NAME: _____

CSE 331

Introduction to Algorithm Analysis and Design

Sample Final Exam Solutions: Fall 2017

Atri Rudra November 22, 2017

DIRECTIONS:

- Closed Book, Closed Notes except for two $8\frac{1}{2}" \times 11"$ review sheet.
- Time Limit: 2 hours 30 minutes.
- Answer the problems on the exam paper.
- Each problem starts on a new page.
- Make sure you write your NAME on the paper.
- If you need extra space use the back of a page.
- Problem 6 is a bonus problem.
- Keep your desk clear of everything else other than the exam paper, review sheets and writing implements.

1	/10
2	/30
3	/25
4	/20
5	/15
6	/2
Total	/100
Bonus	/2

FEW GENTLE REMINDERS/SUGGESTIONS:

- You can quote any result that we covered in class or any problem that was there in a homework (but remember to explicitly state where you are quoting a result from).
- If the question does not specifically ask for a formal proof, just a correct proof idea should fetch you at least 80% of the points.
- If you get stuck on some problem for a long time, move on to the next one.
- The ordering of the problems is somewhat related to their relative difficulty. However, the order might be different for you!
- You should be better off by **first reading all questions** and answering them in the order of what you think is the easiest to the hardest problem. Keep the points distribution in mind when deciding how much time to spend on each problem.
- Spend time on the bonus problem only if you are done with the rest of the exam.
- Finally, relax and enjoy the exam! (If not, think of a time when you'll be done with 331!)

1. ($5 \times 2 = 10$ points) Answer True or False to the following questions. **No justification** is required. (Recall that a statement is true only if it is logically true in all cases while it is false if it is not true in some case).

   (a) BFS is a linear time algorithm.

   TRUE BFS runs in time $O(n+m)$ and the input size (i.e. the graph) is also $\Theta(n+m)$.

   (b) Every undirected connected graph on n vertices has exactly $n-1$ edges.

   FALSE (3.2) Consider G that is a cycle.

   (c) If all the edge weights of an undirected connected graph G are distinct, then G has a unique minimum spanning tree.

   TRUE This I had mentioned in class but here is a very hand wavy proof idea: say there are two distinct MSTs T_1 and T_2 and let e be the largest weighted edge that is present in one but not the other (WLOG say it is in T_1). Then the claim is that one can remove e and replace it with and egde e' in $T_2 \setminus T_1$. The claim follow by arguing that $T \setminus \{e\} \cup \{e'\}$ is a spanning tree and that $c_{e'} < c_e$.

   (d) Given n numbers $a_1, \ldots, a_n$, the median of the smallest ten numbers and the largest ten numbers among them can be computed in $O(n)$ time.

   TRUE With a linear scan one can compute the smallest and largest ten numbers. Then the median of the 20 numbers can be found in $O(1)$ time.

   (e) The *maximum* spanning tree problem (i.e. given a connected undirected weighted graph output a spanning tree with the maximum weight) is an NP-complete problem. (Recall that we have shown that the *minimum* spanning tree problem is in P.)

   FALSE Consider the related graph instances where the edge weights are $-c_e$ (where c_e are the original costs) and then run Kruskal's algorithm, which gives a polynomial time algorithm and hence the maximum spanning tree problem is in P.

2. ($5 \times 6 = 30$ points) Answer True or False to the following questions and **briefly JUSTIFY** each answer. A correct answer with no or totally incorrect justification will get you 2 out of the total 6 points. (Recall that a statement is true only if it is logically true in all cases while it is is false if it is not true in some case).

   (a) $2^{O(n)}$ is $O(2^n)$. (Or more precisely, every function $f(n)$ that is $2^{O(n)}$ is also $O(2^n)$.)

   FALSE Consider $f(n) = 4^n$. Note that $f(n) = 2^{2n} = 2^{O(n)}$ but it is not $O(2^n)$. (A quick way to see this note that $\lim_{n\to\infty} 4^n/2^n = \infty$.)

   (b) Given n numbers $a_1, \ldots, a_n$, where for every $1 \le i \le n$, $a_i \in \{-5, 9, 100\}$; their sorted order can be output in $O(n)$ time.

   TRUE Do a linear scan of the numbers and construct three lists: one each for every a_i equal to -5, 9 or 100. The final sorted output is to output the lists for $-5, 9$ and 100 in that order.

   (c) Given two numbers with n *octal* digits (i.e. the numbers are in base 8), they cannot be multiplied in time asymptotically faster than $O(n^2)$.

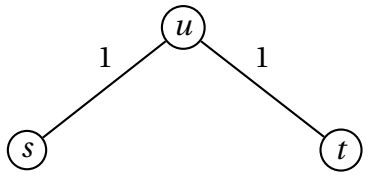
   FALSE The algorithm we saw in class (that runs in time $O(n^{\log_2 3})$ works for any base that is constant. Thus we can multiply octal number faster than $O(n^2)$ time.

