CS180 Exam 1

TOTAL POINTS

24.5 / 26

QUESTION 1

Problem 1 10 pts

1.1 Part 1: Asymptotics 1 / 1

- 0 All 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

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} 0.5 / 2

- 1.5 Partial attempt.

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 2 / 2

- 0 Correct.

QUESTION 5

5 Problem 5 4 / 4

- 0 Correct

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	
UID	
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	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]

$a=5,\ b=2,\ f=n^2\ \ d=2$

$\log_b(a) = \log_2(5) > 2 = d \rightarrow$ case 1: $n^{\log_2(5)}$

False

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

- Divide the problem of length n into subproblems
- Recursively call the algorithm on each subproblem
- "Merge" / iterate through each recursive call's solution to get the final solution.

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

$a=2 \quad b=2 \quad f=100n \rightarrow d=1$ since $n^1 = n^d$

$\log_b a = \log_2(2) = 1$ which $= d = 1$

So by case 2 or MT: $O(n \log n)$

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]

$DFT_n(a_0 \ldots a_{n-1})$:
$S = DFT_{\lfloor n/2 \rfloor}(\bar{a}_{even})$
$\bar{S} = DFT_{\lfloor n/2 \rfloor}(\bar{a}_{odd})$
for $j = 0$ to $n/2 - 1$
$r_j = S_j + \bar{S}_j \cdot \omega^j$
$r_{j+n/2} = S_j + \bar{S}_j \cdot \omega^j \cdot \omega^{n/2}$
return r

$$DFT_n = \sum_{j=0}^{n-1} e^{\left(\frac{2\pi i}{n}\right) \cdot j} \cdot a_j$$

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

Matrix

Pro	Con
• $O(1)$ lookup time	• $O(\|V\|^2)$ space

List

Pro	Con
• $O(\|E\|)$ space	• Have to iterate through list of neighbors to find edge

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. Dijkstra's (Ds) alg. is a greedy algorithm that adds nodes & edges to T such that edge added is the closest to the vertices already added to T. If we increase the weights of edges not added to T but not those in T, then Ds alg will again add the same edges since all $e \in T$'s edges are still closest to vertices already in T. They will add least weight to get shortest path as well.

* (that also has shortest paths satisfies the d(u) requirement)

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 $S \neq V$:

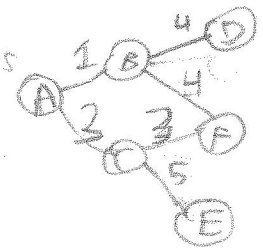
      i. For each vertex $v \notin S$, let $c'(v) = \min\{\ell_{(u,v)} : u \in S, (u, v) \in E\}$.

      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 — line 6 fails

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]



$S = \{A, B, C$
c: $A = 0$
$B = 2$
$C = 3$
$D = \infty$
$E = \infty$
$F = 6$

False. The algorithm fails at the Graph to left for when we have $S = A$ as start. This is because ABF is ~~longest~~ shortest path of 5 but the algorithm chose ACF ~~Edge~~ length 6. This is because the algorithm only looks for the shortest edge to any vertex in S & if we delay the shortest path as much as possible (by sticking in an edge that is larger than the others but overall sum is smaller) i.e. $BF = 4$, then the ~~route~~ alg. might go through a path of edges smaller than it, but perhaps each only 1less & make the sum larger. So the algorithm fails.

(Alg goes to A [+0], then B [$AB = 1$] & goes to C [$AC = 3$] and ignores $BF = 4$ since $BF = 4 > CF = 3$ so adds $CF = 3$.)

2 Problem

Let n be an even integer and let Q_n denote the $n \times n$ matrix with rows and columns indexed by $0 \le j, k \le n-1$ 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) $n=2$ $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{pmatrix} \omega^0 & \omega^0 \\ \omega^0 & \omega^1 \end{pmatrix}$

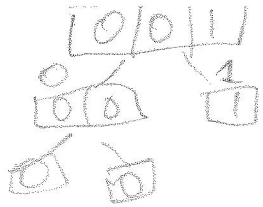
$n=4$ $\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^3 & \omega^2 & \omega^1 \\ \omega^0 & \omega^2 & \omega^4 & \omega^2 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$

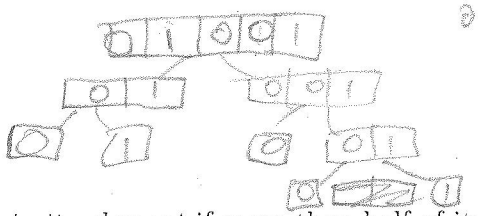
Like with normal FFT matrices, the even columns largely stay the same as before; they repeat after the n^{th} root of unity like ω^2 to ω^4 & back to ω^2. But the odd entries in Q_n are now in reverse order as the normal FFT counterparts both seen via column or row ($\omega^3, \omega^2, \omega^1$ & $\omega^1, \omega^2, \omega^3$). See the circled columns. This is because of the unit circle symmetry & that the (-) sign simply reverses direction around circle.

