INFO 6205 –Assignment 2

Student Name: Yuetong Guo
Professor: Nik Bear Brown

Q1(5 points)

Explain divide and conquer theorem with code and example.

Ans:

The divide and conquer theorem is a powerful algorithmic technique that involves breaking a problem into smaller sub-problems, solving them recursively, and then combining the solutions of the sub-problems to solve the original problem.
I will illustrate the most common use of the divide and conquer algorithm - find the position of a target number in a sorted array of integers from 1 to 100 in below pseudocode:

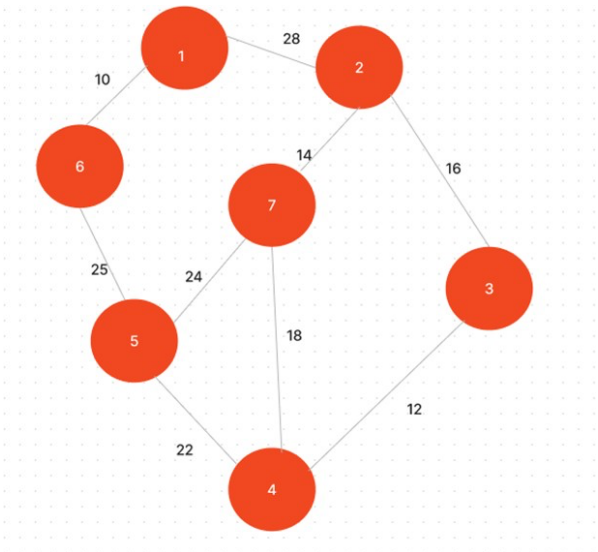
1. Let `low` be the index of the first element in the array, and `high` be the index of the last element in the array.
2. Repeat while `low` is less than or equal to `high`:
    a. Calculate the middle index as `mid = (low + high) // 2`.
    b. If the middle element is equal to the target number, return `mid`.
    c. If the middle element is greater than the target number, update `high = mid - 1`.
    d. If the middle element is less than the target number, update `low = mid + 1`.
3. If the target number is not found, return -1.

This approach works by dividing the problem in half at each step, effectively reducing the number of elements to search by half at each step. This approach optimizes the worst-case time complexity to O(n log n).

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?



Ans:

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree of a connected, weighted graph. The Kruskal's algorithm pseudocode are as below:

A: Create a forest T (a set of trees) containing all vertices in the graph as separate trees

B: Create a set S containing all the edges in the graph

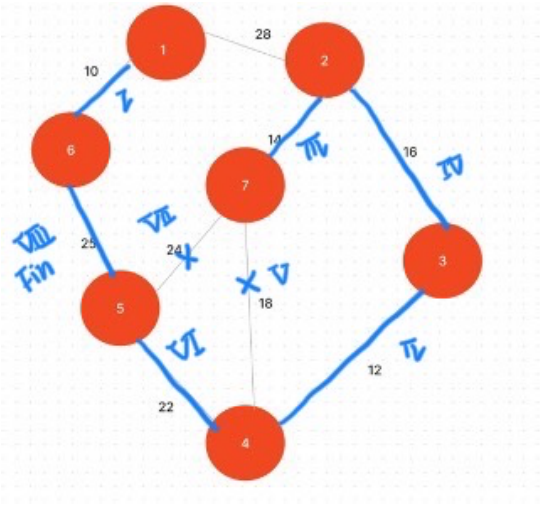
C: Sort edges by weight

While S is nonempty, and T is not yet spanning:

I: remove an edge with minimum weight from S

II: if that edge connects two different trees, then add it to the forest, combining two trees into a single tree, otherwise discard that edge.

III. the algorithm stops when T contains only one tree, which means that it is now a single spanning tree of the graph.



From this question, the steps would be:

I: Connect 1-6 (10)

II: Connect 3-4 (12)

III: Connect 2-7 (14)

IV. Connect 2-3 (16)

V. Skip 7-4 (18) (Forms cycle)

VI. Connect 5-4 (22)

VII. Skip 5-7 (24) (Forms cycle)

VIII. Connect 5-6 (25) (Forms cycle)

Stop. MST formed (6 edges, 7 vertices)

MST = {1-6, 6-5, 5-4, 4-3, 4-2, 2-7}

Kruskal's algorithm is O(E log V) time, where E is the set of edges and V is the set of vertices.

