Sample Solutions - Past Exam 2

Q1

This specific recurrence relation of this question could be solved using methods such as telescoping, Master's theorem or recursion trees. Below we present a detailed solution using the telescoping method, which is quite general. Note that in the exam question you only had to select the correct asymptotic complexity.

We first have that

$$T(n) = 2T\left(\frac{n}{2}\right) + cn\log(n)$$

First note that the base of this log is not explicitly given, but this is not relevant for the asymptotic complexity as for constants a and b a change from base a to base b would only result in a multiplication by a constant, which is irrelevant for the asymptotic complexity. In the following we will use base 2 to fully develop using the telescoping method the recurrence relation

$$T(n) = 2T\left(\frac{n}{2}\right) + cn\log_2(n)$$

To now compute this we want to substitute in a lower recurring term, we can do this via using our recurrence relation to find $T(\frac{n}{2})$ and appropriately subbing into the above. Doing this we get

$$\begin{aligned}
T(n) &= 2\left[2 \cdot T\left(\frac{n}{2^2}\right) + c\left(\frac{n}{2}\right)\log_2\left(\frac{n}{2}\right)\right] + cn\log_2(n) \\
&= 2^2T\left(\frac{n}{2^2}\right) + cn\left(\log_2(n) + \log_2\left(\frac{n}{2}\right)\right) \\
&= 2^2T\left(\frac{n}{2^2}\right) + cn\left(2\log_2(n) - (1)\log_2(2)\right)
\end{aligned}$$

We keep repeating this procedure, by again using our recurrence relation to find $T(\frac{n}{2^2})$ and appropriately subbing into the above. Doing this we get

$$\begin{aligned}
T(n) &= 2^2 \cdot \left[2 \cdot T\left(\frac{n}{2^3}\right) + c\left(\frac{n}{2^2}\right)\log\left(\frac{n}{2^2}\right)\right] + cn(2\log_2(n) - \log_2(2)) \\
&= 2^3 \cdot T\left(\frac{n}{2^3}\right) + cn\left(2\log_2(n) - \log_2(2) + \log\left(\frac{n}{2^2}\right)\right) \\
&= 2^3 \cdot T\left(\frac{n}{2^3}\right) + cn(3\log_2(n) - (1+2)\log_2(2))
\end{aligned}$$

From here we can infer the general form which is just

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + cn\left(k\log_2(n) - \left(\frac{k(k-1)}{2}\right)\log_2(2)\right)$$

We now want to get rid of the recurrence term, we can find an appropriate value of k such that we invoke the base case. This occurs when $T(\frac{n}{2^k}) = T(1) = b$, hence $k = \log_2(n)$. Subbing this

into our general form we get

$$\begin{aligned}
T(n) &= 2^{\log_2(n)} \cdot T\left(\frac{n}{2^{\log_2(n)}}\right) + cn\left[\log_2(n)\log_2(n) - \left(\frac{\log_2(n)(\log_2(n)-1)}{2}\right)\log_2(2)\right] \\
&= nb + cn\left[\log_2(n)\cdot\log_2(n) - \left(\frac{\log_2(n)\cdot\log_2(n) - \log_2(n)}{2}\right)\right] \\
&= nb + cn\left[\frac{2\cdot\log_2(n)\cdot\log_2(n) - \log_2(n)\cdot\log_2(n) - \log_2(n)}{2}\right] \\
&= nb + cn\left[\frac{\log_2(n)\cdot\log_2(n) - \log_2(n)}{2}\right] \\
&= nb + \frac{cn}{2}\cdot\log_2(n)\cdot\log_2(n) - \frac{cn}{2}\log_2(n)
\end{aligned}$$

Thus clearly $T(n) = \Theta\left(n\cdot\log_2(n)\cdot\log_2(n)\right)$.

