1 Problem

The answers to the following should fit in the white space below the question.

1. For each pair (f, g) below indicate the relation between them in terms of O, Ω, Θ. For each missing entry, write-down Y (for YES) or N (for NO) to indicate whether the relation holds (no need to justify your answers here). For example, if $f = O(g)$ but not $\Omega(g)$, then you should enter Y in the first box and N in the other two boxes. Similarly, if $f = \Theta(g)$, then you should enter Y in all the boxes. [1 point]

f	g	O	Ω	Θ
$\log_3 n$	$\log_9 n$	Y	Y	Y
3^n	6^n	Y	N	N

2. Is the following True or False: Consider a divide and conquer algorithm which solves a problem on an instance of length n by making five recursive calls to instances of length ($\lfloor n/2 \rfloor$) each, and combines the answers in $O(n^2)$ time. Then, the time-complexity of the algorithm is $O(n^2)$. [1 point]

False

$T(n) = 5T(\frac{n}{2}) + O(n^2) \Rightarrow O(n^{\log_2 5})$

3. State the principles behind the divide and conquer technique for designing algorithms. [1 point]

divide the problem into several subproblems, solve each subproblem, then combine all the subproblems to get the answer

4. What is the solution to the recurrence $T(1) = 1$, $T(n) = 2T(n/2) + 100n$? [1 point]

$T(n) = O(n \log n)$

$k = \log_2 2 = 1$, $100n = O(n^1)$, $k = 1$

$\Rightarrow T(n) = O(n \log n)$

5. Write down the definition of the discrete Fourier Transform for signals of length n (i.e., write down the expression we used for $DFT_n(a_0, a_1, \ldots, a_{n-1})$). [1 point]

matrix $n \times n$

$a_{ij} = w^{i \cdot j}$ $0 \le i \le n-1,\ 0 \le j \le n-1$

where $w = e^{i \cdot \frac{2\pi}{n}}$

6. Write down some pros and cons of the adjacency-list and adjacency-matrix representations of graphs. [1 point]

	pro	cons
adjacency-list	saves space, uses $O(\lvert E \rvert)$ space	slow to check existence of an edge
adjacency-matrix	$O(1)$ time check edge, easy to check existence of an edge.	waste space, uses $O(\lvert V \rvert^2)$ space

7. Let G be a weighted directed graph with positive weights. Suppose we ran Dijkstra's algorithm starting from a vertex s to compute the shortest distances from s to all vertices in G and let T be the tree formed by the PARENT links that are computed during the run of the algorithm.

Now suppose we change the weights of the graph as follows: for every edge e that is **not** part of T, its weight is doubled, i.e., its weight w_e is replaced with $2 \cdot w_e$. The weights of edges in T are not changed. This creates a new instance G' of the problem with the same underlying graph but different costs on all the edges which are **not part of** T.

True or false: "The shortest distances from s to other vertices in the new instance are the same as they were in the original weighted graph." If true, provide a brief explanation why and if false, provide an example of a graph G where the statement fails. [2 points]

True

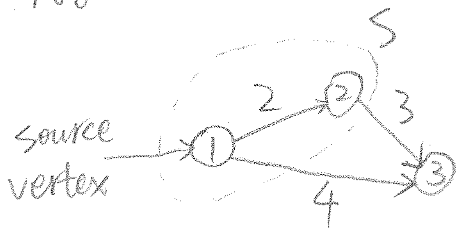
at each iteration of the algorithm, only the edge that causes a shortest distance to s is added to the tree. This edge is still the shortest, even through other edge lengths are doubled.

