1. • Give an asymptotically tight bound for the following equation:

$$T(n) = T(n-2) + n^2, n > 4 \text{ and } T(n) = 1, 1 \le n \le 2$$

Solution: If we draw out a recursion tree, there are $n/2$ layers and each layer is doing $O(n^2)$ work, thus the total upper bound is $O(n^3)$. To see that total work is also $\Omega(n^3)$ we note that the work at each of the first $n/4$ levels is at least $(n/2)^2$. Thus the total work is $\Theta(n^3)$.

■

• $a_1, \cdots, a_k$ are all at most h digit binary numbers.

- Give an asymptotic upper bound on the number of digits of $a_1 a_2 \cdots a_k$

Solution: The number of bits required to represent a number n (ie. the number of digits in a binary number) is equal to $\lceil \log_2 n \rceil$. We drop the ceiling since we are only interesting in an asymptotic estimate and assume all logarithms below are to base 2..

Thus, the number of digits in $a_1 a_2 \cdots a_k = \log(a_1 a_2 \cdots a_k) = \log a_1 + \cdots + \log a_k$. Since each a_i has at most h digits, it means $\log a_i \le h$. Thus the total number of bits to represent the product is $O(hk)$. ■

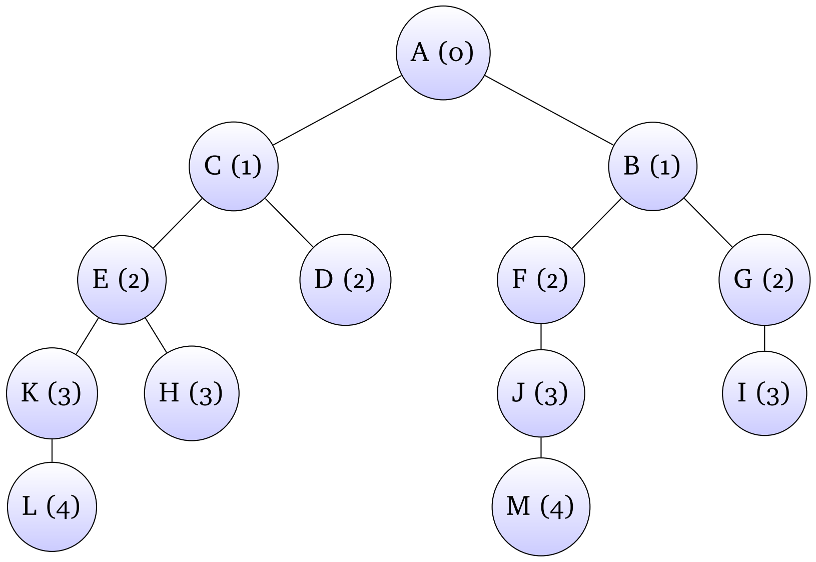
- Give an asymptotic upper bound on the number of digits of 2^{a_1}

Solution: The number of digits in $2^{a_1} = \log 2^{a_1} = a_1$. $a_1 \le 2^h$ since it has h digits, so the answer we are looking for is $O(2^h)$. ■

- Give an asymptotic upper bound on the number of digits of $a_1^{a_2}$ (alternate question from conflict exam)

Solution: The number of digits is $\log(a_1^{a_2}) = a_2 \log a_1 \le 2^h h$. ■

2. **Solution:** (a) Here is one of many possible trees. The distances from A are noted in parentheses.



(b) Let d be the distance between s and u. By our assumptions, d is also the distance between s and v. Since the BFS tree rooted at s will contain the shortest paths from s

to each vertex, u and v are both at level d of T. Note that this also means that the edges (u, v) is not in T.

Let w be the lowest common ancestor of u and v, and let d' be the level of T containing w. Since w is an ancestor of u, we can create a path P from u to w by repeatedly traveling to the parent of the current node. Similarly, we can form a path Q from v to w by repeatedly traveling to the parent of the current node. These two paths cannot intersect except at w, since if they did, their intersection would be a common ancestor of u and v at a lower level than w. Neither of these paths can contain (u, v), since (u, v) is not in T.

