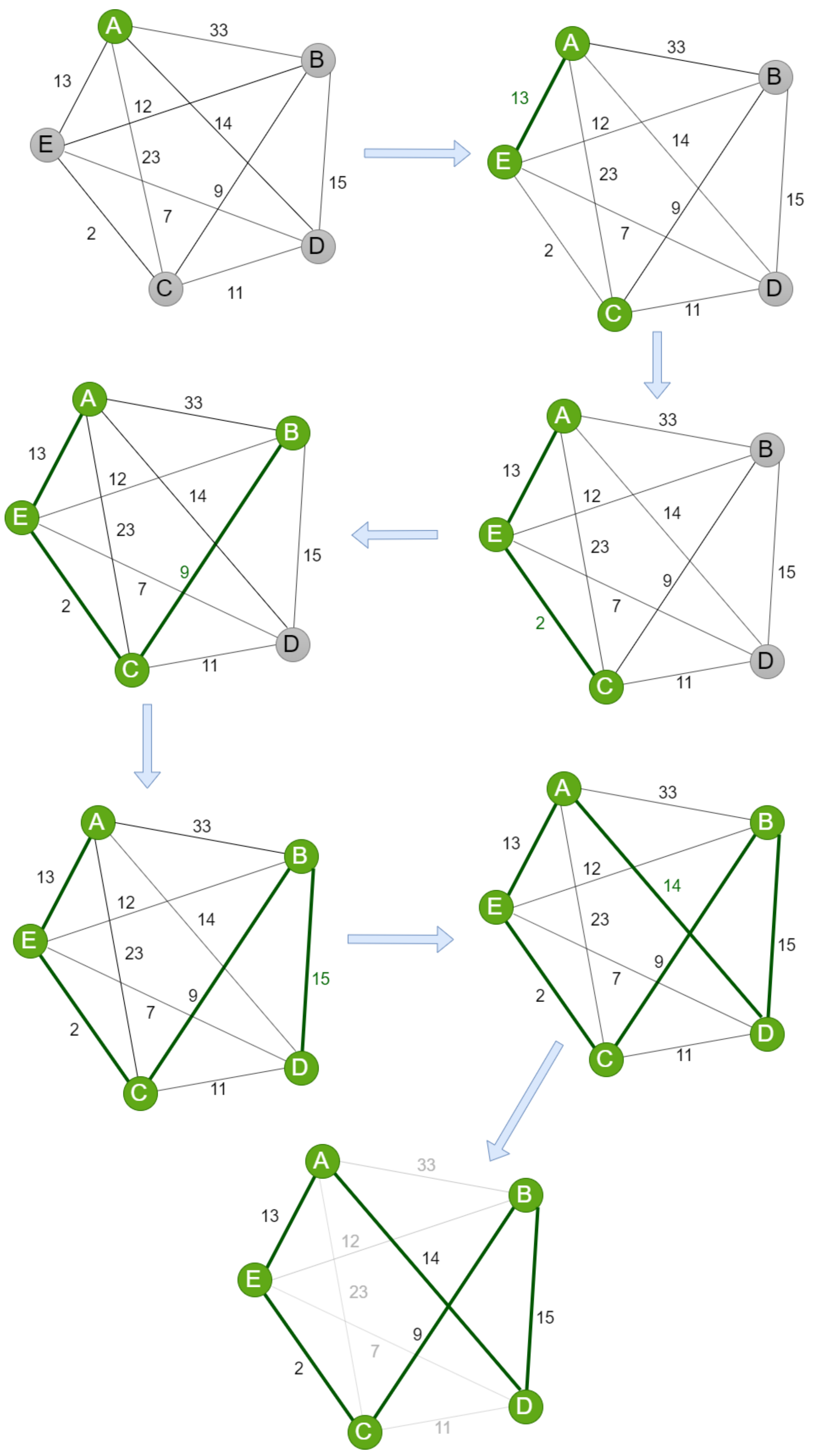
Successor of node 3: p(3, {2,4}) = 4

Successor of node 4: p(4, {2}) = 2

Backtracking the optimal TSP tour reaches: 1 → 3 → 4 → 2 → 1

Greedy Algorithm:

We always choose the city which has the smallest cost from the current city and repeat the process until we get back to the city we started from.