8. Let $G = (V, E)$ be a weighted undirected graph with positive weights and let s be a vertex in G. Consider a variant of Dijkstra's algorithm where we grow the set of vertices S by picking the vertex $v \notin S$ that has the shortest edge to any vertex in S. That is, consider the following algorithm:

```
(a) Set $S = \{s\}$. Set $c(s) = 0$ and $c(v) = \infty$ for all $v \neq s$.
(b) While $S \neq V$:
    i. For each vertex $v \notin S$, let $c'(v) = \min\{\ell_{(u,v)} : u \in S, (u,v) \in E\}$.
    ii. Find the vertex $v \notin S$ with least $c'(v)$ and let $u \in S$ be the corresponding vertex
        that achieves the minimum in the definition of $c'(v)$.
    iii. Set $c(v) = c(u) + \ell_{(u,v)}$.
    iv. ADD v to S
```

Does the above algorithm compute the lengths of the shortest paths from s to all other vertices? That is, are the numbers $c(v)$ computed by the algorithm the distances to v from s? If yes, provide a brief explanation why this may be true. If not, provide an example of a graph G where the algorithm fails to compute the lengths of the shortest paths. [2 points]

No



at second iteration of the algorithm, edge (2,3) is picked, but edge (1,3) causes a shortest distance from 1 to 3

2 Problem

Let n be an even integer and let Q_n denote the $n \times n$ matrix with rows and columns indexed by $0 \leq j, k \leq n-1$ and $Q_n[j,k] = e^{-2\pi i (j \cdot k)/n}$.

1. Can you identify any repeating pattern in the matrix Q_n (like the one we saw in our derivation of FFT for computing the discrete Fourier Transform)? [2 points]

2. Can you connect the matrices in the pattern to the matrix $Q_{n/2}$? [2 points]

1. let $w = e^{\frac{-2\pi i}{n}}$

$Q_n[j,k] = w^{j \cdot k}$ which is <u>equivalent</u> to FFT we derive except for value of w.

thus

$$\begin{array}{c|cc} & 0\ 2\ 4\ \cdots\ n\text{-}2 & 1\ 3\ 5\ \cdots\ n\text{-}1 \\ \hline 0,1,2,\ldots & A & B \\ \ldots,n\text{-}1 & A & w^{\frac{n}{2}} \cdot B \end{array}$$

move all even columns to front and all odd columns to back

repeating patterns:

$Q_n[j, 2k] = w^{j \cdot 2k} = w^{(\frac{n}{2}+j) \cdot 2k} = Q_n[\frac{n}{2}+j, 2k]$ for $0 \leq k \leq \frac{n}{2}-1$

$Q_n[j, 2k+1] = w^{j \cdot (2k+1)} = w^{\frac{n}{2}} \cdot w^{(\frac{n}{2}+j) \cdot (2k+1)} = w^{\frac{n}{2}} \cdot Q[\frac{n}{2}+j, 2k+1]$ for $0 \leq k \leq \frac{n}{2}-1$

$Q_n[j, 2k] = \frac{1}{w^j} \cdot Q_n[j, 2k+1]$ for $0 \leq k \leq \frac{n}{2}-1$

2. for submatrix A (top left of reordered matrix Q_n)

$Q_n[j, 2k] = w^{j \cdot 2k} = (w^2)^{j \cdot k} = Q_{\frac{n}{2}}[j,k]$ with $w' = w^2$

for $0 \leq j \leq \frac{n}{2}-1$ $0 \leq k \leq \frac{n}{2}-1$

$Q_{\frac{n}{2}}[j,k] = (w')^{j \cdot k}$ where $w' = w^2$
$= e^{-4\pi i (j \cdot k)/n}$

3 Problem

An array $A[0, 1, \ldots, n-1]$ is said to have a *majority element* if more than half of its elements are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as mp3 files, say; so in particular, you cannot sort the elements.) However you can answer questions of the form: "is $A[i] = A[j]$" in constant time.

Give an algorithm to solve the problem. For full-credit, your algorithm should run in time $O(n \log n)$. (You don't have to prove correctness or analyze the time-complexity of the algorithm.) [4 points]

(Hint: Split the array A into two arrays A_L and A_R of half the size each. Does knowing the majority elements of A_L and A_R help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

```
                        // return -1 means no majority,
def majority(A) :       // otherwise has majority
    split A into 2 arrays A_L and A_R of half size each
    let majorityL = majority(A_L)
    let majorityR = majority(A_R)

    if majorityL = -1 :
        if majorityR = -1 :
            return -1
        if count(A, majorityR) > 1/2 (length of A) :
            return majorityR
        else
            return -1
    if majorityR = -1 :
        if count (A, majorityL) > 1/2 (length of A) :
            return majorityL
        else
            return -1

    if count(A, majorityL) > 1/2 (length of A)
        return majorityL
    if count (A, majorityR) > 1/2 (length of A)
        return majorityR
    return -1
```