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.

Ans:

Master theorem provides a solution in asymptotic terms (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \text{ where } a \geq 1, \text{ and } b > 1,$$

Then $T(n)$ can be classified as one of four forms to determine asymptotic bounds as follows:

A. $n^{\log_b a}$ is polynomially larger than $f(n) \Rightarrow T(n) = \Theta(n^{\log_b a})$

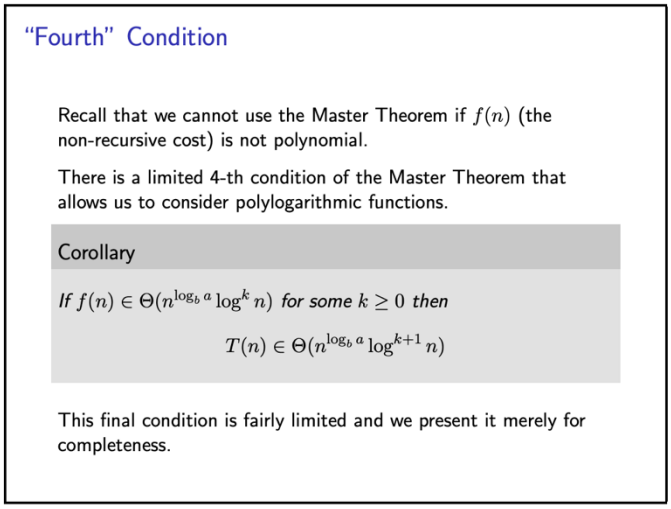
B. $f(n)$ and $n^{\log_b a}$ are same size $\Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \lg n)$

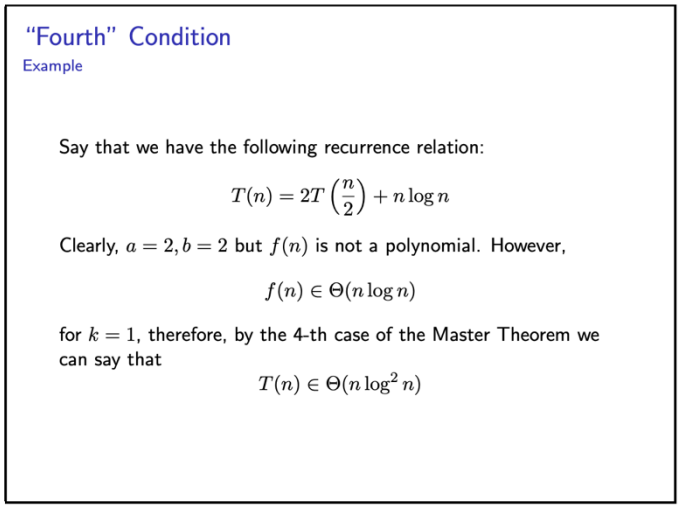
C. $f(n)$ is polynomially larger than $n^{\log_b a} \Rightarrow T(n) = \Theta(f(n))$

D. The master theorem does not apply.

Follow above rules, the solutions for each question are as below:

i. Although it can not directly conform to the above rules (or you can sat it conform to the case 2) according to the following extra rule of Master Theorem, this question can still be applied to it:





Q: T(n)=4T(n/4) + n log n
A=4, B=4, $\log_b a = k = 1$, f(n) = n log n

P=1 since f(n)=n log n=n $\log^1$ n

Therefore $\Theta(n^{\log_b a} * \log^{p+1} n) = \Theta(n \log^2 n)$

ii.

Q: $T(n) = 4T(n/3) + n^{0.33}$
A=4, B=3
f(n) = O(n^$\log_3 4 - \epsilon$) for some $\epsilon > 0$, we have T(n) = Θ(n^$\log_3 4$) = Θ(n^1.2618...).
Therefore, T(n) = Θ(n^$\log_3 4$) = Θ(n^1.2618...)

iii.