Q2

The worst-case time complexity of the code is $O(V+E)$. This is because the code first creates an array of zeroes for each vertex, taking $O(V)$ time. Next, it iterates over all the V edges in the graph and does work corresponding to the outdegree of each vertex. Although this is variable amount of work in each iteration, overall, after every vertex has been processed, the summation of scanning all the outdegrees will simply be the same as scanning every edge in the graph at most twice. Hence, this provides $O(E)$ work to update the `degrees` array. Finally, the last line describes an $O(V)$ operation to obtain the maximum degree found over all the tracked degrees for each vertex. Hence, overall the worst-case time complexity is $O(V+E)$. In this same logic, the total space complexity would be $O(V+E)$ due to storing the graph representation as an adjacency list, and the auxiliary space complexity would be $O(V)$ due to the array `degrees`.

Q3

Answer: **C**

(A) This is false. Assuming $\mathsf{L}[1\ldots i]\%m$ returns a list of all the integer remainders if every element is divided by m, x is definitely not the same as this. Small example take the list $\mathsf{L}[m, m, \ldots, m]$, the invariant states $x = 0$ but it in reality $x = mn$.

(B) Same reasoning as (A)

(C) Since all others are false you can induce this is the correct answer. For a breakdown as to why it's correct see Q4

(D) This is false. Doesn't include any iterator in the invariant itself making the proposed "loop" invariant not a loop invariant. The statement is saying x should always be equal the sum of the all elements in the list with a factor m no matter the state of the algorithm, but at initialisation $x = 0$.

(E) Same as (A) but worse as it doesn't include any iterator in the invariant itself making it not a loop invariant, like (D).

(F) This is false, this can be seen by looking at the initialisation of the proposed loop invariant. At initialisation $x = 0$, $i = 1$, the invariant states "$x = 0$ is the sum of numbers with factor of m in $\mathsf{L}[1 \dots 1]$", but $\mathsf{L}[1]$ could have a factor m, in which case x should have been $\mathsf{L}[1]$ and not zero if this invariant was to hold.

Q4

To justify the loop invariant, we should follow a procedure similar to an induction proof. The true invariant was: "x is the sum of all numbers with factor m in list, $\mathsf{L}[1...i-1]$". To justify its correctness each time the loop runs, we should start by identifying if the base case is true. On the initial entry to the loop, the current value of x is 0 and the current value of i is 1. If the loop invariant is applied it would read as: "0 is the sum of all numbers with factor m in list, $\mathsf{L}[1...0]$". This expression $\mathsf{L}[1...0]$ denotes the empty list and hence the statement is true, since 0 would indeed be the sum of no numbers to consider.

Next, assume we are in the k^{th} loop, meaning $i = k$, and the invariant statement has so far held true. That means the current value of x correctly represents the sum of all numbers with factor m involving the list, $\mathsf{L}[1..k-1]$. Following the logic of the problem, the `if` condition will be met if the next element of the list, $\mathsf{L}[k]$, which has yet to be seen, has a factor m or not by confirming through the modulo operation. If it has, the direct next line will execute and x will be increased by $\mathsf{L}[k]$. If it hasn't, x will remain the same. No matter what happens, after this conditional, the value of x will now represent the sum of numbers with factor m of the list range $\mathsf{L}[1...k]$. At this point, the `for` loop repeats and i is updated to $k + 1$. Hence, the invariant will now read as: "x is the sum of all numbers with factor m in list, $\mathsf{L}[1...k]$". Since $i = k+1$ and x was untouched since the last iteration, the invariant at the next loop is true!

The invariant has now been proven to be true each time the loop runs. It is now important to prove the termination of the loop and that the correctness of the result. The terminating condition of the loop would be when $i = n + 1$ where n is the length of the list. When this value is achieved, the loop contents will not be run, and instead the function will return x. No logic has been executed to change the value of x, hence the invariant should still hold for this final terminating i value. Therefore, x represents the sum of the numbers with factor m in the list, $\mathsf{L}[1..n] = \mathsf{L}$. Hence, the termination and final correctness of the algorithm has now been proven.

Q5

No. Whilst increasing the base representation does decrease the number of "digits" of each of our input integers, and therefore decreasing the number of Counting Sorts performed, the cost of our Counting Sort will increase. This is because the complexity of Counting Sort is $O(n+u)$ where n is the number of numbers being sorted and u is the universe size. Within the context of Radix Sort u is simply the base. Thus it isn't a good idea to naively increase the representation, you should seek to balance it. Generally the base being equal to n is appropriate (theoretically).

