CS180 Exam 1

Nicole Wong

TOTAL POINTS

24.05 / 26

QUESTION 1

Problem 1 10 pts

1.1 Part 1: Asymptotics 1 / 1

- **0** 5/6 correct

1.2 Part 2: D&C Master 1 / 1

- **0** Correct answer

1.3 Part 3: D&C Principles 1 / 1

- **0** Correct principle. You mentioned 'divide/split' into subproblems, and combine their solutions.

1.4 Part 4: Recurrence 1 / 1

- **0** Correct answer

1.5 Part 5: DFT Definition 1 / 1

- **0** Correct expression, showing the value representation of the polynomial or the matrix representation of DFT calculation

1.6 Part 6: List vs Matrix 1 / 1

- **0** Correct answer, with time and space complexities for both representations

> Provide the storage/space complexities in terms of V and E, instead of a general n variable.

1.7 Part 7: Dijkstra changing weights 2 / 2

- **0** Correct answer, with an explanation as to why the shortest distances don't change

1.8 Part 8: Modified Dijkstra 2 / 2

- **0** Correct answer with a valid example as to why the modified algorithm will not work

QUESTION 2

Problem 2 4 pts

2.1 Pattern in matrix Q. 2 / 2

- **0** Correct

2.2 Relation to Q_{n/2} 1.25 / 2

- **0.75** Both side are wrong, but tried

QUESTION 3

3 Problem 3: Finding majority 4 / 4

- **0** Correct

QUESTION 4

Problem 4 4 pts

4.1 BFS Tree 2 / 2

- **0** Correct

4.2 DFS Tree 1.8 / 2

- **0.2** Order incorrect in one edge at a node close to 5.

QUESTION 5

5 Problem 5 3 / 4

- **1** Reasonable attempt with some minor issues or unclear presentation.

Mid-term. February 3, 2017

CS180: Algorithms and Complexity
Winter 2017

Guidelines:

- The exam is closed book and closed notes. Do not open the exam until instructed to do so.
- Write your solutions clearly and when asked to do so, provide complete proofs. You may use results we proved in class without proofs as long as you state what you are using.
- I recommend taking a quick look at all the questions first and then deciding what order to tackle to them in. Even if you don't solve the problems fully, attempts that show some understanding of the questions and relevant topics will get reasonable partial credit.
- You can use extra sheets for scratch work, but try to use the white space (it should be more than enough) on the exam sheets for your final solutions.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly and any cheating reported.

Problem	Points	Maximum
1		10
2		4
3		4
4		4
5		4
Total		26

Name	NICOLE WONG
UID	[redacted]
Section	1C

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	N	N	N

$\downarrow$

$(3 \cdot 2)^n$

$\downarrow$

$3^n 2^n$

not a constant

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]

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

$k = \log_2 5 = 2.\_\_ (>2)$

Master Thm case 1: $f(n) = O(n^{k-\varepsilon})$

$T(n) = O(n^k) = O(n^{\log_2 5})$ → False

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

1. Divide into subproblems
2. Conquer the subproblems by solving recursively
3. Combine the subproblems

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

$k = \log_2 2 = 1$

$f(n) = 100n = O(n)$

Master Theorem case 2: $f(n) = \Theta(n^k)$

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

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]

Let $\omega = e^{i(\frac{2\pi}{n})}$ and $DFT_n(\vec{a}) = \vec{r}$

$r_j = \sum_{i=0}^{n-1} (\omega^i)^j a_i$

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

Adjacency list: pro – takes less space to store
con – takes $O(|E|)$ to check if 2 vertices are directly connected by an edge.

Adjacency Matrix: pro – it takes $O(1)$ time complexity to see if 2 vertices (u, v) are directly connected by an edge.
con – it takes $O(n^2)$ space complexity to store

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

For every edge we connecting vertices (u, v), the fact that we was not added to T means that there is a shorter path from s to v. Doubling the weight of we does not create a shorter path to v, so when the algorithm is run on G', the same tree T is built.

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 $c'(v) = \min\{\ell_{(u,v)} : u \in S, (u, v) \in E\}$.

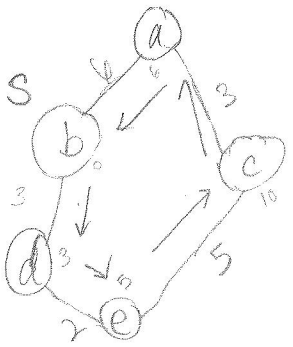
      ii. Find the vertex $v \notin S$ with least $c'(v)$ and let $u \in S$ be the corresponding vertex that achieves the minimum in the definition of $c'(v)$.

      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 Take the example graph on the left. Starting at vertex b, the shortest edge would be d, then e, then c. The value c(c) would be 10, which is incorrect since the shortest path to c is length 9.



2 Problem

$\frac{1}{e^{i(\frac{2\pi}{n})jk}}$

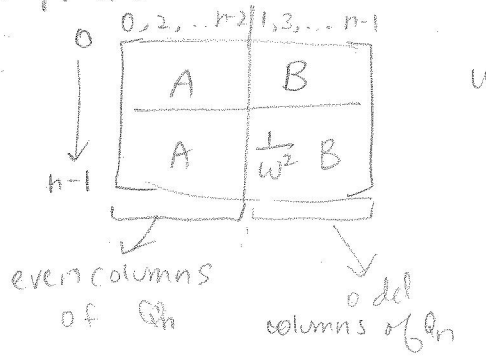
Let n be an even integer and let Q_n denote the $n \times n$ matrix with rows and columns indexed by $0 \leq j, k \leq n-1$ and $Q_n[j,k] = e^{-2\pi i(j \cdot k)/n}$.