Q: $T(n) = 0.5^n T(n/2) + n^3 \log n$
The Master Theorem does not apply to this recurrence, since A is a function of n.

iv.

Q: $T(n)= 4T(n/2) + n^2$

This recurrence falls under Case 1 of the Master Theorem, with a = 4, b = 2, and f(n) = n^2.
Since f(n) = Θ(n^c) for c = 2, which is equal to log_b(a) = log_2(4) = 2.
Therefore, T(n) = Θ(n^2 log n)

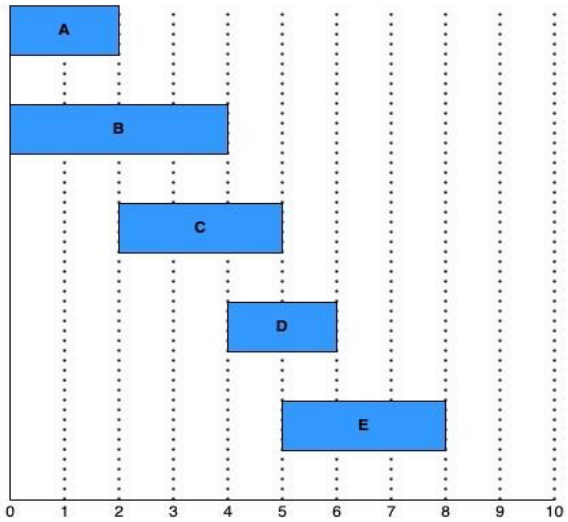
v.

Q: $T(n)= n^2T(n/3) + n$

The Master Theorem does not apply to this recurrence, since A is a function of n.

Q4 (5 Points)

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



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

Ans:

To gain a better understanding of the question, I will illustrate the information and process through diagrams. Further explanation was provided after two pictures.

Interval	Value	Previous	Max
A	2	n/a	Max(2+0, 0) = 2
B	4	n/a	Max(4+0, 2) = 4
C	3	A	Max(4, 3+2) = 5
D	2	B	Max(5, 2+4) = 6
E	3	C	Max(6, 3+5) = 8

Interval	Trace(i)	S
E	3+5 =8, jump to C	{E}
D	6 < 8	
C	3+2=5, jump to A	{E, C}
B	jump to A	
A	2=2	{E, C, A}

Using the intervals A through E, we can apply these steps as follows:

- Step 1. Sort the intervals by their end points in ascending order:
  - A (0, 2, 2)
  - B (0, 4, 4)
  - C (2, 5, 3)
  - D (4, 6, 2)
  - E (5, 8, 3)

- Step 2. Define an array OPT of size n+1, where OPT[i] represents the maximum combined value of a subset of non-overlapping intervals ending at the i-th interval: OPT = [0, 2, 4, 5, 6, 8]

- Step 3. Initialize OPT[0] = 0: OPT = [0, 0, 0, 0, 0, 0]

- Step 4. For each interval j from 1 to n, compute OPT[j] as follows:
  Find the index i of the last interval that does not overlap with j:
  For interval A (j=1), i=0 (no intervals before A)
  For interval B (j=2), i=0 (no intervals before B)
  For interval C (j=3), i=1 (interval A ends before C starts)
  For interval D (j=4), i=2 (interval C ends before D starts)
  For interval E (j=5), i=1 (interval A ends before E starts)
  Compute the value Vj of the combined subset of intervals that ends with interval j:
  For interval A (j=1), Vj=2
  For interval B (j=2), Vj=4
  For interval C (j=3), Vj=5 (A and C)
  For interval D (j=4), Vj=6 (B and D)
  For interval E (j=5), Vj=8 (A, C, and E)
  Set OPT[j] to the maximum value of either OPT[j-1] or Vj:
  For interval A (j=1), OPT[1]=2
  For interval B (j=2), OPT[2]=4
  For interval C (j=3), OPT[3]=5
  For interval D (j=4), OPT[4]=6
  For interval E (j=5), OPT[5]=8

- Step 5. The maximum combined value of a subset of non-overlapping intervals is given by OPT[n]=8.