Q6

Answers: $\mathbf{\Theta(1)}, \mathbf{\Theta(E)}, \mathbf{\Theta(N(A))}, \mathbf{\Theta(N(A))}$.

For the second part, note that graph is connected.

Q7

1. A
2. B
3. E
4. G
5. H
6. F
7. D
8. C

Q8

Answer: **A**, **B**

(A) Using a max heap will mean the highest edge weights are used first and as long as the same cycle avoidance is built in, this will correctly find the maximum spanning tree. Therefore this statement is **true**.

(B) Negating the edges will mean that when the edges are sorted, the most negative (which were the largest positive weights) will be chosen first and hence this heuristic will lead to a maximum spanning tree as it will follow the same logic. Therefore this statement is **true**.

(C) When adding a certain edge, $E = (u, v)$, in Kruskal's algorithm, the tree containing vertex u and the tree containing vertex v will be merged as long as they have different parent IDs. However, the parent ID of only the root of the smaller tree will be updated to the root of the larger tree (due to union-by-size heuristic). This does not correspond to an actual edge in the graph and hence the parent-array does not indicate the edges of a minimum spanning tree. Therefore this statement is **false**.

(D) Prim's algorithm's greedy nature comes from just selecting the lowest weighted edge possible, incurring the smallest overall cost to its minimum spanning tree total weight. It's greedy selection does not track distances, nor will negative numbers intefere with any utilised assumption such as in Dijkstra's algorithm. Hence, Prim's algorithm will function correctly with negative numbers. Therefore this statement is **false**. See solution of **Problem 2 of Week 6 Applied** for another explanation.

Q9

We first create an order array as we need to return some ordered list which gives a topological sorting. The empty array is created at the beginning of the TRAVERSE function.

We realise that during a depth first search we will visit all the vertices that depend on some vertex before completing the depth first search from that vertex itself. Therefore if during depth-first search we add each vertex to an array after we have finished visiting all of its descendants, its descendants will have already been added to the array before it, so the array will contain a reverse topological order.

Hence we just need to add the vertex to the order array once its completed its DFS. So a line can be added at the end of the DFS function appending u to the order array.

And finally we just need to return this array in reversed order. Hence we add a return statement in end of the TRAVERSE fucntion returning the reversed order array.

Q10

Answer: **32**

Performing the Bellman-Ford algorithm for the first outer iteration will provide the following distance estimates (following the given relaxation order):
S) 0, A) 1, B) 5, C) ∞, D) 20.

The distance estimates the end of the second iteration of the outer loop are:
S) 0, A) 1, B) 4, C) 7, D) 20.

Therefore dist[A] + dist[B] + dist[C] + dist[D] $= 1 + 4 + 7 + 20 = 32$.

Q11

Answer: **46**.

Q12

Answer: **8**

The current flow is 6 into the sink. However, 1 more unit of flow can be augmented through $s \to a \to c \to b \to d \to t$ and another unit of flow can augmented through $s \to a \to c \to d \to t$.

Q13

Answer: **S**, **A**, **B**, **C**.

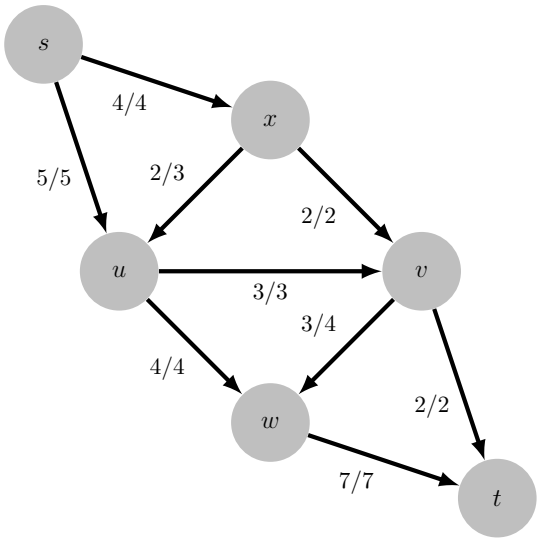
Quickest way to derive is simply perform Ford-Fulkerson and then complete a BFS/DFS to see what nodes can be reached in the residual graph, this will make up your S set.