Thus, we can form a cycle involving the edge (u, v) by following Q from v to w, following P backwards from v to u, and finally following (u, v) back to v. The length of this cycle will be $2(d - d') + 1)$. Since $2(d - d')$ is clearly even, $2(d - d') + 1$ is odd, and thus we have found an odd-length cycle containing (u, v) and proven that such a cycle must exist.

■

3. (a) **Solution:** We solve this question using a DP algorithm. First we root T at any arbitrary vertex. Let the root be r. For any vertex v, let T_v denote the subtree rooted at v. Let $C(v)$ denote the set of children of v, and let $G(v)$ denote the set of grand children of v. For any vertex v, let $\text{VC}[v]$ denote the minimum weight of a vertex cover of T_v.

We have two choices, either to include v or not, and we can take the better of the two choices. If the vertex v is included, then the edges between v and its children are covered by v and thus we only need to include, on top of v, the minimum vertex cover of T_u for each child u of v. If v is not included, then the edges between v and $C(v)$ have to be covered by the children and thus we have to include all of $C(v)$. Since $C(v)$ also covers all edges between $C(v)$ and $G(v)$, we only need to include, on top of $C(v)$, the minimum vertex cover of T_x for any grand child x of v. Therefore, we have the following recurrence:

$$\text{VC}[v] = \min\{w(v) + \sum_{u \in C(v)} \text{VC}[u], \sum_{u \in C(v)} w(u) + \sum_{x \in G(v)} \text{VC}[x]\}$$

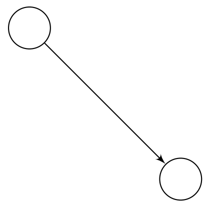
The base cases are $\text{VC}[v] = 0$ for any leaf v since there is no edge that needs to be covered and $\text{VC}[v] = \min\{w(v), \sum_{u \in C(v)} w(u)\}$ if v is the parent of any leaf. Note that these base cases are implicitly defined by the recursive case as the sum of any value over an emptyset is 0. The value we need to return is $\text{VC}(r)$. We can maintain an array of size n for the value we need to compute. Since the value for any vertex v depends on the values of its children and grand children, we can order the vertices in a post-order tree traversal and process the vertices in order. The above algorithm is summarized in the following pseudocode.

```
MinTreeVertexCover($T = (V, E)$, $w : V \to \mathbb{Z}_+$):
    pick arbitrary vertex $r$ and root the tree at $r$
    initiate VC to be an empty array of size $n$
    for vertex $v$ in the post-order traversal of $T$:
        if $v$ is a leaf:
            $\mathrm{VC}[v] = 0$
        else if $v$ is a parent of a leaf:
            $\mathrm{VC}[v] = \min\{w(v), \sum_{u \in C(v)} w(u)\}$
        else:
            $\mathrm{VC}[v] = \min\{w(v) + \sum_{u \in C(v)} \mathrm{VC}[u], \sum_{u \in C(v)} w(u) + \sum_{x \in G(v)} \mathrm{VC}[x]\}$
    return $\mathrm{VC}[r]$
```

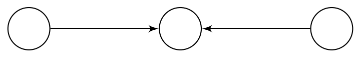
Let $n := |V|$, $m := |E|$. Since T is a tree, $m = O(n)$. Computing the post-order traversal after rooting the tree takes $O(m + n) = O(n)$ time as it can be done by a DFS from the root. The array has n entries and we may need to do $O(n)$ work for computing each entry, so a naive time bound for filling the table is $O(n^2)$. However, for any $v \in V$, $\mathrm{VC}(v)$ is only accessed at most twice when we compute the value for its parent and its grand parent. Therefore, the total work done for filling the array is $O(n)$. The running time of this algorithm is thus $O(n)$. ■

(b) **Solution:** The minimum weight vertex cover is $\{B, C, I\}$ with a weight of 7. ■

4. (a) **Solution:** i. Solutions vary, one possible solution is shown below. Note that if a DAG G can be made strongly connected by adding a single edge, then it must be the case that G has exactly one source and exactly one sink.



ii. Solutions vary, one possible solution is shown below. Remark that any DAG with more than one source or more than one sink is a valid solution.