(d) Given an undirected unweighted graph G in n vertices and m edges and two distinct vertices $s \neq t$, the shortest $s-t$ path can be computed in $O(m+n)$ time.

TRUE Run BFS on G starting with s. The shortest path between s and t is the unique path between them in the BFS tree.

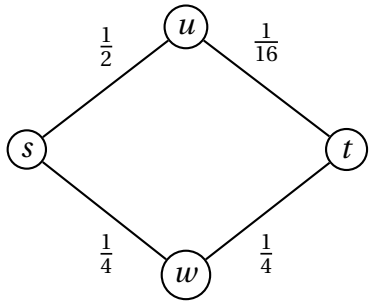
(e) Let G be an undirected connected graph. If G has a unique minimum spanning tree, then all the edge weights in G are distinct.

FALSE. Consider the graph



3. (25 points) You're given the Internet as a graph $G = (V, E)$ such that $|V| = n$ and $|E| = m$. Further for each edge $e \in E$, you're given it's probability $0 \leq p_e \leq 1$ that it'll transmit a packet (or equivalently, it is the probability it will not *fail*). Further, the probabilities are *independent*, i.e. given a path with edges $e_1, \ldots, e_\ell$, the probability of that the entire path does not fail is given by $\prod_{i=1}^{\ell} p_{e_i}$. Design an $O(m \log n)$ time algorithm, which given two distinct vertices $s \neq t \in V$, outputs the $s-t$ path with the largest probability of not failing. (If you were deciding to route packets from s to t, you should use this path.)

   For simplicity, you can assume that each p_e is a power of $\frac{1}{2}$. For example, in the graph below,



   The path s, u, t has a probability of not failing of $\frac{1}{2} \times \frac{1}{16} = \frac{1}{32}$, while the path s, w, t has a probability of not failing of $\frac{1}{4} \times \frac{1}{4} = \frac{1}{16}$. Thus, your algorithm should output s, w, t.

   Argue the correctness of your algorithm (formal proof is not required). Also briefly justify the run time of the algorithm.

   (*Hint:* It *might* be useful to transform the input and then apply a known algorithm on the transformed input. Further, these identities might be useful: $\log(ab) = \log a + \log b$ and $\log(a/b) = \log a - \log b$. (These identities hold irrespective of the base in the logarithms and hence are not explicitly stated.))

SOLUTION. Consider the following algorithm:

(a) Construct a new weighted graph $G' = (V', E')$ with $V' = V$ and $E' = E$ but for every $e \in E'$ define $\ell_e = \log(1/p_e)$.

(b) Run Dijkstra's algorithm on G' (and edges lengths ℓ_e as above). Return the shortest $s-t$ path from the run of the Dijkstra's algorithm as the path with the smallest probability of failing in G.

Correctness. Consider the path with edges $e_1, \ldots, e_a$. Note that the probability of failure of this path is

$$\prod_{i=1}^{a} p_{e_i} = \prod_{i=1}^{a} \left(\frac{1}{2}\right)^{\ell_{e_i}} = \left(\frac{1}{2}\right)^{\sum_{i=1}^{a} \ell_{e_i}} = 2^{-\sum_{i=1}^{a} \ell_{e_i}}.$$

Note that the length of the path in G' is $\sum_{i=1}^{a} \ell_{e_i}$. Thus, a path has the smallest probability of failure if and only if it is a shortest path in G'. Since Dijkstra is known to output the shortest path (note that since p_e is a power of 2, $\ell_e \geq 0$ for every $e \in E$), the correctness of the algorithm above follows.

Runtime analysis. We assume that all probabilities are constant and hence ℓ_e for every $e \in E'$ can be computed in $O(1)$ time. This implies that we can construct G' with the edge length in time $O(m+n)$, which is the runtime of Step (a). We saw in class that step (b) takes $O(m \log n)$ time, which proves the required time bound.

4. (20 points) (This is an CS job interview question.) Given two arrays A and B each with n numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ respectively, the algorithm needs to output the order $a_1, b_1, a_2, b_2, \ldots, a_n, b_n$. E.g. if $n = 4$ and $A = (1,3,5,7)$ and $B = (2,4,6,8)$, then the output is $(1,2,3,4)(5,6,7,8)$. The output must be in the same array as the input. You can think of the input as an array of size $2n$ which contains A and B and your algorithm must rearrange these values as specified.

What makes the problem interesting that you are only allowed $O(\log n)$ amount of *temporary space*. By temporary space, I mean the total amount of space among all data structures and temporary variables *except* the arrays A and B. In particular, the output has to be written in A and B.

Give an $O(n \log n)$-time algorithm that uses $O(\log n)$ temporary space. To receive full credit, you must fully specify your algorithm. Argue why your algorithm is correct and justify the running time and temporary space usage of your algorithm.