Q14

Answer: **A**

Problem 2 can be immediately discarded since a circulation with demands problem is only feasible if the sum of all demands equals 0. This condition is violated in Problem 2.

Problem 1 satisfies the summation constraint as well as its edges clearly have the capacity to service each node, so we cannot immediately rule it out. We must construct a network flow graph and solve the problem to see if it has a feasible solution.



By performing Ford-Fulkerson on the Problem 1 graph, the source and sink nodes will be correctly saturated and hence Problem 1 is feasible.

Q15

Answer: **B**

Deleting 90 doesn't not result in any re-balancing. Inserting 12 the tree rooted at 10 would need to be re-balance. This is a right left case. After re-balancing no further re-balance is needed.

Q16

Answer: **D**

In the worst case, all N items would be hashed to same position and exist within the same data structure as part of the chain.

(A) A binary search tree can become unbalanced on one side if all the numbers inserted arrive in either an ascending or descending manner, meaning that an insert in the worst case would take $O(N)$ time.

(B) A sorted linked list would require $O(1)$ to perform the insert once the correct position is known, however it would still require $O(N)$ work to search if the item already exists/what is the correct position to insert. Methods such as binary search for this step are not applicable since you cannot immediately check entries in a linked list in $O(1)$ time.

(C) In the case of a sorted array, in the worst case $O(N)$ elements need to be moved to insert the new element in the correct position.

(D) An AVL tree have $O(\log(N))$ search and insert operations after the hashing due to maintaining balance.

Q17

Answer: **C**, **D**

We have compared up to length $k = 2$ beforehand. If the ranks of suffixes with IDs x and y are equal, then we need to compare the ranks of suffixes with IDs $x + k$ and $y + k$ to determine their relative rank after sorting on the first $2k$ characters.

(A) $\mathsf{Rank}[4 + k] < \mathsf{Rank}[6 + k]$ hence false.

(B) $\mathsf{Rank}[5 + k] \neq \mathsf{Rank}[7 + k]$ hence false.

(C) $\mathsf{Rank}[4+k] < \mathsf{Rank}[6+k]$ hence true.

(D) $\mathsf{Rank}[3+k] = \mathsf{Rank}[5+k]$ hence true.

Q18

Given the unsorted list of N superheroes, we can run the Quickselect algorithm to find the position and subsequently have partitioned the array into the upper 10 percentile. You would need to run `Quickselect(superheroes, 0.9N)`. The algorithm would ensure that the elements in the array above position $0.9N$ are all superheroes in the top 10% based on power level. This is done in $O(N)$ time.

Removing these heroes from the list takes $O(N)$ (simply just take the other side of the list as a subarray). Finally, the process is repeated, where N now corresponds to the size of the remainder list.

The median power level of the first team can be found using Quickselect again for the 50^{th} percentile. Once found, a simple loop over team 2 will allow the differences of power levels to be calculated and summed to get the total power level required to be gained.

The algorithm is done in $O(N)$ due to 3 calls of Quickselect which runs in $O(N)$ and a simple $O(N)$ loop at the end.

Q19

This question can be modelled as a bipartite matching problem. The network can be created as follow:

- Create 240 student nodes/vertices and 30 company nodes/vertices.
- From the source have an edge going to each student node/vertex with capacity 1.
- From each student node/vertex have 1-3 outgoing edges to their respective preferred company node/vertex each with capacity 1.
- From each company node have an outgoing edge to the sink with capacity 8.

The resulting max flow of this network would place the most of amount of students to a preferred a preferred company whilst maintaining that only 8 can be placed at each company.

Q20

This problem requires understanding the DP optimal substructure to get the number of paths from the starting position to any cell or equivalently from any cell to the final position. For the substructure, please refer to the solution of **Problem 2 of Week 7 Applied**.