The total distance of the path $\boldsymbol{A} \rightarrow \boldsymbol{E} \rightarrow \boldsymbol{C} \rightarrow \boldsymbol{B} \rightarrow \boldsymbol{D} \rightarrow \boldsymbol{A}$ obtained using the greedy method is $\mathbf{13 + 2 + 9 + 15 + 24 = 42.}$

Backtracking:

In this approach we update the minCost and the CostSoFar for each branch.

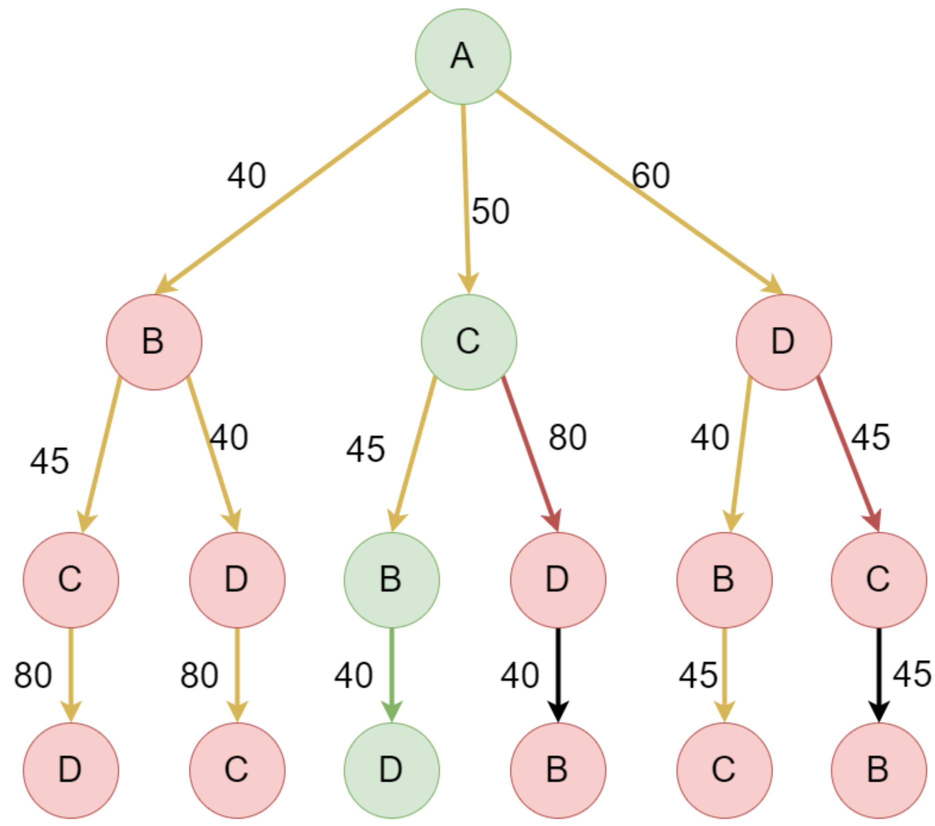
Initially the minCost is infinity or max_int, the sum cost of first branch is 165 so we update the minCost to 165.

In the next branch the cost at the end is 160 so we update the minCost to 160

In the next branch the cost at the end is 135 so we update the minCost to 135

In the next branch the costSoFar exceeds minCost at D so we skip this branch

Similarly, for the next two branches, so the final cost is 135.