■

(b) **Solution:** i. Let $G = (V, E)$ be a directed acyclic graph, and let $v_1, v_2, \ldots, v_n$ be a topological ordering of the vertices of G. Suppose there is some edge e we can add to G such that the resulting graph G' is strongly connected. If $n = 1$ G is trivially strongly connected so the answer is YES. We will assume that $n \geq 2$. A strongly connected graph cannot contain a source vertex or a sink vertex; in particular, v_1 must have positive in-degree in G' and v_n must have positive out-degree in G'. As v_1 and v_n respectively have in-degree and out-degree zero in G, it follows that the edge e must be (v_n, v_1).

To check if adding (v_n, v_1) to G results in a strongly connected graph G', we simply compute the meta-graph G'^{SCC} and check if G'^{SCC} consists of a single vertex. If so, then G' is strongly connected; otherwise, G' is not strongly connected. This yields the algorithm below.

```
CanMakeDAGStronglyConnected?($G = (V, E)$):
  If $|V| = 1$ return YES
  Compute a topological ordering $v_1, v_2, \ldots, v_n$ of $G$
  $E' \leftarrow E \cup \{(v_n, v_1)\}$
  $G' \leftarrow (V, E')$
  Compute the meta-graph $(G')^{\text{SCC}}$
  If $(G')^{\text{SCC}}$ consists of a single vertex:
      Return YES
  Else:
      Return NO
```

Computing a topological ordering of the vertices of G can be done in $O(|V| + |E|)$ time. Constructing the graph G' takes constant time. Computing $(G')^{\text{SCC}}$ and checking if $(G')^{\text{SCC}}$ consists of a single vertex can be done in $O(|V| + |E|)$ time. Hence the entire algorithm runs in $O(|V| + |E|)$ time.
An alternate algorithm is to simply count the number of sources and number of sinks and output YES if there is exactly one source and exactly one sink and NO otherwise.

ii. Observe that if G can be made strongly connected by the addition of a single edge, then the same is true of G^{SCC}. Conversely, suppose we can add a single edge e to G^{SCC} to make the resulting graph strongly connected. Denote this edge by $e = (S_1, S_2)$, where S_1 and S_2 are two strongly connected components of G. If we add a single edge in G from a vertex in S_1 to a vertex in S_2, then it is not hard to see that the resulting graph will be strongly connected. Thus, there is a single edge we can add to G which makes it strongly connected if and only if the same is true of G^{SCC}. As G^{SCC} is a DAG, we can simply compute G^{SCC} and run the algorithm from the previous part on G^{SCC}. This gives us the following algorithm.

```
CanMakeStronglyConnected?($G = (V, E)$):
  Compute the meta-graph $G^{\text{SCC}}$
  Return CanMakeDAGStronglyConnected?($G^{\text{SCC}}$)
```

Computing G^{SCC} can be done in $O(|V| + |E|)$ time. Note that G^{SCC} has at most $|V|$ vertices and $|E|$ edges, so the algorithm from part (i) will run in $O(|V| + |E|)$ time on G^{SCC}. Hence this algorithm runs in $O(|V| + |E|)$ time.

■

5. (a) **Solution:** We observe that if the first element of B is smaller than the first element of C, then it will be smaller than every element of C. B's first element does not contribute to the number of inversions count in this case. This element can be removed from B without affecting the answer.

On the other hand, if B's first element is larger than C's first element, then every element of B is larger than C's first element. We can add the size of B to the count, and delete the first element from C.

This leads to the following recursively algorithm. We start with the given arrays B and C, and based on the relation between the first elements of B and C, remove the first element of either B or C, adding the corresponding value to the number of inversions count. This process continues until all elements of either array have been deleted. Clearly, the count value for this final (base) case, when at least one array has zero elements, is 0.

