Be neat and concise, but complete.

1. (5 points) An incomplete instance of the `wgraph` data structure is shown below. Fill in all the missing entries.

```
| firstedge | a | 5 |
|           | b | 3 |
|           | c | 8 |
|           | d | 9 |
|           | e | 1 |
|           | f | 10 |
```

```
<table>
<thead>
<tr>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>
```
2. (10 points) In the worst-case analysis of Prim’s algorithm, we saw that the number of calls to the changekey procedure was $O(m)$. Show that it is also $\Omega(m)$. More specifically, show that it is $\Omega(n^2)$. Do this by showing that for each value of $n$, there is a weighted graph with $\Omega(n^2)$ edges and that changekey is invoked when most of these edges are examined. Be sure to specify the edge weights. Draw a picture of the graph for the case of $n=5$.

The figure below shows a complete graph on 5 vertices. If we start Prim’s algorithm from vertex $u_1$, all the other vertices will be inserted into the heap during the initialization phase, and they will be removed from the heap in the order of their subscripts. On each iteration, changekey will be called on every edge incident to the current vertex connecting to a vertex with higher subscript.

The example generalizes directly to any $n$. The edge weight on edges of the form $\{u_i, u_{i+1}\}$ is 1. The weight of all other edges $\{u_i, u_j\}$ with $i < j$ is $n-i$. For the general case, changkey is called $(n-2) + (n-3) + \ldots + 1$ times, which is $\Omega(n^2)$. 

![Complete Graph on 5 Vertices](image-url)
3. (10 points) The figure below shows an incomplete representation of an intermediate state in the execution of the round-robin algorithm. In particular, the state of the partition data structure is not shown. Show the complete state of the algorithm after one more iteration. You should include the state of the partition data structure (show the sets in the partition data structure as trees). Also, circle all the nodes in the leftist heaps that should be considered “deleted”.

```
queue: f c g d
partition:

<table>
<thead>
<tr>
<th>g</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
</tr>
</tbody>
</table>

leftist heaps

```

[Diagram showing the leftist heaps and partition data structure]
4. (15 points) The minimum spanning tree algorithm shown below is similar to Prim’s algorithm, but in this case the heap stores all the edges with exactly one endpoint in the set of tree vertices.

```
procedure minspantree(graph G, set tree_edges);
    vertex u,v; set tree_vertices; heap S;
    tree_vertices := {1}; tree_edges := {};
    for {1,v} ∈ edges(1) ⇒ insert({1,v},cost(1,v),S); rof;
    do S ≠ {} ⇒
        {u,v} := deletemin(S); // assume u∈tree_vertices
        tree_vertices := tree_vertices ∪ {v};
        tree_edges := tree_edges ∪ {u,v};
        for {v,w} ∈ edges(v) ⇒
            if w ∈ tree_vertices ⇒ delete({v,w},S);
                | w /∈ tree_vertices ⇒ insert({v,w},cost(v,w),S);
            fi;
        od;
    end;
```

Explain why this algorithm is an instance of the general greedy method.

The algorithm maintains the invariant that the heap contains the edges that cross the cut defined by the set of tree_vertices and its complement. The minimum cost edge crossed by this cut is the one selected by the blue rule of the greedy algorithm. Edges not included in the tree are implicitly colored red. For each of these, the edge, together with the tree path joining its endpoints is a cycle to which the red rule can be applied.

Give an expression for the time required by all the deletemin operations, in terms of the number of vertices (n), the number of edges (m) and the heap parameter (d). Similarly, give an expression for the time required by all the insert operations and all the delete operations.

- total deletemin time: $O(nd\log_d n)$
- total insert time: $O(m\log_d n)$
- total delete time: $O(md\log_d n)$

What choice of $d$ gives the best overall running time? Why?

The overall running time is $O(md\log_d n)$. This is minimized by selecting a small value for $d$ (like 2).
5. (10 points) Problem set 1 discusses a min-max heap, a data structure that supports both a
\textit{findmin} operation and a \textit{findmax} operation. In the implementation discussed in the problem
set, nodes at even distances from the root have the smallest key value in their subtree, while
nodes at odd distances from the root have the largest key value in their subtree.

An alternative way implement a min-max heap is to store the items in two separate \textit{d}-heaps,
one of which is organized to support the \textit{findmin} operation and the other organized to
implement the \textit{findmax} operation. How does the space used in these two implementations
compare? Assume that the min-max heap is not required to provide a general \textit{delete}
operation or a \textit{changkey} oprationion, but is required to implement both \textit{deletemin} and
\textit{deletemax}.

The original min-max heap implementation requires 2\(n\) words of memory for a heap of size \(n\). It uses
\(n\) words to store the heap array and \(n\) words for the key values. The alternate implementation requires
5\(n\) words. This is because when we do a deletemin, we need to delete the item from both heaps and in
the “max” heap, it will not be at the root of the tree. Hence, we need an array to give us the position of
each node in the max heap. For the same reason, we need an array to give us the position of each node
in the min heap. The key values are the same in both heaps, so it’s not necessary to store them twice.