(*Note:* If you present an $O(n^2)$ time algorithm with $O(\log n)$ temporary space or a $O(n \log n)$ time and $n + O(1)$-temporary space algorithm then you can get at most 5 points.)

(*Hint:* It *might* be useful to think of a divide and conquer algorithm.)

SOLUTION. To make the description of the algorithm easier we first re-state the problem equivalently. We are given the input in the array $A[0, \ldots, 2n-1]$ and output should be as follows: for $0 \leq i \leq n-1$, $A[2i]$ is the original $A[i]$ and $A[2i+1]$ is $A[n+i]$.

Now note that if we swap the 2nd the 3rd quarter of the original array A and *then* break it up into the middle, then we have two *independent* instance of the problem each with $2 \cdot n/2 = n$ numbers. We make sure that we perform the swap in $O(n)$ time and use only constant space to do this. We then recurse and the runtime analysis and space usage follows by solving the simple recurrence relations.

Here is the statement of the algorithm (we will use A as a global array that all recursive calls can access and we begin with the call Swap$(0, 2n-1)$):

Swap(ℓ, r) //ℓ is left most index and r is right most index into A
$m \leftarrow r - \ell + 1$
If $m = 2$ then **return**.
For $i = 0 \ldots m/4 - 1$
  $tmp \leftarrow A[\ell + m/4 + i]$
  $A[\ell + m/4 + i] \leftarrow A[\ell + m/2 + i]$
  $A[\ell + m/2 + i] \leftarrow tmp$
Swap$(\ell, \ell + m/2 - 1)$
Swap$(\ell + m/2, r)$

Correctness. Note that when $n = 1$ (i.e. $m = 2$ above) then the array A already has permuted order. For the more general case, the argument follows from the discussion above the algorithm description.

Resource Usage. It is easy to check that the run time $T(n)$ and the temporary space usage $S(n)$ are given by the following recurrences:

- $T(2) \leq c$ and $T(n) \leq cn + 2T(n/2)$.
- $S(1) \leq c$ and $S(n) \leq c + S(n/2)$.

We have seen in class (and HWs) that the above implies $T(n) \leq O(n \log n)$ and $S(n) \leq O(\log n)$, as desired.

5. (15 points) Recall that the Bellman-Ford algorithm for input graph $G = (V, E)$ with costs c_e in each edge $e \in E$ builds an $n \times n$ matrix M such that $M[i, u]$ for $0 \leq i \leq n-1$ and $u \in V$ contains $OPT(i, v)$, i.e. the cost of a shortest $u - t$ path using at most i edges (for a given target $t \in V$). In particular, it computes

$$M[i, u] = \min\left(M[i-1, u], \min_{(u,w) \in E}\left(M[i-1, w] + c_{(u,w)}\right)\right).$$

In class we discussed a way to compress the matrix above: if $M[i, u] = M[i-1, u]$, then we do not need to explicitly store $M[i, u]$. In this problem, you will show that even with this compression idea a naive implementation of the Bellman-Ford algorithm still needs to store $\Omega(n^2)$ entries. (Recall that in class we later saw that the algorithm only needs to

store two columns at a time– what this problem is saying that the idea of *compression alone* will not give you any benefit over the naive algorithm.)

In particular, define an entry (i,u) to be *fresh*, if $M[i,u] < M[i-1,u]$. For every large enough $n \geq 1$ show that there exists a problem instance to the shortest path problem (with possibly negative weights but not negative cycle) on n vertices such that $\Omega(n^2)$ entries in the matrix M as computed by the Bellman-Ford algorithm are fresh.

(*Note*: If you present an example for any fixed $n \geq 4$ such that the matrix has M has at least $n(n-1)/2$ fresh entries, then you can get up to 3 points.)

SOLUTION. Consider the following graph $G = ([n], E)$, where E has the following edges and costs:

- $(i, i+1) \in E$ for every $1 \leq i \leq n-1$ with $c_e = -2$.
- $(i, n-j) \in E$ for every $1 \leq i \leq n-2$ and $0 \leq j \leq n-i-1$ with $c_e = j$.

The terminal vertex will be n.

Atri's note: In the actual exam it would be better to present the graph as a figure instead of the formal definition above. It's harder to do figures in LaTeX and hence, I went through the route above.

It can be checked that in the graph above, for every $1 \leq j \leq n-1$ the shortest path from $n-j$ to n has cost $-2j$ and has length j. (Take the path $n-j, n-j+1, \ldots, n-1, n$.) For $i < j$, the shortest path from $n-j$ to n of length i is to take the edge $(n-j, n-(i-1))$ and then to take the shortest path from $n-i+1$ to n. Note that the cost of this path is $i-1-2(i-1) = -(i-1)$. In other words, for every $u \in [n-1]$ we have $M[i,u] < M[i-1,u]$ for every $i \leq n-u$. Thus, we have $\Omega(n^2)$ fresh entries, as desired.