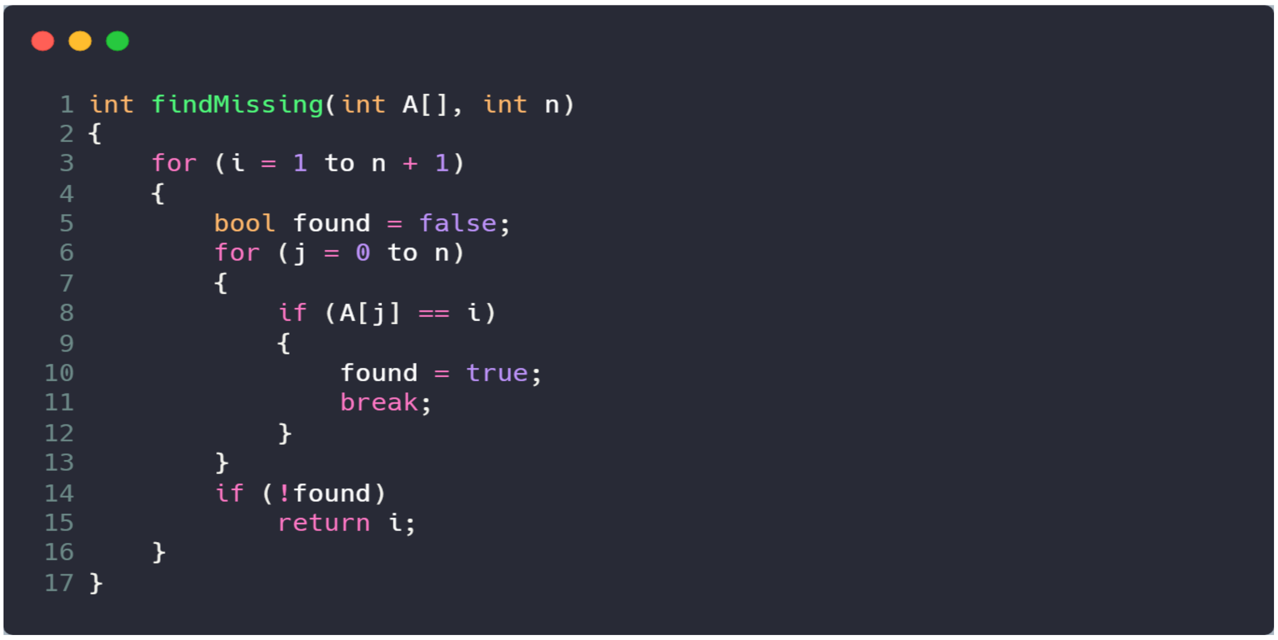


Solution to Question No. 3:

We are given a list of n-1 integers which are in the range of 1 to n. There are no duplicates in the list. One of the integers is missing in the list. Our job is to find the missing integer.
Example:
Input: [1, 2, 4, 6, 3, 7, 8]
Output: 5.
This problem can be solved by three approaches, i.e. Brute Force, Sorting, and Hashing Approach.