Here's the pseudo code for this algorithm:

```
NumInversions($B[i_1 \dots j_1], C[i_2 \dots j_2]$):
    $n_1 = j_1 - i_1 + 1$
    $n_2 = j_2 - i_2 + 1$
    if ($n_1 \le 0$ or $n_2 \le 0$):
        return 0
    if ($B[i_1] < C[i_2]$):
        return NumInversions($B[i_1 + 1, j_1], C[i_2, j_2]$)
    else:
        return $n_1$ + NumInversions($B[i_1, j_1], C[i_2 + 1, j_2]$)
```

Note that this algorithm has tail recursion, hence can be easily converted to an iterative algorithm.

Time analysis: The time taken by the algorithm satisfies the recurrence:

$$T(n,m) = \begin{cases} O(1) & n = 0 \text{ or } m = 0 \\ \max\{T(n-1,m), T(n,m-1)\} + O(1) & \text{otherwise} \end{cases}$$

We can prove by induction, for example on the variable $n+m$, that the solution to this recurrence is $T(n,m) = O(n+m)$.

■

(b) **Solution:** As suggested in the hint, we try to modify the MergeSort algorithm to count the number of inversions. We split the input array into two halves (call them first sub-array and second sub-array), similar to MergeSort and count the number of inversions in each sub-array and also sort them. The necessity for sorting shall be explained later. Now, we need to come up with a way to compute the number of inversions in the input array, given the number of inversions in the two halves. For an inversion, we need to choose two indices, i and j, such that $i < j$ and $A[i] > A[j]$. We have the following cases:

- i is in second sub-array: Since $j > i$, j is also in second sub-array. This would be counted as an inversion in the second sub-array.
- i is in first sub-array and j is in first sub-array: This would be counted as an inversion in the first sub-array.
- i is in first sub-array and j is in second sub-array: This doesn't get counted anywhere and needs to be accounted for now.

The third case inversions can be counted during the Merge phase of the algorithm, by calling the algorithm in part (a) over sorted arrays. This is why we need to sort the sub-arrays before. Here's the pseudocode for this algorithm:

```
GetInversionCountAndSort($A[1...n]$):
    if($n <= 1$):
        return 0
    split $A$ into two equal sub-arrays $A[1...n/2]$ and $A[n/2+1..n]$
    numInversions = GetInversionCountAndSort($A[1...n/2]$) +
                    GetInversionCountAndSort($A[n/2+1...n]$) +
                    NumInversions($A[1...n/2], A[n/2+1...n]$)
    Merge the sorted arrays A[1...n/2] and A[n/2+1...n] and store in A[1...n]
        return numInversions
```

In the algorithm, we have two recursive calls, each of size $n/2$. The merge step, as in MergeSort takes O(n) time. By the implementation for Part(a), the NumInversions call would take $O(n/2+n/2) = O(n)$ time. Thus, the runtime recurrence here would be $T(n) = 2T(n/2) + O(n)$. This is exactly the same as the recurrence for MergeSort and yields a runtime of $O(nlogn)$. ■

6. **Solution:** We give two solutions. The first solution uses the fact that the length of the shortest path from s to t that visits a grocery store (and visits a gas station if necessary) is equal to

$$\min \begin{cases} \min\limits_{\substack{x\in X \\ d(s,x)+d(x,t)\le R}} (d(s,x)+d(x,t)) & \text{(Alice can reach Bob's without getting gas)} \\ \min\limits_{\substack{x\in X, y\in Y \\ d(s,x)+d(x,y)\le R}} (d(s,x)+d(x,y)+d(y,t)) & \text{(grocery store then gas station)} \\ \min\limits_{\substack{x\in X, y\in Y \\ d(s,y)\le R}} (d(s,y)+d(y,x)+d(x,t)) & \text{(gas station then grocery store)} \end{cases}$$

where $d(u,v)$ is the shortest path from u to v in G. We use the convention that $\min \emptyset = \infty$. We also assume there is always a gas station within distance R from s, which implies that the minimum is finite.

To compute this distance, we run Dijkstra's algorithm from every vertex to find the shortest path distance between every pair of vertices. This would take $O(nm + n^2 \log n)$ time. Then to compute the minimum above, it would require $O(n^2)$ time, leading to an overall running time of $O(nm + n^2 \log n)$.

