

INFO 6205 – Program Structure and Algorithms

Assignment 2

Student Name: Vidhi Bharat Patel

Professor: Nik Bear Brown

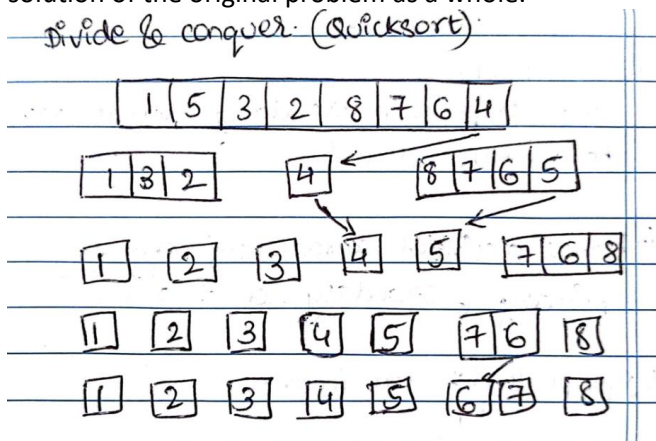
Instructions:

- Support your approach with algorithms and run through an example if possible.
- Write pseudo codes wherever possible.
- Your code should be a runnable Jupyter notebook.

Q1(5 points)

Explain divide and conquer theorem with code and example.

- The divide and conquer theorem is basically an algorithm pattern where a huge input is taken and then it is broken down into minor pieces, then decide the problem on each of the small pieces, lastly merge the solutions into a global solution.
- In simple terms, first divide the original problem into a set of sub problems, then conquer or solve every subproblem individually and recursively, lastly, combine together the solutions of the subproblems to get the solution of the original problem as a whole.



- Some algorithms which follow the divide and conquer theorem are: Maximum and minimum problem, binary search, sorting which includes quick sort and merge sort, as well as Tower of Hanoi.
- The code for the divide and conquer in quicksort can be given as followed: -

```
def quicksort(arr):  
    n = len(arr)  
    if n <= 1:  
        return arr  
    else:  
        # Choose a pivot element  
        pivot = arr.pop()  
  
        # Partition the array into two subarrays  
        left = []  
        right = []  
        for element in arr:  
            if element <= pivot:  
                left.append(element)  
            else:  
                right.append(element)  
  
        # Recursively sort the subarrays  
        left_sorted = quicksort(left)  
        right_sorted = quicksort(right)  
  
        # Combine the results (no combining needed)  
        return left_sorted + [pivot] + right_sorted
```

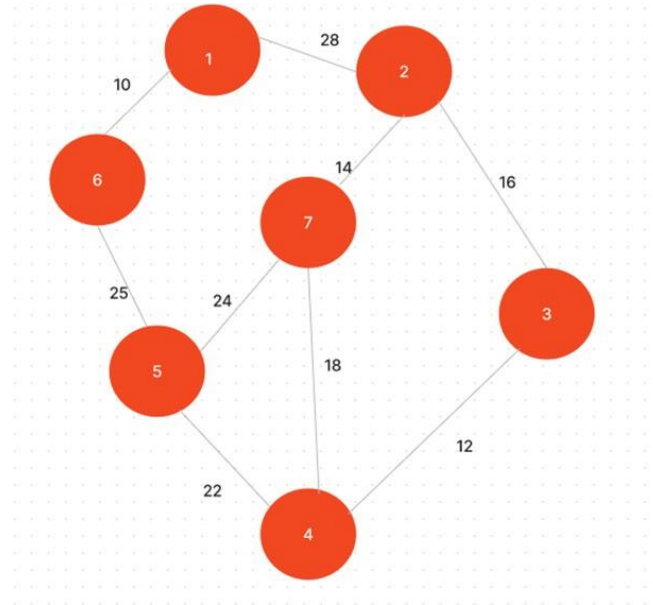
- Here, the algorithm will first check if the array has only one element or it is empty. If so, the array is already sorted and is returned as it is. If not, the algorithm will then choose a pivot element and partitions the array into

two subarrays using that pivot.

- It will then recursively sort the two subarrays using the quicksort function and combine the sorted subarrays to obtain the final sorted array.
- The time complexity of QuickSort using the divide and conquer approach is $O(n \log n)$ on average, and $O(n^2)$ in the worst case.
- The worst case occurs when the pivot element is consistently chosen to be the smallest or largest element in the array, causing the recursion tree to be unbalanced.
- However, by choosing a good pivot element and using various techniques to optimize the algorithm, the worst-case scenario can be avoided in practice.

Q2(5 points)

Use Kruskal's algorithm to find a minimum spanning tree for the connected weighted graph below:



What is the Time Complexity of Kruskal's algorithm?

2) Kruskal's

weight	source
10	1-6
12	3-4
14	2-7
16	2-3
18	4-7 x
22	4-5
24	5-7 x
25	5-6
28	1-2 x

```

#Q2
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    # Search function
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # Applying Kruskal algorithm
    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("%d - %d: %d" % (u, v, weight))

g = Graph(7)
g.add_edge(0, 1, 28)
g.add_edge(0, 5, 10)
g.add_edge(1, 2, 16)
g.add_edge(1, 6, 14)
g.add_edge(2, 3, 12)
g.add_edge(3, 4, 22)
g.add_edge(3, 6, 18)
g.add_edge(4, 5, 25)
g.add_edge(4, 6, 24)

g.kruskal_algo()

```

```

0 - 5: 10
2 - 3: 12
1 - 6: 14
1 - 2: 16
3 - 4: 22
4 - 5: 25

```

The time complexity of Kruskal's algorithm implemented in this code is $O(E \log E)$, where E is the number of edges in the graph. Since the edges are sorted in non-decreasing order of their weights, the sorting step takes $O(E \log E)$ time, and the find and union operations in the while loop together take $O(E \log^* V)$ time, where V is the number of vertices in the graph and \log^* denotes the iterated logarithm function. Therefore, the total time complexity is $O(E \log E + E \log^* V)$.

