1. (15 points) In the Fibonacci heaps data structure, a cut between a vertex \( u \) and its parent \( v \) causes a cascading cut at \( v \) if \( v \) has already lost a child since it last became a child of some other vertex. Suppose we change this, so that a cascading cut is performed at \( v \) only if \( v \) has already lost \textit{two} children. How does this change alter the lemma shown below (this lemma is from the analysis of the running time of Fibonacci heaps)? Explain your answer.

**Lemma.** Let \( x \) be any node in an F-heap. Let \( y_1, \ldots, y_r \) be the children of \( x \), in order of time in which they were linked to \( x \) (earliest to latest). Then, \( \text{rank}(y_i) \geq i-2 \) for all \( i \).

The inequality in the lemma becomes \( \text{rank}(y_i) \geq i-3 \). Since \( y_i \) had the same rank as \( x \) when it became a child of \( x \) and \( x \) must have had at least \( i-1 \) children at that time, \( y_i \) must have had rank of at least \( i-1 \) when it became a child of \( x \). Since it still is a child of \( x \), it can have lost at most two children since that time, so its rank must be at least \( i-3 \).

Let \( S_k \) be the smallest possible number of descendants that a node of rank \( k \) has, in our modified version of Fibonacci heaps. Give a recursive lower bound on \( S_k \). That is, give an inequality of the form \( S_k \geq f(S_0, S_1, \ldots, S_{k-1}) \) where \( f \) is some function of the \( S_i \)’s for \( i < k \).

Clearly \( S_0 = 1 \), \( S_1 = 2 \) and \( S_2 = 3 \). For \( k > 2 \), we can use the modified lemma to conclude that 
\[
S_k \geq 3 + S_0 + S_1 + \cdots + S_{k-3}. 
\]
Note that the difference between the bounds for \( S_k \) and for \( S_{k-1} \) is \( S_{k-3} \).

Use this to give a lower bound on the smallest number of descendants that a node with rank 6 can have.

\[
\text{From the above, we have } S_3 \geq 3 + S_0 = 4, \ S_4 \geq 4 + S_1 = 6, \ S_5 \geq 6 + S_2 = 9, \ S_6 \geq 9 + S_3 = 13.
\]
2. (15 points) Consider an execution of the breadth-first shortest path algorithm on the graph shown below, assuming that the source vertex is $a_1$.

What vertices are scanned in pass 1 of the algorithm? What vertices are scanned in pass 2? Pass 3? Pass 4? (Recall that pass 0 ends after the source vertex is scanned the first time and for $j>0$, pass $j$ ends when all the vertices on the queue at the end of the previous pass have been scanned).

Vertices $b_1$ and $c$ in pass 1,
vertices $a_2$, $d_1$, $d_2$, $d_3$ and $d_4$ in pass 2,
vertices $b_2$, $c$, $e_1$, $e_2$, $e_3$ and $e_4$ in pass 3,
vertices $a_3$, $d_1$, $d_2$, $d_3$ and $d_4$ in pass 4.

How many times is each edge of the form $(d_i, e_j)$ examined?

4 times each.

Explain how to generalize this example to show that there are graphs on $n$ vertices for which the breadth-first scanning algorithm takes $\Omega(n^3)$ time.

The example can be generalized by extending the chain at left to $2k-1$ vertices and increasing the size of the complete bipartite subgraph at right to $2k$ vertices. The $k$ vertices of the form $a_i$ will each have an edge to vertex $c$, with decreasing lengths, as in the above example. This will cause each edge in the bipartite subgraph to be examined $k$ times. Since the graph has $4k$ vertices and there are $k^2$ edges in the bipartite subgraph, the total number of times an edge is examined will be $\Omega(k^3)=\Omega(n^3)$.
3. (10 points) The figure below shows a graph in which we have used the edge length transformation described in the notes to eliminate negative edge lengths. The distances used to compute the transformation are shown along with the transformed edge lengths. What is the original length of the path $d-a-e-c$?

The length in the transformed graph is 6. To this, we add $-2$ and subtract $-6$ to get the original length, so the original length was 10.

What is the shortest path from $d$ to $c$? What is its length (using the original edge lengths)?