One could do slightly better by computing $d(u,y)$ for all $u \in Y$ by running Disjkstra's on the graph G with the direction of each edge reversed. Therefore, to compute the minimum above, we would only have to run Dijkstra's once for distances from s, once for distances to t, once for every vertex in X and once for every vertex in Y. Let $k = |X| + |Y|$. As computing the minimum takes $O(|X| \cdot |Y|)$ time, this would lead to an overall running time of $O(km + kn \log n + |X| \cdot |Y|)$. Note that this algorithm is better than the previous algorithm when $k = o(n)$.

The high-level idea of the second solution is to construct four copies of G, one corresponding to the case where we haven't reached a gas station or a grocery store yet, one for the case where we have only reached a gas station but no grocery store, one for the case where we have reached a grocery store but haven't reached a gas station yet, and a final one for the case where we have reached both a gas station and a grocery store. For the copies that have yet to reach a gas station, we remove all vertices beyond distance R from s. This is because without going to a gas station, we cannot travel to any vertex that has distance greater than R from s. To do this, we first run Dijkstra's algorithm on the input graph G to find the set of vertices that are within distance R of s. Let V_{close} denote this set. Next, we define our new graph G'.

The graph we create is $G' = (V', E')$ where V' and E' and the edge weights ℓ' are defined as

- $V' = (V_{close} \times \{\mathsf{neither}, \mathsf{grocery}\}) \cup (V \times \{\mathsf{gas}, \mathsf{both}\})$ where V_{close} is the set of vertices within distance R from s
- $E' = E_1 \cup E_2 \cup E_3$ where
  - $E_1 = \{(u,a)(v,a) : uv \in E \text{ and } a \in \{\mathsf{neither}, \mathsf{grocery}, \mathsf{gas}, \mathsf{both}\}\}$, where we only consider $uv \in E$ when both endpoints of the edge are in the vertex set of this

copy. In other words, only when $(u,a),(v,a) \in V'$. E_1 corresponds to the edges that are also present in our original graph and do not traverse the four copies of G', but rather stay within the same copy.

 - $E_2 = \{(x,\mathsf{neither})(x,\mathsf{grocery}) : x \in X\} \cup \{(x,\mathsf{gas})(x,\mathsf{both}) : x \in X\}$. E_2 corresponds to the edges that take us from a copy where we hadn't visited a grocery store yet to a copy where we have. Traveling on an edge in E_2 corresponds to visiting a grocery store.
 - $E_3 = \{(y,\mathsf{neither})(y,\mathsf{gas}) : y \in Y\} \cup \{(y,\mathsf{grocery})(y,\mathsf{both}) : y \in Y\}$. E_3 corresponds to the edges that take us from a copy where we hadn't visited a gas station yet to a copy where we have. Traveling on an edge in E_3 corresponds to visiting a gas station.

- For any $a \in \{\mathsf{neither},\mathsf{grocery},\mathsf{gas},\mathsf{both}\}$, if $e = (u,a)(v,a) \in E_1$, then $\ell'(e) = \ell(uv)$. In other words, we do not change the edge weights in the four copies of the original graph.
- If $e \in E_2 \cup E_3$, then $\ell'(e) = 0$. Visiting a grocery store or a gas station when we're already there incurs no cost to us and therefore the cost of traversing these edges has to be 0.

Now notice that there are two cases. Either we can get from s to a grocery store and then to t without traveling a distance greater than R and therefore without visiting a gas station, or we have to visit a gas station to refill our tank because going from s to any grocery store and then to t has length greater than R for any path.

Therefore, we can now run Dijkstra's algorithm on our new graph G' to compute the distance from $(s,\mathsf{neither})$ to any other vertex. In particular, if we are in the first case and we can get from s to a grocery store and then to t without traveling a distance greater than R, the shortest path from $(s,\mathsf{neither})$ to $(t,\mathsf{grocery})$ in G' will have length at most R, meaning it will be the optimal solution to our problem.

On the other hand, if we have to visit a gas station to refill our tank because going from s to any grocery store and then to t has length greater than R for any path, the shortest path from $(s,\mathsf{neither})$ to $(t,\mathsf{grocery})$ in G' will have length greater than R. Then, the optimal solution to our problem corresponds to the shortest path from $(s,\mathsf{neither})$ to (t,both), which again is computed by our run of Dijkstra's algorithm in G'.