Q3 (10 Points)

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

i. $T(n) = 4T(n/4) + n \log n$

Master Theorem can be applied here,

$a = 4, b = 4, f(n) = n \log n$.

In this case, the runtime complexity can be determined to be $\Theta(n \log n)$ using the second case of the Master Theorem:

If $f(n) = \Theta(n \log b a(\log k + 1) n)$ where $k \geq 0$, then $T(n) = \Theta(n \log b a(\log k + 1) n)$.

ii. $T(n) = 4T(n/3) + n^{0.33}$

The Master theorem cannot be directly applied to this recurrence relation because it does not satisfy the required form for the master theorem, which is $T(n) = aT(n/b) + f(n)$. Specifically, the exponent of n in the recurrence relation's subproblem size reduction factor is not a log base b expression.

iii. $T(n) = 0.5^n T(n/2) + n^3 \log n$

The Master theorem can be applied to this recurrence relation where $a=0.5$, $b=2$, and $f(n) = n^3 \log n$. The runtime can be determined by comparing $f(n)$ with $n^{\log_b(a)} = n^{\log_2(0.5)} = n^{-1}$. Since $f(n) = \Theta(n^{1+\epsilon})$, where $\epsilon = 3 \log n / n$, we can see that $f(n)$ grows faster than n^{-1} but slower than n^1 . There, we can apply case 2 of the Master Theorem: If $f(n) = \Theta(n^c)$ for some constant c and $\log_b(a) = c$, then $T(n) = \Theta(n^c \log n)$.

In this case, $c = 1 + \epsilon$, so $T(n) = \Theta(n^{1+\epsilon} \log n) = \Theta(n^{1+3 \log n / n} \log n) = \Theta(n \log n)$.

Therefore, the runtime of the recurrence relation $T(n) = 0.5^n T(n/2) + n^3 \log n$ is $\Theta(n \log n)$.

iv. $T(n) = 4T(n/2) + n^2$

The Master theorem can be applied to this recurrence relation

$a = 4, b = 2$, and $f(n) = n^2$

The critical exponent is $\log_2(4) = 2$, which is equal to the exponent of $f(n)$.

Therefore, we have case 2 of the master theorem, which gives us:

$T(n) = \Theta(n^2 \log n)$.

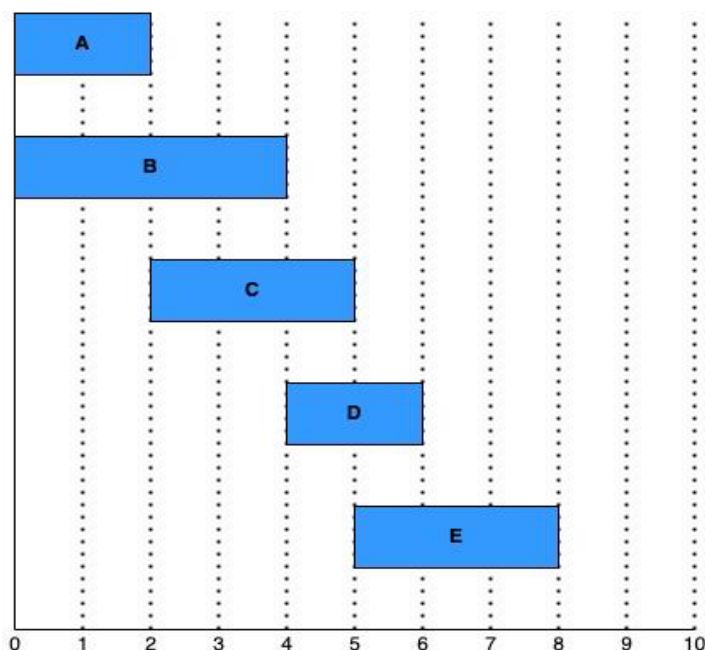
So the runtime of the algorithm is $\Theta(n^2 \log n)$.

v. $T(n) = n^2 T(n/3) + n$

The master theorem cannot be directly applied to this recurrence relation because it doesn't satisfy the conditions of the theorem.

Q4 (5 Points)

Given the five intervals below, and their associated values; select a subset of non-overlapping intervals with the maximum combined value. Use dynamic programming. Show your work.



Interval	Value
A	2
B	4
C	3
D	2
E	3

- To solve this problem using dynamic programming, we can define an array $dp[i]$ to represent the maximum combined value of the non-overlapping intervals that end at index i .
- We can initialize $dp[0]$ to the value of the first interval. Then, we can iterate through the remaining intervals and compute $dp[i]$ using the recurrence relation above. Finally, the maximum value in dp will be the answer.
- In this code, we first initialize the dp array and set the value of $dp[0]$ to the first interval. Then, we iterate through the remaining intervals using a for loop and compute $dp[i]$ using the recurrence relation above. To find the maximum value of $dp[j]$ for all $j < i$ that do not overlap with interval i , we iterate through all such j values using another for loop. We check whether interval j is non-overlapping with interval i by comparing their values, and we also check whether $dp[j]$ is greater than the current maximum value. Finally, we update $dp[i]$ to be the maximum value found plus the value of interval i . At the end, we print the maximum value in dp , which is the answer to the problem.

```
#Q4
intervals = [2, 4, 3, 2, 3]
n = len(intervals)

dp = [0] * n
dp[0] = intervals[0]

for i in range(1, n):
    max_value = 0
    for j in range(i):
        if intervals[j] <= intervals[i] and dp[j] > max_value:
            max_value = dp[j]
    dp[i] = max_value + intervals[i]

print(max(dp))
```

8

Q5 (5 Points)

Given the weights and values of the five items in the table below, select a subset of items with the maximum combined value that will fit in a knapsack with a weight limit, W , of 10. Use dynamic programming. Show your work.