Q5 (10 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.

Capacity of Knapsack = 10

$Item_i$	Value v_i	Weight w_i
1	5	5
2	3	1
3	2	3
4	4	4
5	1	3

Ans:

We could use a 2-D dynamic programming algorithm to solve this problem.

The pseudocode is as below:

```
knapsack(values, weights, capacity):
  // values is an array of item values, weights is an array of item weights,
  // and capacity is the maximum weight the knapsack can hold

  n = length(values)
  // Initialize a 2D array to store the maximum value that can be obtained
  max_value = [[0]*(capacity+1) for i in range(n+1)]

  // Fill the array using dynamic programming
  for i in range(1, n+1):
    for j in range(1, capacity+1):
      // If the current item's weight is greater than the current weight limit, skip it
      if weights[i-1] > j:
        max_value[i][j] = max_value[i-1][j]
      else:
        // Select the item and add its value to the maximum value that can be obtained
        // with the remaining weight
        include = values[i-1] + max_value[i-1][j-weights[i-1]]
        // Don't select the item
        exclude = max_value[i-1][j]
        // Choose the option with the maximum value
        max_value[i][j] = max(include, exclude)

  // Traverse the 2D array to find the items that were selected
  items = []
  j = capacity
  for i in range(n, 0, -1):
    if max_value[i][j] != max_value[i-1][j]:
      items.append(i-1)
```

```
        j -= weights[i-1]

    // Reverse the list of items to get the correct order
    items.reverse()

    // Return the maximum value and the selected items
    return max_value[n][capacity], items
```

right is Item.i below is W.i	1	2	3	4	5
1	0	3	3	3	3
2	0	3	3	3	3
3	0	3	3	3	3
4	0	3	5	5	5
5	5	5	5	7	7
6	5	8	8	8	8
7	5	8	8	8	8
8	5	8	8	9	9
9	5	8	10	10	10
10	5	8	10	12	12

To use this algorithm to solve the specific problem given in the previous question, you can call the knapsack function with the values array [5, 3, 2, 4, 1], the weights array [5, 1, 3, 4, 2], and the capacity 10.

The function will return the maximum value that can be obtained, which is 12, and the indices of the selected items, which are [0, 1, 3] (corresponding to items 1, 2, and 4).

Q6 (10 Points)

We are given T of n unique integers that are sorted in increasing order,

a. 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).
b. 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.

Ans:

A:

To find an index i such that T[i] = i in a sorted array T of n unique integers, we can use a divide-and-conquer algorithm that takes advantage of the fact that the array is sorted. We start by comparing the value of T[mid] with mid, where mid is the middle index of the array.

If T[mid] is greater than mid, it means that the desired index i must be in the left half of the array because T[i] is increasing and T[i] cannot be greater than i. Therefore, we can recursively search the left half of the array.

Similarly, if T[mid] is less than mid, it means that the desired index i must be in the right half of the array because T[i] is increasing and T[i] cannot be less than i. Therefore, we can recursively search the right half of the array.

If T[mid] is equal to mid, then we have found the desired index i and we can return mid.

We can repeat this process until we find the desired index i or we determine that it does not exist. Since each iteration cuts the size of the array in half, the running time of this algorithm is O(log n), making it a very efficient way to solve this problem.

Overall, this algorithm is similar to the classic "guess the number" game where you are trying to find a target number between 1 and 100, but with the added constraint that the target number must be equal to its index in the array.

Here is the pseudocode for the algorithm:

```
def findIndex(T, start, end):
    if start > end:
        return -1  # desired index not found

    mid = (start + end) // 2

    if T[mid] == mid:
        return mid

    if T[mid] > mid:
        return findIndex(T, start, mid - 1)

    if T[mid] < mid:
        return findIndex(T, mid + 1, end)
```

B:

here's an algorithm that computes an index i such that T[i] = i or correctly reports that no such index exists, even if T[1] > 0:

Set left = 1 and right = n.
While left <= right, do the following:
    a. Set mid = (left + right) // 2.
    b. If T[mid] == mid, return mid.
    c. If T[mid] > mid, set right = mid - 1.
    d. If T[mid] < mid, set left = mid + 1.
Return -1 to indicate that no such index exists.

