Sample Solutions - Past Exam 1

Q1

Answer: **C**

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 initially have

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n^3 \tag{1}$$

and we want to get rid of the $T\left(\frac{n}{2}\right)$ term in the right-hand side. From the recurrence relation we get that

$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{2^2}\right) + c \cdot \left(\frac{n}{2}\right)^3.$$

Substituting $T\left(\frac{n}{2}\right)$ in Equation 1 by this expression we get

$$\begin{aligned} T(n) &= 2\left[2 \cdot T\left(\frac{n}{2^2}\right) + c \cdot \left(\frac{n}{2}\right)^3\right] + c \cdot n^3 && (2)\\ &= 2^2 T\left(\frac{n}{2^2}\right) + c \cdot \left(\frac{n^3}{4}\right) + c \cdot n^3 && (3)\\ &= 2^2 T\left(\frac{n}{2^2}\right) + c \cdot n^3 \left[1 + \frac{1}{4}\right] && (4) \end{aligned}$$

and now we want to get rid of the $T\left(\frac{n}{4}\right)$ term in the right-hand side. From the recurrence relation we get that

$$T\left(\frac{n}{2^2}\right) = 2 \cdot T\left(\frac{n}{2^3}\right) + c \cdot \left(\frac{n}{2^2}\right)^3.$$

Substituting $T\left(\frac{n}{4}\right)$ in Equation 4 by this expression we get

$$\begin{aligned} T(n) &= 2^2\left[2 \cdot T\left(\frac{n}{2^3}\right) + c \cdot \left(\frac{n}{2^2}\right)^3\right] + c \cdot n^3 \left(1 + \frac{1}{4}\right) && (5)\\ &= 2^3 \cdot T\left(\frac{n}{2^3}\right) + c \cdot n^3 \left[1 + \frac{1}{4} + \left(\frac{1}{4}\right)^2\right] && (6) \end{aligned}$$

Continuing this pattern for k iterations would lead to:

$$\begin{aligned} T(n) &= 2^k \cdot T\left(\frac{n}{2^k}\right) + c \cdot n^3 \sum_{i=0}^{k-1} \left(\frac{1}{4}\right)^i && (7) \\ &= 2^k \cdot T\left(\frac{n}{2^k}\right) + \frac{4}{3} \cdot c \cdot n^3 \left(1 - \left(\frac{1}{4}\right)^k\right) && (8) \end{aligned}$$

Letting $k = \log_2(n)$ will allow the base case to be utilised for the recurrent term and provide the closed-form solution:

$$T(n) = \left(b - \frac{4}{3} \cdot c\right) n + \frac{4}{3} \cdot c \cdot n^3 = \Theta(n^3)$$

Q2

(a) A useful invariant could be that $s = \text{sum}(L[0..i-1])\%m$ at the beginning of loop i.

(b) On loop entry, $i = 0$. Therefore $L[0..i-1]$ is an empty list so its sum is zero. $0\%m = 0$ and s is initialised to 0. Therefore, the invariant is true on loop entry.

At the beginning of the i^{th} loop, we assume that the statement is true. This means that $s = \text{sum}(L[0..i-1])\%m$. This value is then updated in the following way:

$$\begin{aligned} s &= (s + L[i])\%m \\ &= (\text{sum}(L[0..i-1])\%m + L[i])\%m \\ &= (\text{sum}(L[0..i-1]) + L[i])\%m \\ &= \text{sum}(L[0..i])\%m \end{aligned}$$

The required value at the beginning of the $(i+1)^{\text{th}}$ loop for s under the invariant would need to be $s = \text{sum}(L[0..i])\%m$, which correctly matches the result above.

(c) Due to the usage of the `for` loop it is easy to argue that the loop will terminate, and specifically will do so at the terminating value of $i = n$. This means the value of s at the point when it breaks out of the loop will be $\text{sum}(L[0..n-1])\%m$. The term $L[0..n-1]$ represents the whole list, L! Therefore, if the function returns $s == 0$, it will return `True` if m divides $\text{sum}(L)$. Otherwise, it will return `False`. This is in line with the target goal and hence the algorithm is correct.

Q3

Answer: **A**, **E**

(A) Insertion sort best case is $O(n)$ when the input is a sorted list.

(B) Merge sort is always $O(n\log(n))$ for best, average and worst cases.

(C) Heap sort uses the heap data structure, which performs swap operations that cause it to no longer be stable.

(D) Selection sort will swap the newly found minimum with the current front of the unprocessed items. This means it will not be stable.

(E) Finally, selection sort will always spend $O(n)$ time to find and confirm the minimum and does so for each position, hence will take $O(n^2)$ every time.