Item _{i}	Value v_i	Weight w_i
1	5	5
2	3	1
3	2	3
4	4	4
5	1	3

Capacity of Knapsack = 10

- The basic idea of the dynamic programming approach is to create a table with dimensions (number of items + 1) x (maximum weight capacity + 1). We then fill in the table using a recursive formula that considers two cases:
 - The item cannot fit in the knapsack, so we skip it.
 - The item can fit in the knapsack, so we consider whether to include it or not.
- For each cell in the table, the value represents the maximum value that can be obtained using the first i items and a knapsack with weight capacity j .

```
#Q5
def knapsack(weights, values, W):
    n = len(weights)
    table = [[0 for j in range(W+1)] for i in range(n+1)]

    for i in range(1, n+1):
        for j in range(1, W+1):
            if weights[i-1] > j:
                table[i][j] = table[i-1][j]
            else:
                table[i][j] = max(table[i-1][j], values[i-1] + table[i-1][j-weights[i-1]])

    selected_items = []
    i = n
    j = W
    while i > 0 and j > 0:
        if table[i][j] != table[i-1][j]:
            selected_items.append(i-1)
            j -= weights[i-1]
            i -= 1

    return (table[n][W], selected_items[::-1])

weights = [5, 1, 3, 4, 3]
values = [5, 3, 2, 4, 1]
W = 10
max_value, selected_items = knapsack(weights, values, W)
print("Maximum value:", max_value)
```

Maximum value: 12

Q6 (10 Points)

We are given $T[1..n]$ of n unique integers that are sorted in increasing order, (used 1-based indexing)

- Give a divide-and-conquer algorithm that finds an index i such that $T[i] = i$ (if one exists) and runs in time $O(\log n)$.
- If $T[1] > 0$. Provide with a better algorithm that either computes an index i such that $T[i] = i$ or correctly reports that no such index exists. ($T[1] > 0$ i.e. first element greater than 0)

a. To find an index i such that $T[i] = i$ (if one exists) using a divide-and-conquer algorithm that runs in time $O(\log n)$, we can modify the binary search algorithm as follows:

- Set the left endpoint of the search interval to $L = 1$ and the right endpoint to $R = n$.
- While $L \leq R$ do the following:
 - Compute the middle index $m = \text{floor}((L + R)/2)$.
 - If $T[m] = m$, return m as the answer.
 - If $T[m] < m$, search the right half of the interval by setting $L = m + 1$.
 - If $T[m] > m$, search the left half of the interval by setting $R = m - 1$.
- If the search interval is exhausted and no index i such that $T[i] = i$ is found, return "no such index exists".
- Since each step reduces the size of the interval by half, the algorithm runs in $O(\log n)$ time.

b. If $T[1] > 0$, we can modify the above algorithm as follows:

- Set the left endpoint of the search interval to $L = \max(1, T[1])$ and the right endpoint to $R = n$.
- While $L \leq R$ do the following:
 - Compute the middle index $m = \text{floor}((L + R)/2)$.
 - If $T[m] = m$, return m as the answer.
 - If $T[m] < m$, search the right half of the interval by setting $L = \max(m + 1, T[m])$.
 - If $T[m] > m$, search the left half of the interval by setting $R = m - 1$.
- If the search interval is exhausted and no index i such that $T[i] = i$ is found, return "no such index exists".
- By setting the left endpoint of the search interval to $\max(1, T[1])$, we ensure that the search interval always

contains the element $T[i] = i$ (if one exists). The algorithm still runs in $O(\log n)$ time because each step reduces the size of the interval by half.

Q7 (10 Points)

Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within six miles of one of the base stations.

Give an efficient algorithm along with pseudo code that achieves this goal, using as few base stations as possible

- An efficient algorithm to solve this problem is the greedy algorithm.
- The key is to start with the leftmost house, place a base station at the furthest point that covers it (i.e. six miles to the right), then move to the next uncovered house and repeat the process until all houses are covered. Here's the pseudo code:
 - 1) Sort the house locations in increasing order.
 - 2) Initialize an empty list to store the base station locations.
 - 3) Set the current position to the leftmost house location.
 - 4) While there are still uncovered houses, do the following:
 - a) Find the furthest house to the right that is within six miles from the current position.
 - b) Add a base station at the location of the furthest house.
 - c) Set the current position to the furthest house that was covered.
 - 5) Return the list of base station locations.

Following is the code for the same:

```
#Q7
def place_base_stations(houses):
    # Step 1: Sort the house locations in increasing order.
    houses.sort()

    # Step 2: Initialize an empty list to store the base station locations.
    base_stations = []

    # Step 3: Set the current position to the leftmost house location.
    current_pos = houses[0]

    # Step 4: While there are still uncovered houses, do the following:
    while current_pos < houses[-1]:
        # Step 4a: Find the furthest house to the right that is within six miles from the current position.
        furthest_house = current_pos
        for house in houses:
            if house - furthest_house > 6:
                break
            furthest_house = house

        # Step 4b: Add a base station at the location of the furthest house.
        base_stations.append(furthest_house)

        # Step 4c: Set the current position to the furthest house that was covered.
        current_pos = furthest_house

    # Step 5: Return the list of base station locations.
    return base_stations
```

- Note that this algorithm has a time complexity of $O(n \log n)$, where n is the number of houses, due to the sorting step. However, it has a very good approximation ratio of $1 + \log_6(n)$, meaning that the number of base stations used by the algorithm is at most $1 + \log_6(n)$ times the min. number of base stations needed to cover all houses.