The only difference between this algorithm and the previous one is that we start the search from index 1 instead of index 0. This means that we can no longer use the fact that T[i] is increasing, since T[1] might be greater than 0. Instead, we use the fact that the array is sorted and we adjust the search accordingly.

The algorithm works as follows: we start by setting the left and right boundaries to the first and last indices of the array, respectively. We then enter a loop that continues until the left and right boundaries overlap. In each iteration of the loop, we compute the middle index of the current range (step 2a). If T[mid] equals mid, we have found the desired index i and we return it (step 2b). If T[mid] is greater than mid, the desired index i must be in the left half of the range, so we update the right boundary accordingly (step 2c). If T[mid] is less than mid, the desired index i must be in the right half of the range, so we update the left boundary accordingly (step 2d). If we exit the loop without finding the desired index i, we return -1 to indicate that no such index exists (step 3). This algorithm has the same time complexity as the previous one: O(log n).

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.

Ans:

We could directly use greedy algorithm to solve this problem.

The greedy algorithm is a simple, intuitive algorithmic strategy that makes the locally optimal choice at each step in the hope of finding a globally optimal solution. It is a type of heuristic algorithm that seeks to solve a problem by making the best possible choice at each step without considering the overall effect of the choices made.

To solve this problem, we'll follow the following algorithm design:

1. We start by sorting the distances of all the houses in increasing order from the northern to the southern endpoint of the road. This allows us to place the base stations in a sequential and efficient manner.
2. Then we initialize an empty list to store the locations of the base stations that we will place.
3. We place the first base station 6 miles south from the location of the first house.
4. We move southward along the road, checking the distance between the previously placed base station and each house.
5. If we find a house that is more than six miles away from the previously placed base station, we place a new base station 6 miles south of the house and repeat step 4.
6. We repeat step 4 and step 5 until all houses have been covered.
7. Finally, we return the list of base station locations that we have placed.

Below is the pseudocode implementation of the algorithm:

```
function place_base_stations(houses):
    // Sort houses by distance from north to south
    sorted_houses = sort_by_distance(houses)

    // Initialize an empty list for base station locations
    base_stations = []

    // Place the first base station 6 miles south of the first house
    current_house = sorted_houses[0]
    current_station_location = current_house.location - 6
    base_stations.append(current_station_location)

    // Move southward along the road, checking the distance
    for i = 1 to len(sorted_houses)-1:
        current_house = sorted_houses[i]
        distance_to_last_station = current_house.location - current_station_location

        // If the current house is more than 6 miles away, place a new station 6 miles south of it
```

```
        if distance_to_last_station > 6:
            current_station_location = current_house.location - 6
            base_stations.append(current_station_location)

    // Return the list of base station locations
    return base_stations
```

This approach typically results in a satisfactory solution, but it may not always produce the optimal solution. To achieve the most effective outcome, we may need to refine the greedy algorithm. You can find more information about the specific issues that arise from this in the following link, which offers a detailed explanation.

* http://www.cs.toronto.edu/~jepson/csc373/tutNotes/cellTower.pdf

Q8 (10 Points)

Consider an n-node complete binary tree T, where $n = 2d - 1$ for some d. Each node v of T is labeled with a real number xv. 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 xv is less than the label xw 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 xv 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.