(2) ~~The matrices of Q_n have odd cols in reverse order, so~~

(2) The matrices of Q_n have odd cols in reverse order, so in $Q_{n/2}$, leave off the $\frac{n}{2}$ largest exponents of ω^x and take the $\frac{n}{2}$ smallest & reverse the order & put back into appropriate column (but leave ω^0 untouched)

So in $n=4$ we would take column 1: $\begin{bmatrix} \omega^0 \\ \omega^3 \\ \omega^2 \\ \omega^1 \end{bmatrix}$, drop the $\frac{4}{2}$, largest elements $= \begin{bmatrix} \omega^0 \\ \omega^1 \end{bmatrix}$ & reverse the remaining $\frac{4}{2}$ entries in column $= \begin{bmatrix} \omega^0 \\ \omega^1 \end{bmatrix}$ & put back into $n=2$ matrix at col 1. Even columns stay same, just only the lowest $\frac{n}{2}$ entries (ie: not use ω^4, but only ω^0 & ω^2).





0 is maj

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]

Divide & Conquer

(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.)

getMajority(A, n):
If n is 1, return A[0]
Initialize A_L ← left half of A (A_L = 1st $\lfloor \frac{n}{2} \rfloor$ elements)
Initialize A_R ← right 1/2 of A (A_R = last $\lceil \frac{n}{2} \rceil$ elements)
Let elements E_L & E_R hold left & right results:
E_L = getMajority(A_L, $\lfloor \frac{n}{2} \rfloor$)
E_R = getMajority(A_R, $\lceil \frac{n}{2} \rceil$)
If $E_L = E_R$:
return either of them, ie: (E_L)
Else
count = 0
for each element in A:
if A[i] = E_L, then count = count + 1
else, then count = count − 1
if count > 1
return E_L
else if count < 1
return E_R
else
return ∅ (did not find majority, was split exactly 1/2 & 1/2)

isThereMajority(A, n)
E = getMajority(A, n)
if E is not ∅, return true
else, return false

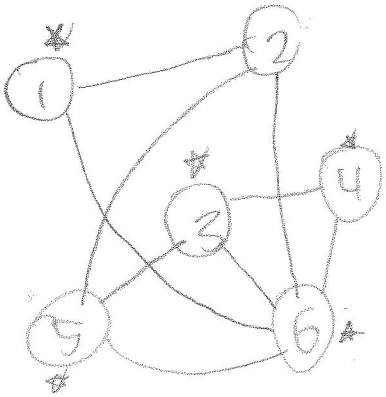
4 Problem

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 $[1, 5, 6]$.

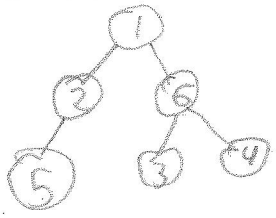
1. Draw the BFS tree that you would get when doing BFS starting from 1. [2 points]
2. Draw the DFS tree that you would get when doing DFS starting from 1. [2 points]

(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.)

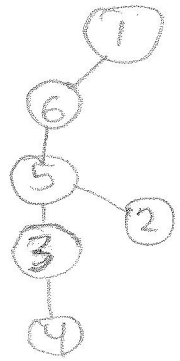
G:



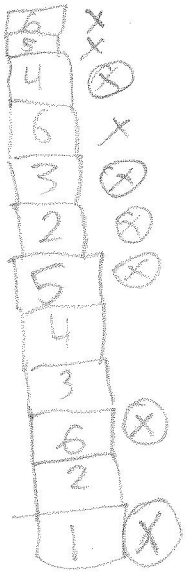
1. BFS tree:

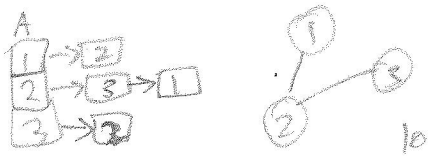


2. DFS tree:



	parent					
	1	2	3	4	5	6
R = {3}	∅	∅	∅	∅	∅	∅
R = {2, 6}	∅	1	∅	∅	∅	1
R = {2, 1, 2, 3, 4, 5}	6	6	6	6	6	1
{2, 1, 3, 4, 2, 3, 6}	6	5	5	6	6	5
{2, 1, 3, 4, 2, 3}	6	5	5	3	3	3
{2, 1, 3, 4, 1, 5, 6}	2	5	5	3	2	2
{2, 1, 2, 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.)

V^2 = look thru each vertex in n^2 manner

int maxD = 0

$V(V+E) = V^2 + V \cdot E$

For each V in V ($v \in V$):

- Let T = (graph of BFS tree)
- perform BFS on graph G starting at V & saving BFS tree in T.
- Go through T starting at V BFS style:
  - increment count by 1 for each edge traveled to & mark each vertex's $distance_G(v, u)$ for each u encountered
  - If $distance_G(v, u) > maxD$, $maxD = distance_G(v, u)$

end loop

return maxD;