Q8 (10 Points)

Consider an n -node complete binary tree T , where $n = 2^d - 1$ for some d . Each node v of T is labeled with a real number x_v . You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a local minimum if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

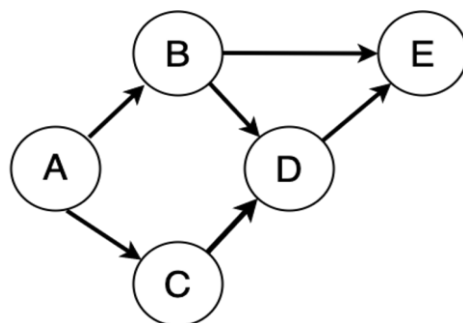
You are given such a complete binary tree T , but the labeling is only specified in the following implicit way: for each node v , you can determine the value x_v by probing the node v . Show how to find a local minimum of T using only $O(\log n)$ probes to the nodes of T .

- To find a local minimum in the given complete binary tree T , we can start at the root and recursively move towards the smaller child until we find a node that is a local minimum.
- This approach takes $O(n)$ probes, which is not efficient.
- Following can be done to get an efficient result:-

1. Start at the root of the binary tree T.
 2. Probe the values of its two children, v_left and v_right .
 3. If $xv < xv_left$ and $xv < xv_right$, then v is a local minimum and we can stop.
 4. Otherwise, if $xv_left < xv_right$, move to the left child v_left and repeat from step 2 on the subtree rooted at v_left .
 5. Otherwise, move to the right child v_right and repeat from step 2 on the subtree rooted at v_right .
- Each iteration reduces the search space to one-half, so the number of probes is $O(\log n)$.
 - The algorithm guarantees that we will find a local minimum because we always move towards the smaller child, and the tree is complete, so each non-leaf node has exactly two children.
 - The running time of the algorithm is $O(\log n)$.
 - Following is the code: -

```
#Q8
def find_local_minimum(T):
    # Start at the root
    v = T
    # Loop until we find a local minimum
    while True:
        # Probe the values of the left and right children
        xv = probe(v)
        xv_left = probe(v.left)
        xv_right = probe(v.right)
        # Check if v is a local minimum
        if xv < xv_left and xv < xv_right:
            return v
        # Move towards the smaller child
        if xv_left < xv_right:
            v = v.left
        else:
            v = v.right
```

Q9 (10 Points) Given the DAG below,



A. (5 points) Express the directed graph above as:

A. An adjacency list

Q 9) A) Adjacency list

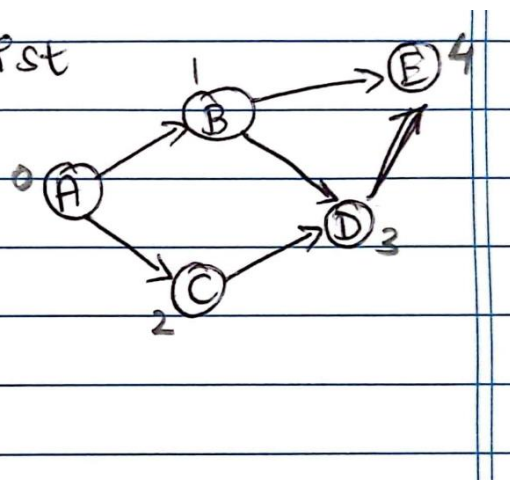
A → B C

B → D E

C → D

D → E

E .



The code for adjacency list can be given as:-

```
#Q9A
# Function to add edges
def addEdge(adj, u, v):
    adj[u].append(v)
# Function to print adjacency list
def adjacencylist(adj, V):
    for i in range(0, V):
        print(i, "->", end="")
        for x in adj[i]:
            print(x, " ", end="")
        print()
# Function to initialize the adjacency list
def initGraph(V, edges, noOfEdges):
    adj = [0]* 5
    for i in range(0, len(adj)):
        adj[i] = []
    # Traverse edges array and make edges
    for i in range(0, noOfEdges) :
        addEdge(adj, edges[i][0], edges[i][1])
    # Function Call to print adjacency list
    adjacencylist(adj, V)
V = 5
edges = [[0, 1 ], [0, 2 ], [1, 3 ], [1, 4], [2, 3], [3, 4]]
noOfEdges = 6;

initGraph(V, edges, noOfEdges)
```

```
0 -> 1 2
1 -> 3 4
2 -> 3
3 -> 4
4 ->
```

B. An adjacency matrix

Q.9) B) Adjacency matrix

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

The code for the adjacency matrix can be given as followed:-



#Q9B

Graph via adjacency list

graph = {

"A": ["B", "C"],

"B": ["D", "E"],

"C": ["D"],

"D": ["E"],

"E": [],

}

keys = sorted(graph.keys())

size = len(keys)

matrix = [[0] * size for i in range(size)]

We iterate over the key:value entries in the dictionary first,

then we iterate over the elements within the value

for a, b in [(keys.index(a), keys.index(b)) for a, row in graph.items() for b in row]:

Use 1 to represent if there's an edge

Use 2 to represent when node meets itself in the matrix (A -> A)

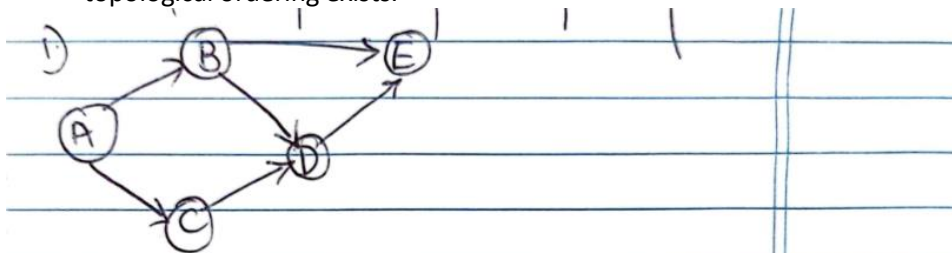
matrix[a][b] = 2 if (a == b) else 1

print(matrix)

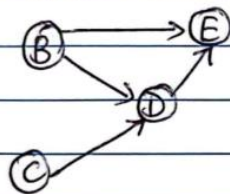
[[0, 1, 1, 0, 0], [0, 0, 0, 1, 1], [0, 0, 0, 1, 0], [0, 0, 0, 0, 1], [0, 0, 0, 0, 0]]

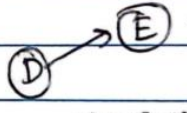
B.(5 points) Can the directed graph be topologically sorted? If so, produce a topological sort for the graph. Show your work.

- The graph can be topologically sorted. This can be done with the following steps:
 - Identify a node with no incoming edges.
 - Add that node to the ordering.
 - Remove it from the graph.
 - Repeat.
- Then, keep looping until there aren't any more nodes with indegree zero. This could happen for two reasons:
 - There are no nodes left. We've taken all of them out of the graph and added them to the topological ordering.
 - There are some nodes left, but they all have incoming edges. This means the graph has a cycle, and no topological ordering exists.

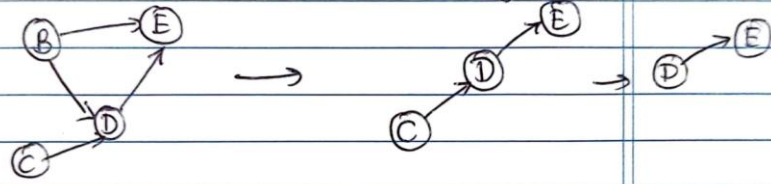


2) Remove node A as it has no incoming edge.



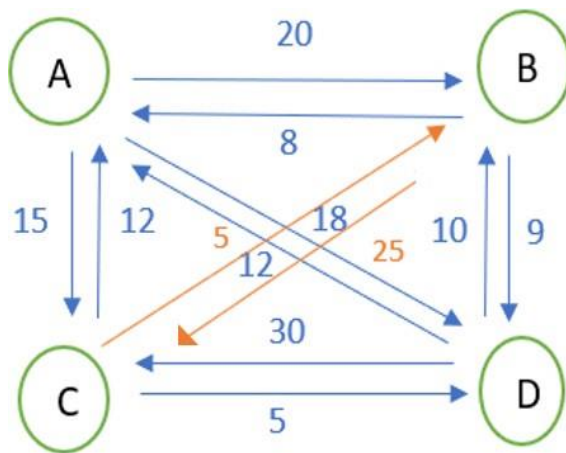
case i:] Remove C node.  then remove D
 $\therefore A - C - B - D - E$

case 2] Remove B from the graph.



∴ The resulting path would be
A - B - C - D - E

Q10 (10 Points) 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?



- The problem can be solved using many ways, this is a travelling salesman problem which is a permutation problem. This problem can be solved using the brute force approach as well. The total ways to solve permutation problems can be $n!$ ways.
- The problem can be solved by generating all the possible permutations or the routes and then find the one with the shortest distance.
- First choose a city which will be the starting point, here we can take A
- Then, find all the possible routes using permutations. For n cities, the permutation will be $(n-1)!$
- Finally, calculate the cost of each route and return the one which has the minimum cost.

Q.10)

	A	B	C	D
A	0	20	15	18
B	20	0	25	9
C	15	25	0	5
D	18	9	5	0

min. (62, 38, 61) ∴ 38

```

#Q10
from sys import maxsize
from itertools import permutations
n = 4

def travelSalesman (graph, s):
    vertex = []
    for i in range(n):
        if i != s:
            vertex.append(i)

    minimum_path = maxsize
    next_perm = permutations(vertex)
    for i in next_perm:
        current_path = 0
        a = s
        for j in i:
            current_path += graph[a][j]
            a = j
        current_path += graph[a][s]
        minimum_path = min(minimum_path, current_path)

    return minimum_path

if __name__ == "__main__":
    graph = [[0,20,15,18], [8,0,25,9],[12,5,0,5],[12,10,30,0]]
    s = 0
    print(travelSalesman(graph,s))

```

38

Q11 (20 Points)

Download the python code and read the instructions at the UC Berkeley The Pac-Man Projects

http://inst.eecs.berkeley.edu/~cs188/pacman/project_overview.html

Specifically download Project 1: Search in Pacman

<http://inst.eecs.berkeley.edu/~cs188/pacman/search.html>

A. (10 points) Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states. You can do this in code or by illustrating how depth-first search would work with a Pac-Man graph.