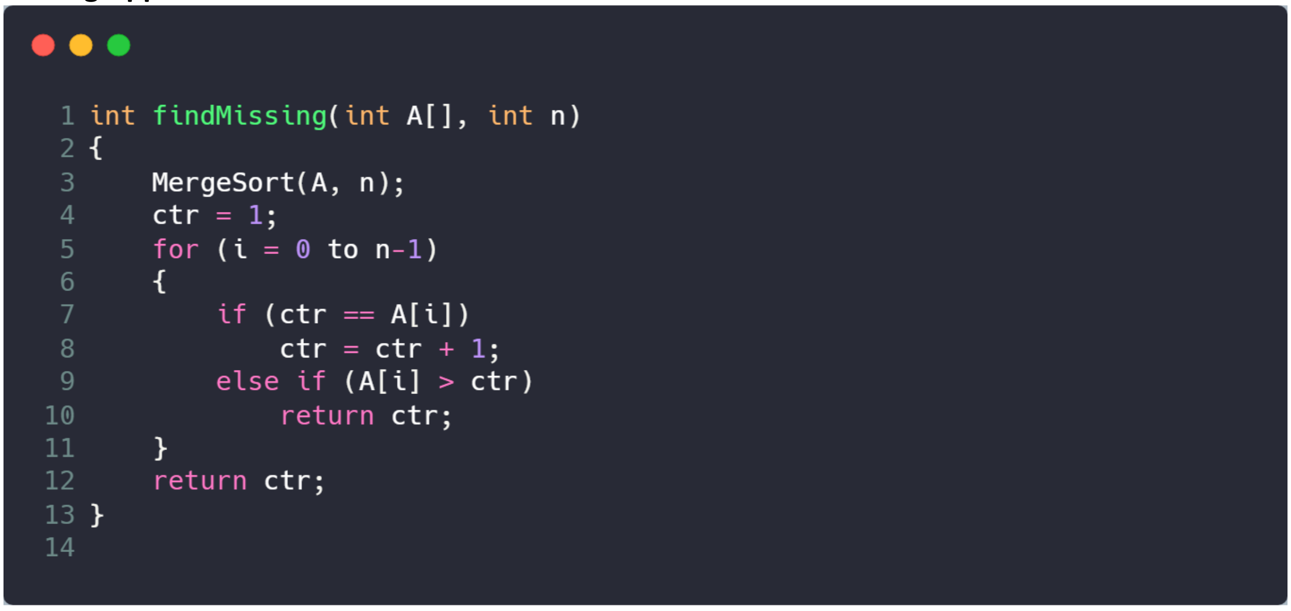
B2.1 Pseudocode for each approach

Brute Force Approach



```
int findMissing(int A[], int n)
{
    for (i = 1 to n + 1)
    {
        bool found = false;
        for (j = 0 to n)
        {
            if (A[j] == i)
            {
                found = true;
                break;
            }
        }
        if (!found)
            return i;
    }
}
```

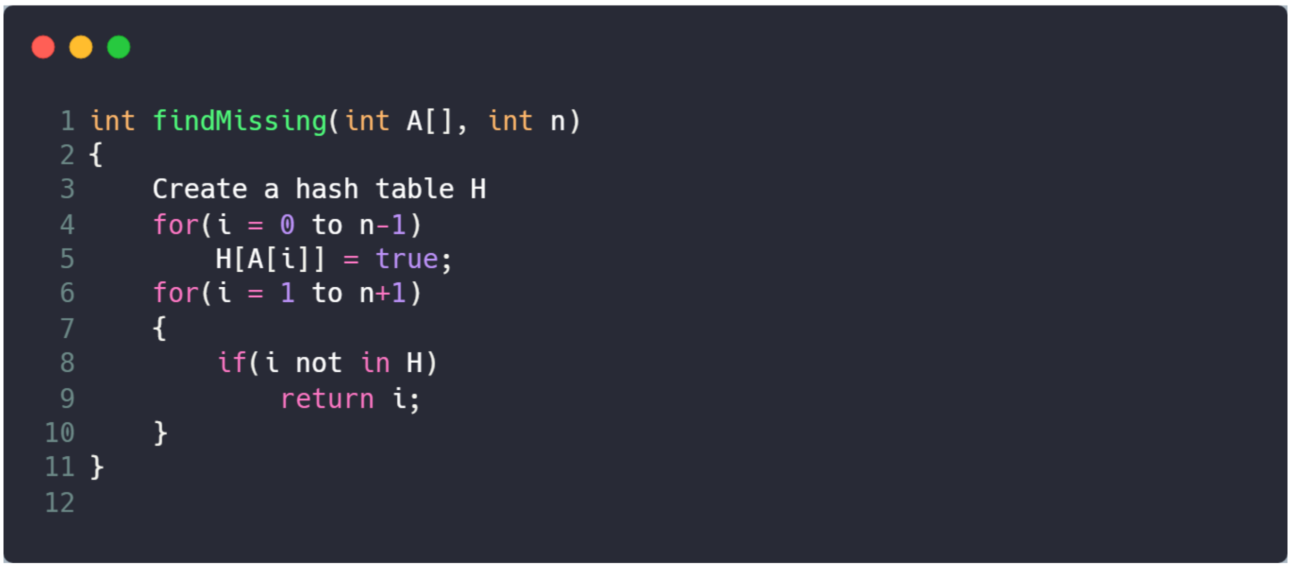
Sorting Approach



```
int findMissing(int A[], int n)
{
    MergeSort(A, n);
    ctr = 1;
    for (i = 0 to n-1)
    {
        if (ctr == A[i])
            ctr = ctr + 1;
        else if (A[i] > ctr)
            return ctr;
    }
    return ctr;
}

```

Hashing Approach



```
int findMissing(int A[], int n)
{
    Create a hash table H
    for(i = 0 to n-1)
        H[A[i]] = true;
    for(i = 1 to n+1)
    {
        if(i not in H)
            return i;
    }
}
```

B2.2 Discussion on efficiency of algorithms.