Formally, the algorithm proceeds as follows.

```
ALGORITHM:
  Run Dijkstra's algorithm on $G$ starting from $s$ to compute $V_{close}$, which is the set of vertices
          that are within distance $R$ from $s$
  Construct $G'$ as described above
  Run Dijkstra's on $G'$ starting from $(s,\mathsf{neither})$
  if $d_{G'}((s,\mathsf{neither}),(t,\mathsf{grocery})) \le R$
      return $d_{G'}((s,\mathsf{neither}),(t,\mathsf{grocery}))$
  else
      return $d_{G'}((s,\mathsf{neither}),(t,\mathsf{both}))$
```

For the running time, we note that $|V'| = O(n)$ and $|E_1| = O(m)$. As E_2 and E_3 connect copies of G, it follows that $|E_2 \cup E_3| = O(n)$. Therefore, we have $|E'| = O(m)$. It then easily follows that we can construct G' in $O(m+n)$ time. The running time of Dijkstra's on G and

G' dominates the running time of the algorithm. That is, the running time is $O(m+n\log n)$, if one uses a Fibonacci heap as the implementation for Dijkstra's, and $O(m\log n)$ time if one uses a binary heap.

Next, we justify the correctness of the algorithm. We want to show that the distance returned by the algorithm is the distance of the shortest path from s to t in G that visits at least one grocery store.

We distinguish between two cases. First, consider the case where Alice can reach a grocery store and then Bob's house without visiting a gas station. This implies that the grocery store $x \in X$ on the optimal shortest path and Bob's house are both within distance R from s, implying $t, x \in V_{close}$. Let P be the shortest path returned by our algorithm, and suppose now that there was another path P' that had smaller length than P. Notice that, by our construction of G', to go from $(s, \mathsf{neither})$ to $(t, \mathsf{grocery})$, we only traversed one edge in E_2 (i.e. we only visited one grocery store). Therefore, while P can contain several edges from E_1, it only contains one edge from E_2. Since in P' we also have to go from $(s, \mathsf{neither})$ to $(t, \mathsf{grocery})$ as we have to visit a grocery store at some point, and P' has smaller length than P, P' has to be "better" than P in at least one of the two segments of the path; from s to the grocery store or from the grocery store to t. Therefore, P' either finds a shorter path from $(s, \mathsf{neither})$ to a vertex $(u, \mathsf{neither})$, where $u \in X$ is a grocery store, or it finds a shorter path from $(u, \mathsf{grocery})$ to $(t, \mathsf{grocery})$. In either case this contradicts the fact that Dijkstra's algorithm finds the shortest path between two vertices in a graph, and therefore P is an optimal solution.

Second, we consider when Alice cannot reach Bob's house without visiting a gas station. Again, let P be the shortest path returned by our algorithm, and suppose now that there was another path P' that had smaller length than P. Notice that, by our construction of G', to go from $(s, \mathsf{neither})$ to (t, both), we only traversed one edge in E_2 (i.e. we only visited one grocery store) and also one edge in E_3 (i.e. we only visited one gas station). Therefore, while P can contain several edges from E_1, it only contains one edge from E_2 and one edge from E_3. Assume that we first visit the gas station and then the grocery store. The analysis in the other case is similar. Since in P' we also have to go from $(s, \mathsf{neither})$ to (t, both) as we have to visit both a grocery store and a gas station at some point, and P' has smaller length than P, P' has to be "better" than P in at least one of the now three segments of the path; from s to the gas station, from the gas station to the grocery store, or from the grocery store to t. Therefore, P' either finds a shorter path from $(s, \mathsf{neither})$ to a vertex $(u, \mathsf{neither})$, where $u \in Y$ is a gas station, or a shorter path from (u, gas) to (v, gas), where $v \in X$ is a grocery store, or a shorter path from (v, both) to (t, both). In any case, this again contradicts the fact that Dijkstra's algorithm finds the shortest path between two vertices in a graph, and therefore P is an optimal solution. ■