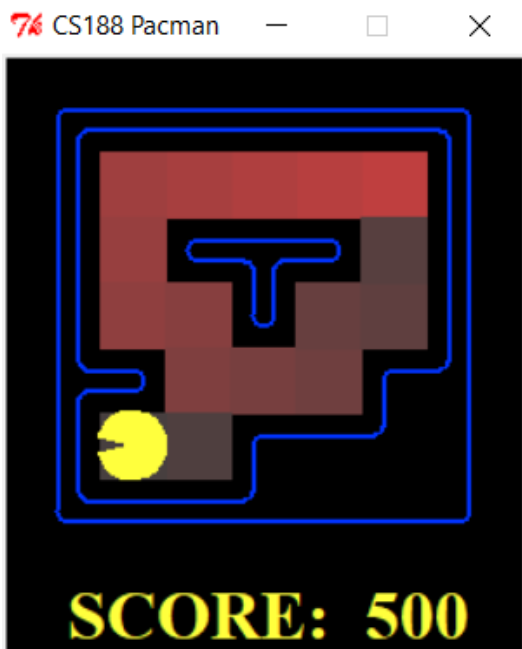
B. (10 points) Implement A* graph search in the empty function `starSearch` in `search.py`. A* takes a heuristic function as an argument. What heuristic function makes sense for a PacMan maze? You can do this in code or by illustrating how depth-first search would work with a Pac-Man graph.

Helpful code for BFS and A*

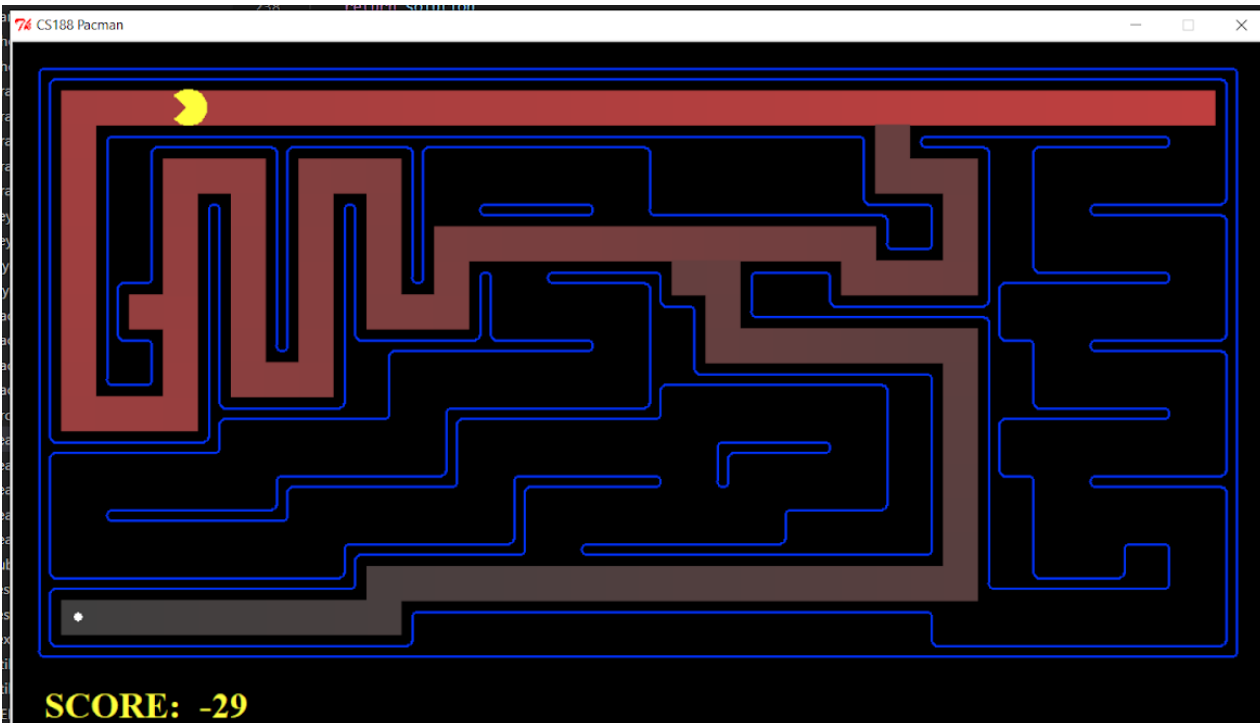
<http://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/><https://github.com/adlawson/bfs.py>

