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   edges
   a  5                  1  a  3  7  9  e
   b  3                  2  d  0  1  7
   c                  3  4  6  8  a
   d  9                  4  c  7  3  0
   e                  5  1  3  2  d
   f  10                  6  c  0  1  0
                  7  c  6  8  0  f
                  8  0  1  4  c
                  9  e 10  6  5
                  10 2  2  6  e
```
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 $n=5$. 
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: d e f c g

partition:

leftist heaps
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.

```plaintext
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;
    rof;
  od;
end;
```

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

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.

What choice of d gives the best overall running time? Why?
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 $d$-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} operation, but is required to implement both \textit{deletemin} and \textit{deletemax}.

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.

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.
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.

Give an upper bound on \( \sum_{i=1}^{r} n_i^{3/2} \). Explain your answer.
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 \( \text{level}(x) = \lceil \log(1 + r(p(x)) - r(x)) \rceil \). What is the largest value that \( \text{level}(x) \) can have?

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 \( f_{\text{credit}} \) after \( m \) operations have been done. Justify your answer.
Suppose that $\text{debit}(x)$ is incremented during a find operation and that $\text{level}(x) = i$ before the find. If $p(x) = w$ before the find and $p(x) = z$ after the find, how does the value of $r(z)$ compare to $r(w)$? Explain.

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

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