Question 1:

1. False. A bipartite graph cannot contain cycle of odd length, however the number of cycles could be odd. For instance, consider the graph with the edge set {(a,b),(b,c),(c,d),(d,a)}. The number of edges is 1, which is odd.

2. True. A tree by definition does not contain cycles, and hence does not contain cycles of odd length and is bipartite.

3. True. Let T_1 and T_2 be two distinct MSTs. Hence there exists an edge e that is in T_1 but not T_2. Add e to T_2 thereby inducing a (unique) cycle C. There are at least two edges of maximum weight in C (for if there were a unique maximum weight edge in C, then it cannot be in an MST thereby contradicting the fact that it is either in T_1 or T_2).

4. True. The claim in question follows from the following two facts: (i) the weight of a tree is linear in its edge weights and (ii) every spanning tree is has exactly |V|-1 edges. Hence the weight of a spanning tree under the modification goes from w(T) to 2w(T). Thus an MST of the original graph remains an MST even after the modification.

5. False. The max element in a min heap with n elements is one of the leafs. Since there are Theta(n) leafs and the structure of min heap does not carry any information on how the leaf values compare, to find a max element takes at least Omega(n) time.

6. False. Consider for instance f(n) = n and g(n)=n^2. Clearly n+n^2 is not in Theta(n).

7. True.

8. True. The number of spanning trees of a completely connected graph with n vertices is n^(n-2) (Cayley's formula) which grows at least exponentially in n. Even without the knowledge of Cayley's formula, it is easy to derive Omega(2^n) lower bound.

9. True. If the edge lengths are identical, then BFS does find shortest paths from the source. BFS takes O(|V|+|E|) where as Dijkstra even with a Fiboannci heap implementation takes O(|E| + |V| log |V|)

10. False. The O() notation merely gives an upper bound on the running time. To claim one is faster than other, you need an upper bound for the former and a lower bound (Omega) for the latter.

Question 2:

Algorithm1:

1. Construct the adjacent list of G^{rev} in order to facilitate the access to the incoming edges to each node.

2. Starting from node t, go through the incoming edges to t one by one to check if the following condition is satisfied:

$$\delta(s,t) == e(u,t) + \delta(s,u),\ \ (u,t) \in E$$

3. There must be at least one node u satisfying the above condition, and this u must be a predecessor of node t.
4. Keep doing the same checking operation recursively on the predecessor node to get the further predecessor node until node s is reached.

Algorithm2:

Run a modified version of Dijkstra algorithm without using the priority queue. Specifically,

1. Set S={s}.
2. While (S does not include t)
   Check each node u satisfying $u \notin S,\ (v,u) \in E,\ v \in S$, if the following condition is satisfied:
   $$\delta(s,u) == e(v,u) + \delta(s,v)$$
   If yes, add u into S, and v is the predecessor of u.
   endWhile

Complexity Analysis:

In either of the above algorithms, each directed edge is checked at most once, the complexity is O(|V|+|E|).

Question 3:

Algorithm:
Sort jobs by starting time so that $s_1 \leq s_2 \leq \cdots \leq s_n$.

Define $f_1, f_2, \cdots, f_n$ as the corresponding finishing times of the jobs.
Define d as the number of allocated processor.

Initialize $d = 0$;

For $j = 1\ to\ n$
   If (job j is compatible with processor)
      Allocate processor k to job j.
   Else
      Allocate a new processor $d + 1$;
      Schedule job j to processor $d + 1$;

Update $d = d + 1$;
EndFor

Implementation:
Sort starting time takes time $O(n \log n)$.

Keep the allocated processors in a priority queue (Min-heap), accessed by minimum finishing time.
For each processor k, maintain the finishing time of the last job added.
To check the compatibility: compare the job j's starting time with the finishing time (key value of the root of the Min-heap) of processor k with last allocated job i. If $s_i > f_i$, allocate processor k to job j; otherwise, allocate a new processor. The operation corresponds to either changing key value or adding a new node. Either operation takes time $O(\log n)$.

For n jobs, the total time is $O(n \log n)$.