The shortest path is $d-a-e-b-c$. Its length in the transformed graph is 2, so its original length is 6.
4. (20 points) Let \( G \) be a directed graph in which each edge has some positive capacity (stored as the edge weight) and let \( t \) be a designated destination vertex. Let \( T \) be a reverse best bottleneck path tree with destination \( t \). That is, \( T \) is a spanning tree of \( G \) directed toward \( t \) in which all paths have the largest bottleneck capacity of any path joining the same two endpoints (recall that the bottleneck capacity of a path is the capacity of its smallest capacity edge). Let \( p(u) \) to be the first vertex (after \( u \)) on the tree path from \( u \) to \( t \). Let \( \text{secondBest}(u) \) be the first vertex (after \( u \)) on a second-best path from \( u \) to \( t \). More precisely, if \( v \) is \( \text{secondBest}(u) \), then \( v \neq p(u) \), \((u,v)\) is an edge and there is no other such vertex \( w \neq p(u) \), for which the best bottleneck path from \( u \) to \( t \) that starts with \( w \) is has a larger capacity than the best bottleneck path from \( u \) to \( t \) that starts with \( v \). Fill in the body of the C++ program shown below to compute \( \text{secondBest}(u) \) for all vertices \( u \neq t \). Your program should run in \( O(m) \) time, where \( m \) is the number of edges in \( G \).

```cpp
void secondBest(wdigraph G, vertex p[], int bc[], vertex sb[]) {
    // Return the second best next hop vertex of u in sb[u] for all vertices u.
    // p[u] is the parent of u in the shortest path tree and bc[u] is the capacity of the best path
    // from u to the destination. The destination is vertex n.
    vertex u, v; edge e; int sbcap;
    for (u = 1; u != G.n; u++) {
        sb[u] = Null; sbcap = 0;
        for (e = G.firstout(u); e != Null; e = G.nextout(u,e)) {
            v = G.head(e);
            if (v != p[u] && min(G.wt(e),bc[v]) > sbcap) {
                sb[u] = v; sbcap = min(G.wt(e),bc[v]);
            }
        }
    }
}
```

Describe in words, how you could generalize this, to allow one to easily determine the third best path to \( t \), the fourth best, etc. What is the running time for this algorithm?

*We could do this by sorting the adjacency lists, so that for vertex \( u \), the edge that starts the best bottleneck capacity path comes first, then the edge that starts the second best, then the edge that starts the third best, and so forth. In general, an edge \((u,v)\) will precede an edge \((u,w)\) if \(\min(cap(u,v),bc(v)) > \min(cap(u,w),bc(w))\). Using an \(O(\log n)\) algorithm to sort each of the adjacency lists gives a running time of \(O(m \log n)\).*
5. (10 points) An instance of the flow graph data structure with source \(a\) and sink \(f\) is shown below. The flow shown is a valid flow

What’s the magnitude of the flow, as shown? 8

Fill in the edges of the residual graph below, including their capacities.

Show how the flow graph data structure changes when we saturate the path \(a, e, c, b, f\). You may just mark the changes on the figure above.
6. (10 points) The figure below shows an instance of the max flow problem with a flow produced by the capacity scaling version of the augmenting path algorithm at an intermediate stage in the computation.

Draw the scaled residual graph, $R_\Delta$ for this flow ($\Delta=4$).

Draw $R_\Delta$ at the start of the next scaling phase ($\Delta=2$).
7. (10 points) The figure below shows an intermediate stage in the execution of the admissible path algorithm with dynamic trees. Modify the flow values on the graph so that they reflect the current flow (including the flows recorded implicitly in the dynamic trees data structure).

![Graph Diagram]

Vertex f is relabeled to 2, the edges entering f in the dynamic trees data structure are removed and the flow of 4 from g to f and the flow of 3 from e to f are recorded in the flow graph data structure.

Explain what happens in the next step of the algorithm.

*Vertex f is relabeled to 2, the edges entering f in the dynamic trees data structure are removed and the flow of 4 from g to f and the flow of 3 from e to f are recorded in the flow graph data structure.*
8. (10 points) The figure below shows an intermediate stage in the execution of the general preflow-push algorithm. In the worst-case, how many more steps are needed before the algorithm terminates? (Assume that whenever there is a choice to push flow to the left or the right, the algorithm pushes flow to the right.)

We would first relabel \( c \) to 3 and send one unit of flow back to \( b \),
then relabel \( b \) to 4 and send flow to \( c \),
then relabel \( c \) to 5 and send flow back to \( b \) then \( a \),
then relabel \( a \) to 5 and send flow to \( b \),
then relabel \( b \) to 6 and send flow to \( c \),
then relabel \( c \) to 7 and send flow back to \( b \), then \( a \),
then relabel \( a \) to 6 and send flow to \( s \).

So the total number of steps is 16.

What would the new distance labels be, if you recomputed them at this point? (Just show the distance labels on the figure.)

The labels for \( a \), \( b \) and \( c \) would be 6, 7 and 8 respectively.

How many more steps are needed if you run the algorithm from this state, using the modified distance labels?

3