⟹ Continue on next page

```
// auxilary function
def count(A, m):
    set count = 0
    for i from 0 to (length of A)-1:
        if A[i] = A[m]:
            count = count + 1
    return count

def hasMajority(A):
    if majority(A) = -1:
        return false
    return -1
```

4 Problem

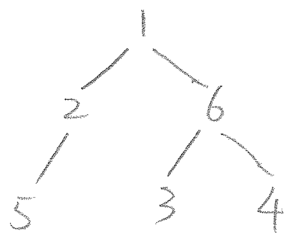
Let $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{\{1,2\}, \{1,6\}, \{2,5\}, \{2,6\}, \{3,4\}, \{3,5\}, \{3,6\}, \{4,6\}, \{5,6\}\}$. Suppose that G was given to you in adjacency list representation where the elements in the adjacency list are ordered in increasing order. For example, the adjacency list of vertex 2 would be $[1, 5, 6]$.

1. Draw the BFS tree that you would get when doing BFS starting from 1. [2 points]
2. Draw the DFS tree that you would get when doing DFS starting from 1. [2 points]

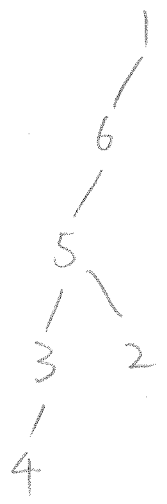
(You don't have to show all the stages of the algorithms just the final trees. Also, keep in mind that you process elements of the adjacency list in increasing order. For example, when doing DFS, you push vertices from an adjacency list onto the stack in increasing order.)

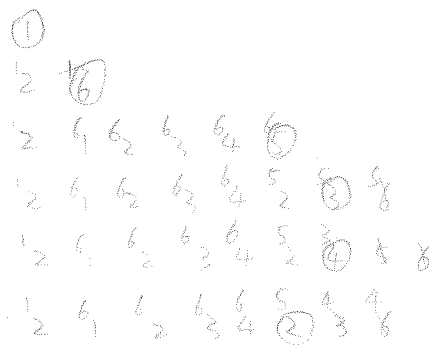
adjacency list: 1: 2 6
2: 1 5 6
3: 4 5 6
4: 3 6
5: 2 3 6
6: 1 2 3 4 5

1. BFS



2. DFS





5 Problem

Let $G = (V, E)$ be an undirected (unweighted) graph and for any two vertices $u, v \in G$, let $distance_G(u, v)$ be the length of the shortest path between u, v if one exists and ∞ if they are not connected. Define the diameter of a graph G to be the maximum distance between any two vertices of the graph G; that is, $diameter(G) = \max\{distance_G(u, v) : u, v \in G\}$. Give an algorithm that given a graph $G = (V, E)$ (in adjacency list representation) as input, computes the diameter of G, $diameter(G)$. For full-credit, your algorithm should run in time $O(|V|^2 + |V| \cdot |E|)$. [4 points]

(You don't have to prove correctness or analyze the time-complexity of your algorithm.) $|V| \cdot (|V| + |E|)$

```
set MAXDistance = 0
for each vertex v in V :
    initialize Discovered[u] = false  $\forall u \in V, u \neq v$
    set Discovered[v] = true
    L[0] = {v};  i = 1
    while L[i] is not empty :
        L[i+1] = {}
        for each vertex u ∈ L[i] :
            for each neighbor w of v :
                if Discovered[w] = false :
                    set Discovered[w] = true
                    Add w to L[i+1]
        i = i+1
    for each vertex u ∈ V :
        if Discovered[u] = false :        // graph disconnected
            MAXDistance = ∞
            return MAXDistance
    set MAXDistance = i-1
return MAXDistance
```