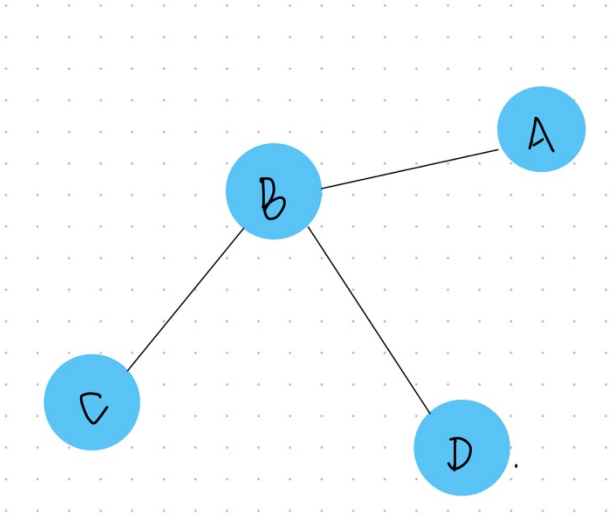
Ans:

Given the information about the complete binary tree T with n nodes and the requirement to find a local minimum using only O(log n) probes, we can naturally think of using a divide-and-conquer strategy.

Starting at the root of T, we compare the values of its two children and the root. If the root is the smallest, then the root is a local minimum node. If one of the children has a smaller value than the root, we recursively search in that subtree. Otherwise, we recursively search in the other subtree. We continue this process until we find a minimum node. If we cannot find any local minimum node during the whole process, we would find at least one local minimum node until we reach a leaf node.

Below is proof of the property that the structure of a complete binary tree has a local minimum node:

In the graph on the right, if A is smaller than B, then A is the minimum node. If A is bigger than B and B is bigger than C or D, then C or D would be the minimum node. If A is bigger than B and B is smaller than both C and D, then B would be the minimum node.



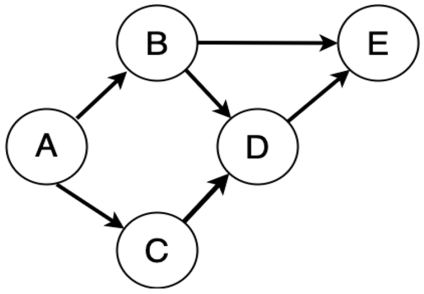
Below is the pseudocode implementation of the algorithm:

```
procedure FindLocalMinimum(T)
    # Start at the root node of T
    v = T.root
```

```
# Continue searching until a local minimum is found
while True:
    # Probe the left and right child of v
    left_child = probe(v.left)
    right_child = probe(v.right)

    # If both children have larger values, v is a local minimum
    if left_child > v.value and right_child > v.value:
        return v

    # If the left child has a smaller value, search the left subtree
    if left_child < right_child:
        v = v.left

    # If the right child has a smaller value, search the right subtree
    else:
        v = v.right
```

Q9 (10 Points)

Given the DAG below,



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

A. An adjacency list

B. An adjacency matrix

Ans:

Asjacency List	
A	B -> C
B	E -> D
C	D
D	E
E	

Asjacency 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

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

Ans:

Yes, this directed graph can be topologically sorted because it does not contain any cycles.
One possible topological sort for this graph is: {A, C, B, D, E}
1. To obtain this topological sort, we can follow the steps of a topological sort algorithm:
2. Find all nodes with no incoming edges. In this case, we start with A and C.

3. Visit one of these nodes, say A, and add it to the sorted list.
4. Remove A from the graph and all its outgoing edges.
5. Repeat steps 1-3 until all nodes have been added to the sorted list.
6. The order in which we visit the nodes with no incoming edges does not matter, so we can also start with C instead of A and obtain the same topological sort.

The pseudocode of above algorithm is:

```
graph = {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['D'], 'D': ['E'], 'E': []}
in_degree = {'A': 0, 'B': 1, 'C': 1, 'D': 2, 'E': 2}
zero_in_degree = ['A', 'C']
sorted_list = []

while zero_in_degree:
    current = zero_in_degree.pop(0)
    sorted_list.append(current)
    for next in graph[current]:
        in_degree[next] -= 1
        if in_degree[next] == 0:
            zero_in_degree.append(next)

if len(sorted_list) < len(graph):
    print("The graph contains a cycle and cannot be topologically sorted.")
else:
    print(sorted_list)
```

And the step-by-step process are as follows:

Step 1: Nodes with no incoming edges: A, C

Step 2: Visit A, add it to the sorted list: A

Step 3: Remove A and its outgoing edges: B, C, D

Step 4: Nodes with no incoming edges: C

Step 5: Visit C, add it to the sorted list: A, C

Step 6: Remove C and its outgoing edges: D

Step 7: Nodes with no incoming edges: B, D

Step 8: Visit B, add it to the sorted list: A, C, B

Step 9: Remove B and its outgoing edges: D, E

Step 10: Visit D, add it to the sorted list: A, C, B, D

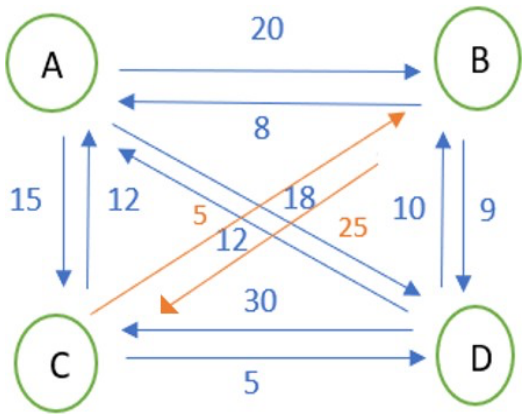
Step 11: Remove D and its outgoing edges: E

Step 12: Nodes with no incoming edges: E

Step 13: Visit E, add it to the sorted list: A, C, B, 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?



Ans:

For the travelling salesman problem, dynamic programming algorithm would be a great way to solve it.

1. Firstly, we start with a base case where we have only one city in the set and then gradually build up the solution by adding one city at a time until we have all the cities in the set.

2. Then considering the case where we have a set of two cities at a time and finding the minimum cost path that starts from the starting city A, visits the cities in the subset and ends at one of them, and then returns to the starting city A. We are using the previously calculated values for smaller subsets to calculate the values for larger subsets.

3. Finally, we use these values to find the optimal tour route that visits all the cities exactly once and returns to the starting city A.

Distance	A	B	C	D
A	0	20	15	18
B	8	0	25	9
C	12	5	0	5
D	12	10	30	0

From the graph in the question, we generate a distance matrix. And we choose A as start point, which is also end point. And we label A, B, C, D as 1, 2, 3, 4.

1. Considering how to travel back to the end city A (T means the path)

f(2, P) = D21=Distance from B to A = 8,
f(3, P) = D31=Distance from C to A = 12,
f(4, P) = D41=Distance from D to A = 12,

2.K = 1, consider set of one element:

Set {2}: f(3, {2}) = D32 + f(2, P) = D32 + D21 = 5 + 8 = 13
f(4,{2}) = D42 + f(2, P) = D42 + D21 = 10 + 8 = 18
Set {3}: f(2,{3}) = D23 + f(3, P) = D23 + D31 = 25+ 12 = 37

f(4,{3}) = D43 + f(3, P) = D43 + D31 = 30 + 12 = 42
Set {4}: f(2,{4}) = D24 + f(4,P) = D24 + D41 = 9 + 12 = 21
f(3,{4}) = D34 + f(4,P) = D34 + D41 = 5 + 12 = 17

3. K = 2 , consider set of two element :

Set {3,4}: f{2,{3,4}} = min {D23 + f(3,{4}) , D24 + f(4,{3})} = min { 42, 51} = 42
Set {2,4}: f{3,{2,4}} = min {D32 + f(2,{4}), D34 + f(4,{2})} = min { 26, 23} = 23
Set {2,3}: f(4,{2,3}) = min {D42 + f(2,{3}), D43 + f(3,{2})} = min { 47, 43} = 43

4. Length of an optimal tour,

F { 1, {2,3,4}} = min{ D12 + D(2,{3,4}), D13 + f(3,{2,4}), D14 + f(4,{2,3})} = min { 62, 38, 61 } = 38

The minimum cost path is 38. The optimal tour route is, A -> C -> D -> B -> A .