<http://code.activestate.com/recipes/576723-dfs-and-bfs-graph-traversal/>

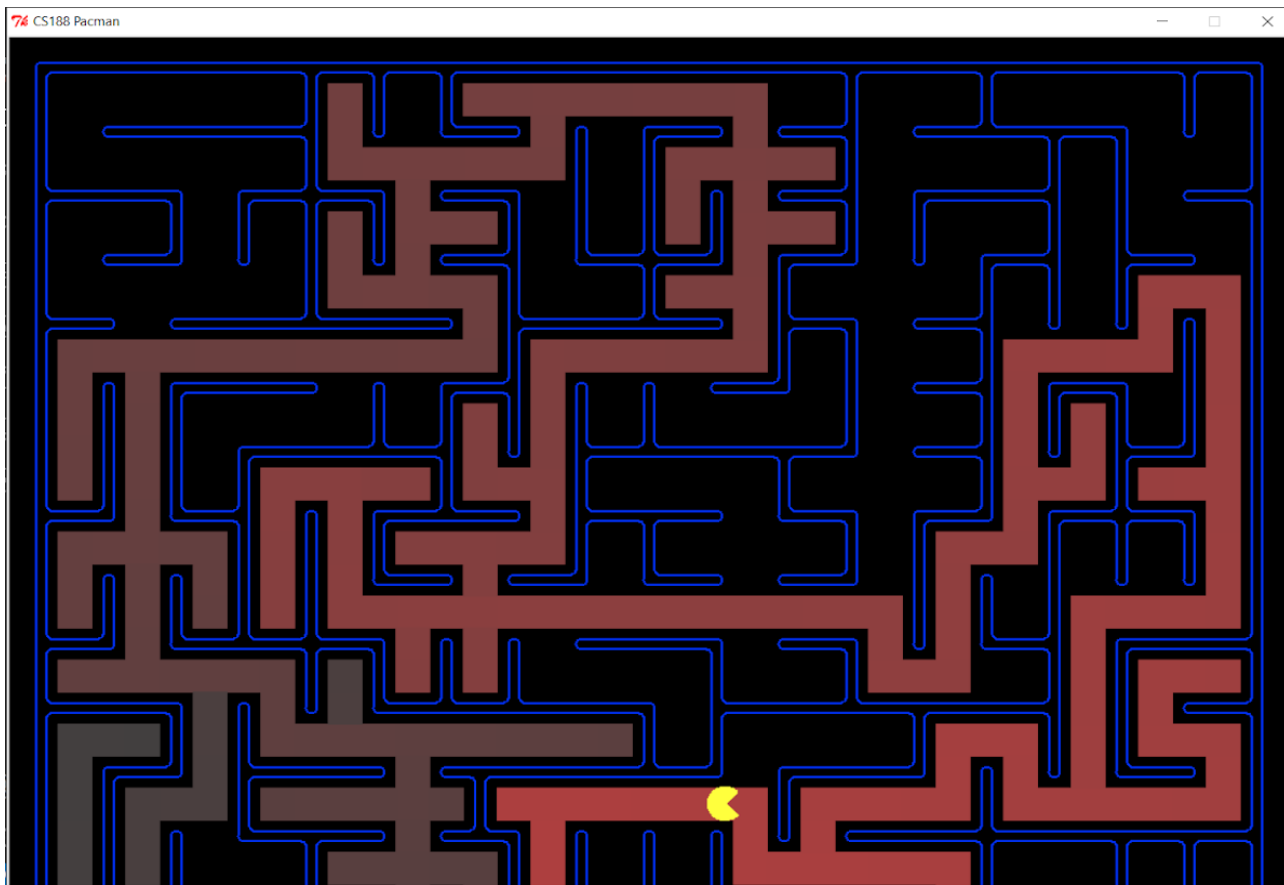
<http://code.activestate.com/recipes/577519-a-star-shortest-path-algorithm/>