Brute Force Approach

In this algorithm, we are iterating through each element from 1 till n elements and searching it throughout the array.
This is the simplest solution to find the missing element from an unsorted array.
At worst case, the algorithms will have to search for n+1 elements.
Hence, the time complexity of this is $\boldsymbol{O(n^2)}$.
This is not a very efficient solution as it has high time complexity.

Sorting Approach

In this algorithm, we are sorting using a sorting technique. Merge Sort technique will be well suited as it has $O(n\ logn)$ complexity for best, average and worst case.
After sorting each element is compared with a counter variable.
Time Complexity: sorting + iterating through each element of array

$$= O(n \log n) + O(n) = \boldsymbol{O(n \log n)}$$

This is a simple & efficient algorithm, although the complexity of the algorithm mainly depends on the sorting technique used.

Hashing Approach

In this algorithm, we are building a hash table to store the integers present in the array. Then we will look for integer starting from 1 to n+1 in the hash table. Steps:

1. Create a hash table H.
2. Insert the elements of the array in the hash table.
3. Iterate i from 1 to n+1 and look for i in H.
4. If i is not present in H, return i.
5. Else, continue the iteration.

Time complexity = Building hash table + Searching element in Hash Table.

$$= O(n) + O(n) = \boldsymbol{O(n)}$$

This is the by far the most efficient in terms of complexity, but it is a complex algorithm and has space complexity of $O(n)$.

B2.3 Comparison of time/ space complexity for each algorithm.

Brute Force Approach

```
int findMissing(int A[], int n)
{
    for (i = 1 to i = n + 1)                ----(n+1)
    {
        bool found = false                  ---- n(1)
        for (j = 0 to j = n)                ---- n(n+1)
        {
            if (A[j] == i)                  ----n(n+1)(1)
            {
                found = true;               ----n(n+1)(1)
                break
            }
        }
        if (!found)                         ----n
            return i
    }
}
```

Time Complexity: $n + 1 + n + n(n+1) + \mathrm{n}(\mathrm{n}+1) + \mathrm{n}(\mathrm{n}+1) + n$

$$= 3n(n+1) + 3n + 1 = \boldsymbol{O(n^2)}$$

Space Complexity: $1 + 1 + 1 = 3 = \boldsymbol{O(1)}$

Sorting Approach

```
int findMissing(int A[], int n)
{
    MergeSort(A)                    ----n(log n)
    ctr = 1                         ----1
    for(i = 0 to n-1)               ----n+1
    {
        if(ctr == A[i])             ----n
            ctr = ctr + 1
        if(A[i] > ctr)              ----n
            return ctr
    }
    return ctr
}
```

Time Complexity: $n(\log n)+1+n+1+n+n = n\log n+3n+2 = \boldsymbol{O(n\log n)}$
Space Complexity: $1 = \boldsymbol{O(1)}$

Hashing Approach

```
int findMissing(int A[], int n)
{
    Create a hash table H
    for(i = 0 to n-1)                    ----n
        H[A[i]] = true                   ----n(1)
    for(i = 1 to n+1)                    ----n
    {
        if(i not in H)                   ----n(1)
            return i
    }
}
```

Time Complexity: $n+n+n+n = 4n = \boldsymbol{O(n)}$
Space Complexity: $creating\ hash\ table[n] = \boldsymbol{O(n)}$

Approach	Time Complexity	Space Complexity
Brute Force	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(1)}$
Sorting	$\boldsymbol{O(n\ log\ n)}$	$\boldsymbol{O(1)}$
Hashing	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$

Bibliography

- https://afteracademy.com/blog/find-missing-positive
- https://medium.com/brandons-computer-science-notes/divide-and-conquer-algorithms-4e83d9999ffa
- http://cs.indstate.edu/~zeeshan/aman.pdf
- http://www.cs.cmu.edu/afs/cs/academic/class/15210-f14/www/lectures/dandc.pdf
- https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/?ref=rp
- https://cs.stackexchange.com/questions/90149/analysis-of-time-complexity-of-travelling-salesman-problem