k/s	{2}	{3}	{4}	{3, 4}	{2, 4}	{2, 3}	{2, 3, 4}
2	n/a	25+D31	10+D41	42	23	n/a	n/a
3	5+D21	n/a	5+D41	30	38	42	38+D24
4	10+D21	12+D31	n/a	17	21+D32	43	38+D23
F	62+D24	38+D31	61+D32	42	23	43	38

Here's a table that shows the values of f(k, S) at each step of the dynamic programming algorithm for the traveling salesman problem with the distance values you provided earlier. In this table, k represents the current city and S represents the set of cities that have not yet been visited. The value f(k, S) represents the minimum distance required to visit all the cities in S exactly once, starting at city k and ending at city A. Note that the diagonal values (where k is in S) are not defined, as it is not possible to visit a city that has already been visited. Also, the values of f are only calculated for non-empty sets S.

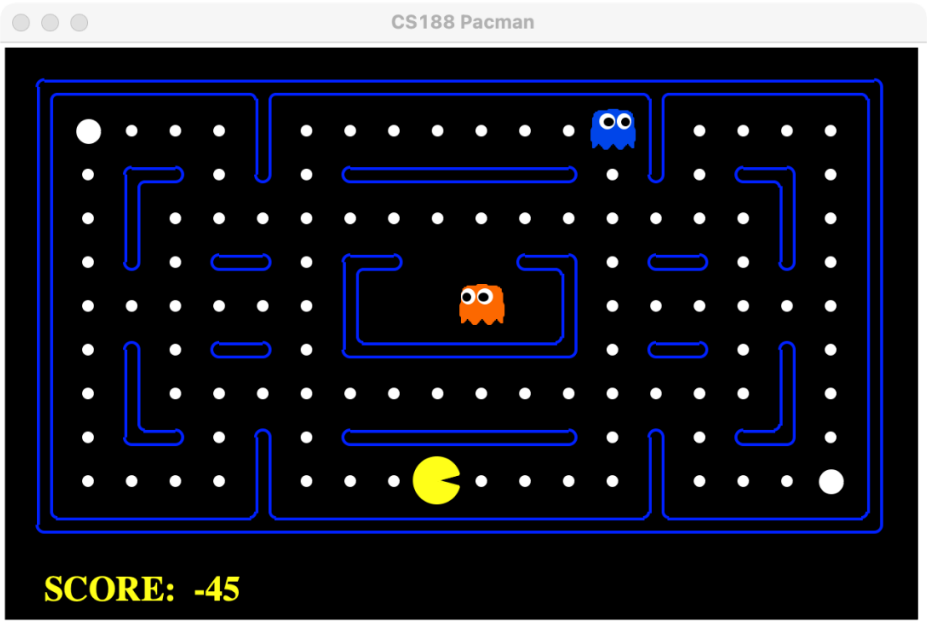
Q11 (20 Points)

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 aStarSearch 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/

Ans:



A. The runnable code is in the search file named search.py, below is the screenshot of it.

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"
    temp = problem.getStartState()
    if problem.isGoalState(temp):
        return []
    visited = []
    stack = util.Stack()
    stack.push((temp, []))
    while not stack.isEmpty():
        temp, path = stack.pop()
        visited.append(temp)
        if problem.isGoalState(temp):
            return path
        for child in problem.getSuccessors(temp):
            if child[0] not in visited:
                stack.push((child[0], path + [child[1]]))

    util.raiseNotDefined()
```

B. The runnable code is in the search file named search.py, below is the screenshot of it.

```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    temp = problem.getStartState()
    if problem.isGoalState(temp):
        return []

    visited = []

    queue = util.PriorityQueue()
    queue.push((temp, [], 0), 0)

    while not queue.isEmpty():

        currentNode, actions, prevCost = queue.pop()

        if currentNode not in visited:
            visited.append(currentNode)

            if problem.isGoalState(currentNode):
                return actions

            for nextNode, action, cost in problem.getSuccessors(currentNode):
                newAction = actions + [action]
                newCostToNode = prevCost + cost
                heuristicCost = newCostToNode + heuristic(nextNode, problem)
                queue.push((nextNode, newAction, newCostToNode), heuristicCost)

    util.raiseNotDefined()
```