Give an asymptotic upper bound on the running time of the \textit{changekey} operation on this
alternate implementation of the min-max heap, in the case when the key increases in value.
Also, give an upper bound on its running time when the key decreases in value. How is the
running time affected by the value of \(d\) used in the underlying \(d\)-heaps? What is the best
choice for \(d\). Justify your answers.

The running time is \(O(d \log_dn)\) regardless of whether changkey increases or decreases. This is because
in both cases, we must do a siftup in one heap and a siftdown in the other. So the running time
increases with \(d\), meaning that a small value (like 2) is best.

Give an asymptotic upper bound on the running time of the \textit{insert} operation. How is the
running time affected by the value of \(d\) used in the underlying \(d\)-heaps? What is the best
choice for \(d\) from the perspective of the \textit{insert} operation? Justify your answers.

The running time is \(O(\log_dn)\), since inserting the item into both heaps requires a siftup operation.
The best choice of \(d\), from the perspective of the insert operation is \(n–1\) where \(n\) is the maximum heap
size, since in this case, the insert takes constant time. Of course, this is a poor choice for deletemin.
6. (10 points) Let \( n = n_1 + n_2 + \cdots + n_r \) and assume all \( n_i \) are positive. Give an upper bound on 
\[ \sum_{i=1}^{r} \sqrt[n_i]{n_i} \] in terms of \( n \) and \( r \). Explain your answer.

The upper bound is \( r \sqrt[n]{r} = \sqrt[n]{nr} \). This follows from the general upper bound discussed in problem set 1. Since the square root function grows more slowly than any linear function, we can get an upper bound by replacing each value in the sum by the average of the \( n_i \) values.

Give an upper bound on \( \sum_{i=1}^{r} n_i^{3/2} \). Explain your answer.

The upper bound is \( n^{3/2} \). This follows from a complementary argument to the one discussed in problem set 1. Since the function is grows faster than linearly, we can get an upper bound by making all but one of the \( n_i \) values equal to zero.
7. (15 points) Consider a version of the partition data structure that uses path compression, but does not use link-by-rank. In such a version, we can reduce the space consumed by the algorithm, by eliminating the rank variables. The link subroutine in this version would simply make the parent pointer of the first argument equal to the second argument. To analyze the performance of this version, it’s useful to define a function \( r \) which takes the role of the rank in the analysis. For each node \( x \), \( r(x) \) is initialized to zero. Then, during the link operation that makes node \( x \) the child of node \( y \), \( r(y) \) is assigned the value \( \max\{r(y), 1+r(x)\} \).

Define \( \text{level}(x) = \left\lfloor \log(1 + r(p(x)) - r(x)) \right\rfloor \). What is the largest value that \( \text{level}(x) \) can have?

\[ \log n \]

Assuming that the \( \text{credit} \) and \( \text{debit} \) variables used in the original analysis are defined and assigned values in the same way as in the original analysis, give a tight upper bound on the value for \( \text{fcredit} \) after \( m \) operations have been done. Justify your answer.

\( m \left( 1 + \log n \right) \). During a top level find, \( \text{fcredit} \) is implemented at most once for each different value of the level function. There are at most \( 1 + \log n \) values that the level function can take on, so \( \text{fcredit} \) can increase by at most \( 1 + \log n \) for each top level find.

Suppose that \( \text{debit}(x) \) is incremented during a \( \text{find} \) operation and that \( \text{level}(x) = i \) before the \( \text{find} \). If \( p(x) = w \) before the \( \text{find} \) and \( p(x) = z \) after the \( \text{find} \), how does the value of \( r(z) \) compare to \( r(w) \)? Explain.

\( r(z) \geq r(w) + 2^i - 1 \). Because \( \text{debit}(x) \) was incremented, it must have a proper ancestor \( y \) with \( \text{level}(y) = i \). By the definition of level, \( r(p(y)) - r(y) \geq 2^i - 1 \). Since \( r \) increases as you go up the tree, it follows that \( r(z) - r(w) \geq 2^i - 1 \).

How many times can \( \text{debit}(x) \) be incremented while \( \text{level}(x) = i \)? Why?

At most twice. Each time \( \text{debit}(x) \) is incremented while \( \text{level}(x) = i \), \( r(p(x)) \) grows by \( 2^i - 1 \). After this happens two times, \( \text{level}(x) \) must increase by at least 1. Also note that because \( r(p(x)) \) never decreases, neither does \( \text{level}(x) \).

Give an upper bound on the final value of \( \text{debit}(x) \). Explain.

\( 2 \left( 1 + \log n \right) \). Since level can take on at most \( 1 + \log n \) distinct values, and \( \text{debit}(x) \) can be incremented at most twice for each of these values it can never exceed \( 2 \left( 1 + \log n \right) \).