Q4

For an array of size N, the median of medians algorithm is known to choose a "close-enough" median in time $O(N)$. In particular, the median of medians algorithm is known to choose a pivot that will always be within the 30th to 70th percentile of the data. Let's consider that the Quickselect algorithm is executed to find the k-th order statistic and that it uses the median of medians algorithm to pick its pivots. Let's analyse its time complexity:

- In the first level of the Quickselect recursion the time taken will be $O(N)$ and in the worst case there will be a recursive call with an input of size $0.7 \cdot N$.
- In the second level of the Quickselect recursion the time taken will be $O(0.7 \cdot N)$ and in the worst case there will be a recursive call with an input of size $0.7^2 \cdot N$.
- In the third level of the Quickselect recursion the time taken will be $O(0.7^2 \cdot N)$ and in the worst case there will be a recursive call with an input of size $0.7^3 \cdot N$.
- And so on.

Therefore, for this approximate median, the above terms of the time complexity will form a geometric series whose sum is $O(n)$.

For that matter, for any linear time pivot selection algorithm that guarantees that on every recursion level at least some fixed fraction of the array is not part of the input for the next recursion level, the resulting asymptotic time complexity of Quickselect would be the same.

Q5

The algorithm that uses calls to a linear time implementation of Quickselect proceeds as follows:

1. Use the Quickselect algorithm to find the value of the $\frac{N}{2}$-th order statistic, i.e., the median.
2. Use the Quickselect algorithm to find the value of the $\frac{4N}{5}$-th order statistic.
3. Use the previous two values to partition the list into three sets:
   (a) All candidates whose rank is smaller than or equal to the first value are unsuccessful.

(b) All candidates whose rank is bigger than the second value go straight to the third round.

(c) All remaining candidates progress to the second round of interviews.

Q6

Answers: $\Theta(V), \Theta(N(A)), \Theta(1), \Theta(V+E)$. For the last part note that one would need to search all outgoing edges of all vertices other than A.

Q7

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

Q8

Answer: **3**

The largest path from the root downwards in the BFS tree would be $A \rightarrow B \rightarrow C \rightarrow D$.

Q9

Answer: **3**, **6**, **10**

This is a slight modification of **Problem 1 of Week 7 Applied**.

To perform backtracking, you must analyse the DP array from the end. Without knowing the profit of each array house, the only way to determine which house must be sold is to understand where differences in optimal values occurred. When considering all houses, the max profit that could be made is 120. Notice that this is the same max value when considering up to House 10. This indicates that the introduction of House 11 and 12 did not change our optimal answer, hence we should exclude them (as we don't have access to the profits of each house and there are profits' arrays for which those houses would not be part of any optimal solution). On the other hand, when considering up to House 9 the profit is smaller than when considering up to House 10. Using the inclusion principle we can safely add House 10 to the solution (no matter what is the profits' array - which we don't have access to - there is an optimal solution which includes House 10). In short, the house that was chosen is the one where the optimal DP array value is different from the cell preceding it. However, since we know House 10 was chosen, we must **not** consider House 8 and 9 due to the rule, and must shift our gaze to Houses 7 and below. Repeating the process will enable you to reach the answer listed.

Q10

Answer: **B**

Performing the algorithm for one outer iteration, each vertex will have the following distance estimates: S) 0, A) -2, B) -1, C) -4, D) -3, E) -5, F) -4.

Q11

Answer: **B**, **D**

(A) There is no negative cycle since there are no negative diagonal entries.

(B) There is a path from vertex A to vertex G as well as a path from vertex G to vertex A according to the table since they are not "None". This implies there is a cycle that involves these two vertices.

(C) There is a path from vertex B to E with a distance of 49 according to the table, but it does not guarantee that it was a direct edge since the path could have traversed through other vertices before reaching E.

(D) Finally, the table clearly suggests a path from vertex C to F with a distance of -6.

Q12

Answer: **7**

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

Q13

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

When the Ford-Fulkerson execution terminates, the vertices s and a are the only ones that are still reachable from s in the residual network. Therefore the vertices on the S partition of the mincut will be vertices s and a. The edges that cross from S to T are $s \to b$ and $a \to c$, resulting in a cut value of 7 (as expected).

Q14

Answer: **A**

Neither problem has a feasible solution. For the first problem, when the graph is reduced into a flow graph, the result of Ford-Fulkerson algorithm would show that it is not possible to get all outgoing edges of the source/all incoming edges of the sink saturated. Hence, the circulation with demands problem has no feasible solution. The second problem is easier to eliminate, as in any feasible solution for a circulation with demands problem, the sum of demands must equate to **zero**. This is violated, hence we can immediately rule it out.

Q15

Answer: **A**, **D**

(A) Deleting 10 would not cause any node to have a height imbalance of greater than or equal to 2.