Proof:
Definition: define the "depth" d of a set of jobs to be the maximum number that pass over any single point on the time-line.
Then we have the follow observations: All schedules use $\geq d$ (the depth) processors

- Assume that greedy uses D processors where $D > d$.
- Processor D is used because we needed to schedule a job, say j, that is incompatible with all $D - 1$ other processors.
- Since we sorted by starting time, all these incompatibilities are caused by jobs with start times $\geq s_j$
- The number of such incompatible processors is at most $d - 1$.
- Therefore, $D - 1 \leq d - 1$, a contradiction

Question 4:

i) The mayor is merely contending that the graph of the city is a strongly-connected graph, denoted as G. Form a graph of the city (intersections become nodes, one-way streets become directed edges). Suppose there are n nodes and m edges. The algorithm is:
(1) Choose an arbitrary node s in G, and run BFS from s. If all the nodes are reached the BFS tree rooted at s, then go to step (2), otherwise, the mayor's statement must be wrong. This operation takes time $\mathrm{O}(\mathrm{m} + \mathrm{n})$

(2) Reverse the direction of all the edges to get the graph G^{inv}, this step takes time $O(m+n)$

(3) Do BFS in G^{inv} starting from s. If all the nodes are reached, then the mayor's statement is correct; otherwise, the mayor's statement is wrong. This step takes time $O(m+n)$.

Explanation: BFS on G tests if s can reach all other nodes; BFS on G^{inv} tests if all other nodes reach s. If these two conditions are not satisfied, the mayor's statement is wrong obviously; if these two conditions are satisfied, any node v can reach any node u by going through s.

ii) Now the mayor is contending that the intersections which are reachable from the city form a strongly-connected component. Run the first step of the previous algorithm (test to see which nodes are reachable from town hall). Remove any nodes which are unreachable from the town hall, and this is the component which the mayor is claiming is strongly connected. Run the previous algorithm on this component to verify it is strongly connected.

Question 5:

No, the modification does not work. Consider the following directed graph with edge set {(a,b),(b,c),(a,c)} all of whose edge weights are -1. The shortest path from a to c is ((a,b),(b,c)) with path cost -2. In this case, w=1 and if 2 is added to every edge length before running Dijkstra starting from a, then Dijkstra outputs (a,c) as the shortest path from a to c which is incorrect.

Another way to reason why the modification does not work is that if there were a directed cycle in the graph (reachable from the chosen source) whose total weight is negative, then the shortest paths are not well defined since you could traverse the negative cycle multiple times and make the length arbitrarily small. Under the modification, all edge lengths are positive and hence clearly the shortest paths in the original graph are not preserved.

Remark: Many students who claimed that the modification works made this common mistake. They claimed that since the transformation preserved the relative ordering of the edges by their weights, the algorithm should work. This is incorrect. Even though the edge lengths are increased by the same amount, the path lengths change as a function of the number of edges in them.

Question 6:

a) We use the $\left\lceil \frac{n}{2} \right\rceil$ smaller elements to build a max heap. We use the remaining $\left\lfloor \frac{n}{2} \right\rfloor$ elements to build a min heap. The median will be at the root of max heap (We

assume the case of even *n*, median is $\frac{n}{2}th$ element when sorted in increasing order (5 pts).

Part (a) grading break down:

- Putting the smaller half elements in max heap and the other half in min heap (4 pts).
- Mention where the median will be at (1 pts).

b) For a new element *x*, we compare *x* with current median (root of max heap). If $x < median$ we insert *x* into max heap. Otherwise we insert *x* into min heap. If $size(maxHeap) > size(minHeap) + 1$, then we ExtractMax() on max heap and insert the extracted value into the min heap. Also if $size(minHeap) > size(maxHeap)$ we ExtractMin() on min heap and insert the extracted value in max heap. (6 pts).

Part (b) grading break down:

- Correctly identifying which heap to insert the value into (2 pts).
- Maintaining the size of the heaps correctly (4 pts).
- If you only got the idea that you have to maintain the size of the heaps equal but don't do it correctly (1 pt).

c) ExtractMax() on max heap. If after extraction $size(maxHeap) < size(minHeap)$ then we ExtractMin() on min heap and insert the extracted value in max heap (5 pts).

Part (c) grading break down:

- Correctly identifying which root to extract and retaining heap property (1 pt).
- Maintaining the size of heaps correctly (4 pts).
- If you only got the idea that you have to maintain the size of the heaps equal but don't do it correctly (1 pt).