python pacman.py --layout tinyMaze -p SearchAgent -a fn=depthFirstSearch

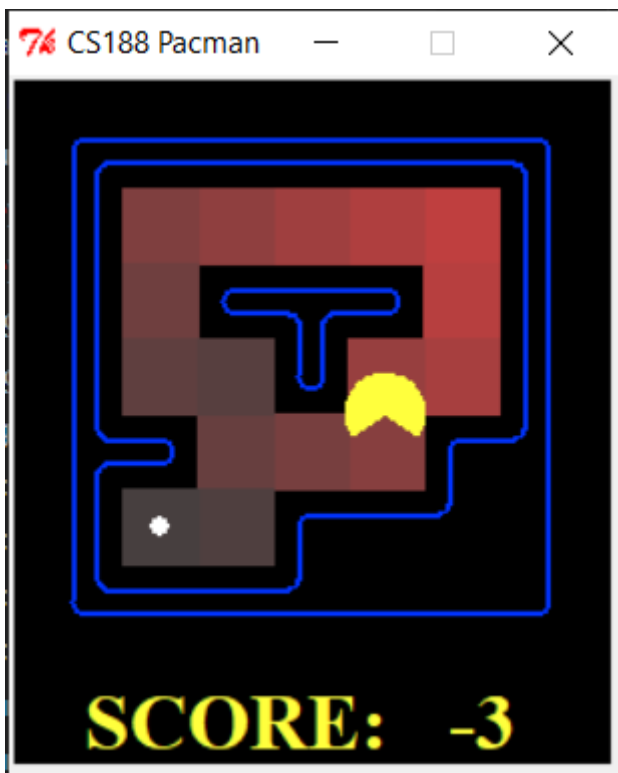


python pacman.py --layout MediumMaze -p SearchAgent -a fn=depthFirstSearch

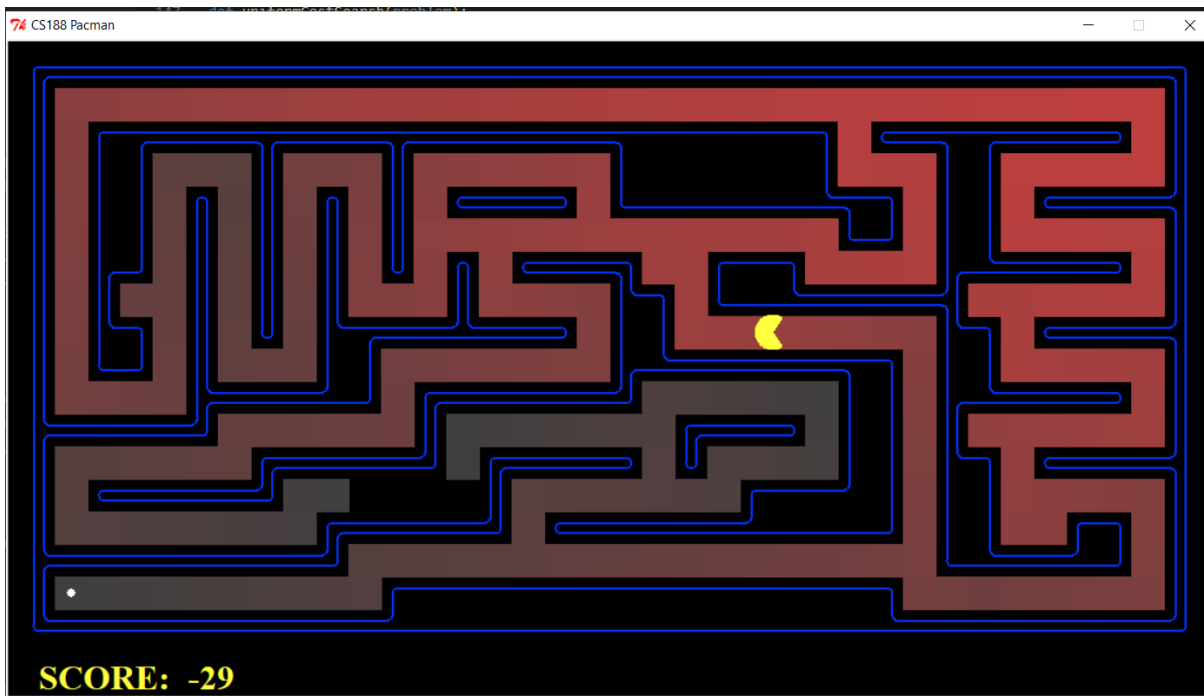


`python pacman.py --layout bigMaze -p SearchAgent -a fn=depthFirstSearch`

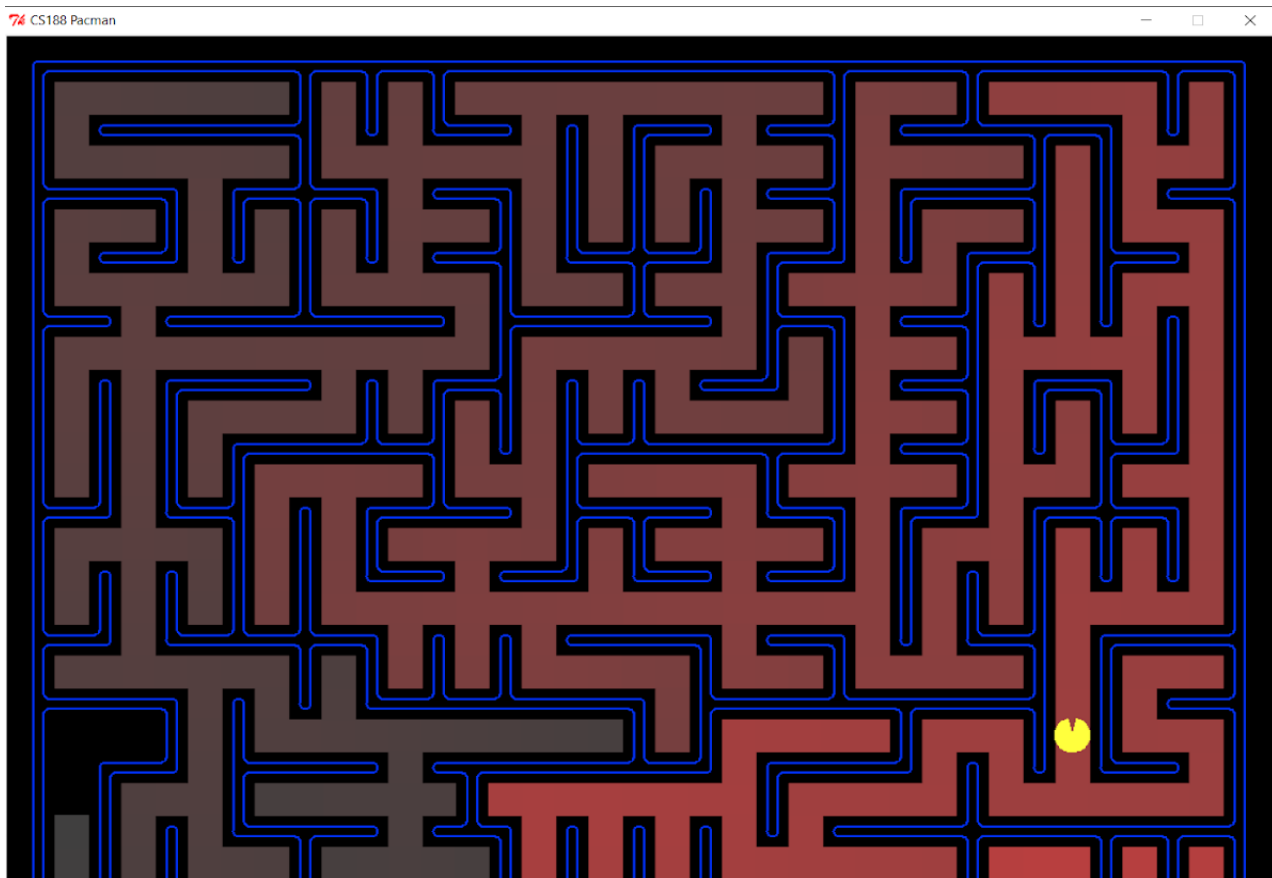
B)



`python pacman.py --layout tinyMaze -p SearchAgent -a fn=aStarSearch`



```
python pacman.py --layout mediumMaze -p SearchAgent -a fn=aStarSearch
```



```
python pacman.py --layout bigMaze -p SearchAgent -a fn=aStarSearch
```