(B) Deleting 30 would create a height imbalance at Node 25, requiring rotations to keep the tree balanced.

(C) The AVL tree is currently balanced.

(D) Inserting 12 would place it as the left child of 15, creating a height imbalance at Node 10, requiring rotations.

(E) Inserting 88 as the right child of Node 75 would not cause any imbalances.

Q16

Answer: **A**, **B**, **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) In the case of an unsorted array, the search operation to check if the element is already there (in order to either insert or update the entry) would take time $O(N)$ in the worst case.

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

(D) 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.

Q17

Answer: $\Theta(N^2 \cdot M), \Theta(N \cdot M)$.

A suffix trie will contain $\Theta(N^2)$ nodes and each node will hold $\Theta(M)$ space. A suffix tree condenses the trie so that there will be $\Theta(N)$ nodes instead, however each node will still hold the same amount of space.

Q18

The current rank array gives the relative ordering for comparisons based on the first 2 characters. For suffix ID 5 and 7, the rank value for both is 2. This means they have the same first two characters. Their relative rank after sorting on the first 4 characters must therefore be drawn from the comparison between their last two characters respectively. In prefix doubling, since the ranks are based on $k = 2$ sorted characters, we can look at suffix ID $5 + k = 5 + 2 = 7$ and $7 + k = 7 + 2 = 9$ to compare the second half of the 4-letter sorting. The rank for ID 7 is 2 whilst the rank for ID 9 is 5. Therefore, after sorting on the first 4 characters, ID 5 must have a lower rank than ID 7. All of this is done in $O(1)$ since it only involves two integer comparisons.

Q19

The question mentions connecting all key locations and only having one possible path between any pair of locations, in other words the solution should be a tree. Since we want to maximise the importance of the selected roads, this is a problem clearly about constructing a maximum spanning tree. Consider all key locations as vertices on the graph and all potential road reconstructions as edges with weights equal to their importance (the cost value is only used to break ties: whenever there is a tie on the importance value, the edge with the smallest cost is prioritised in the approaches described below). To find the maximum spanning trees there are a few options:

- Run a modified Prim's algorithm in which a maxheap is used instead of a minheap.
- Run a modified Kruskal's algorithm in which edges are processed in decreasing order of weights.

- Change each edge weight to its negative and then run a standard minimum spanning tree algorithm such as Prim's or Kruskal's.

Q20

This is a classic problem called "Unweighted Interval Scheduling" and it can be solved using a greedy algorithm after the requests are sorted by finishing time. This is **Problem 7 of Week 6 Applied**, check the detailed solution there. The sorting part takes time $O(N \log N)$ while the greedy part takes $O(N)$, for a total time of $O(N \log N)$.

Alternatively, it is possible to use a dynamic programming approach that actually solves the more general "Weighted Interval Scheduling" problem in which each request has a weight and one wants to maximise the total weight of the compatible set of requests that is chosen (the unweighted version is just equivalent to setting all weights to 1). It is key to understand the optimal substructure that allows this problem to be efficiently solved. Firstly, the N requests should be sorted based on finishing time in ascending order. This would require $O(N \log(N))$ work. We will define the DP table where $\mathsf{DP}[i]$ refers to the maximum weight that can be achieved when considering the sorted $\mathsf{requests}[1..i]$. The optimal substructure can be constructed from the consideration of two choices. For any request being considered, you can either choose to accept it or ignore it. If you ignore it, then the answer is the same as $\mathsf{DP}[i-1]$, but if you choose to accept it, you must use binary search to figure out the biggest $j < i$ such that $\mathsf{requests}[j]$ finishes before $\mathsf{requests}[i]$ starts (so that they are compatible). This process would take $O(\log(N))$ and since the table size is N, the overall DP algorithm will take $O(N \log(N))$ work as well.

Q21

This allocation problem can be solved by modelling it as bipartite matching/max-flow problem:

- Since we are assigning students to topics, we can create a bipartite graph set up where we have all 123 students as nodes on one side and all 25 topics as nodes on the other side.
- Connect a source node to each student node with an edge with capacity of 1, representing that students can be allocated to at most one topic.
- Connect all the topic nodes to a sink node with an edge with capacity of 5, representing that topics are allowed to have 5 students assigned to it.
- For every student, connect their respective node to their preferred topic nodes. Each student will have up to 4 outgoing edges. Each of these preference edges will have a capacity of 1.

- Once created, this graph can simply be passed to the Ford-Fulkerson algorithm to efficiently solve the allocation problem, maximising student satisfaction. The optimal assignments would be the saturated preference edges. This uses the fact that when all capacities are integer numbers, then Ford-Fulkerson returns an integer-valued flow.

Q22

Refer to the solution of **Problem 5 of Week 3 Applied** for an explanation to this problem.