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]

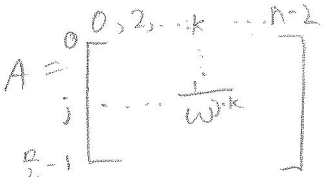
1. Let $\omega_n = e^{i(\frac{2\pi}{n})}$. Note that $\omega_n^n = 1$.

$Q_n[j,k] = \frac{1}{\omega^{j \cdot k}}$ which is essentially the same as our FFT matrix, only that each entry is the 1 FFT entry raised to the -1th power.

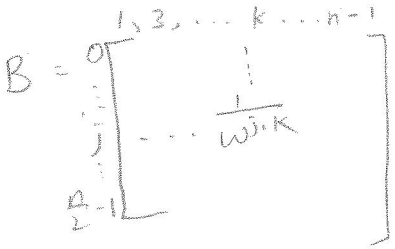
So a similar pattern as the FFT matrix can be found in Q_n, where



where $A =$



$B =$



2. $A = \omega^2 Q_{n/2}$

$B = \omega Q_{n/2}$

$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & & \\ 0 & 3 & & \end{bmatrix}$

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]

(*) I used a return value of -1 to indicate no majority

(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.)

```
MAJORITY(A,n):
    If n = 1, return A[0]
    Let m = ⌈n/2⌉
    M_L = Majority(A[0, 1, ... m-1], m)
    M_R = Majority(A[m, m+1, n-1], n-m)
    If M_L = -1 and M_R = -1
        return -1
    If M_L = -1
        return M_R
    If M_R = -1
        return M_L
    Let x ← 0, y ← 0
    For i in the range 0:n-1, inclusive:
        If A[i] = M_L:
            x ← x+1
    For j in the range 0:n-1, inclusive:
        If A[j] = M_R:
            y ← y+1
    If x > y:
        return M_L
    If x < y:
        return M_R
    If x = y
        return -1
```

1 2 6
2 1 5 6
3 4 5 6
4 3 6
5 2 3 6
6 1 2 3 4 5

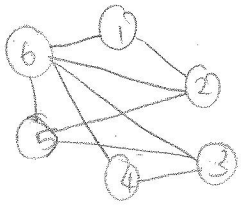
4 Problem

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]

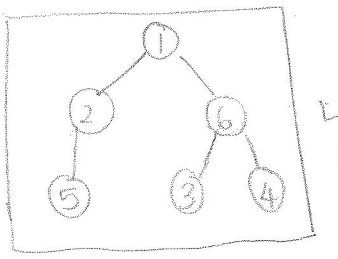
2. Draw the DFS tree that you would get when doing DFS starting from 1. [2 points]

(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.)



A
1 → 2 → 6
2 → 1 → 5 → 6
3 → 4 → 5 → 6
4 → 3 → 6
5 → 2 → 3 → 6
6 → 1 → 2 → 3 → 4 → 5

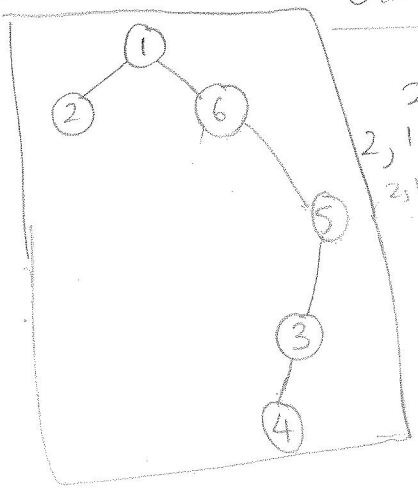
1. BFS Tree



List
$L_0 = \{1\}$
$L_1 = \{2, 6\}$
$L_3 = \{5, 3, 4\}$
$L_4 = \emptyset$

	Discovered					
	1	2	3	4	5	6
L_0	✓					
L_1	✓	✓				✓
L_3	✓	✓	✓	✓	✓	✓

2. DFS Tree



Stack	Explored					
	1	2	3	4	5	6
1						
2, 6	✓					✓
2, 1, 2, 3, 4, 5	✓					✓
2, 1, 2, 3, 4, 3	✓				✓	

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.)

Implementation

Max ← 0

For each vertex $v \in V$:

→ let $S \leftarrow v$

→ initialize dist[u] = ∞ for every $u \in V$, and dist[s] = 0.

→ use Dijkstra's algorithm to compute the shortest paths from s to $u \in V$ where $s \neq u$.

→ keep track of the distances to from s to u in an array dist[u]

→ iterate over dist:

→ if dist[u] > Max

→ Max ← dist[u]

return Max.

~~Dijkstra's Algorithm, modified~~

$S \leftarrow \{s\}$

Find $c'(v) = \min\{dist(u) + l_{uv} : u \in S \text{ and } (u, v) \in E\}$

For the smallest $c'(v)$