These are solutions for the practice final. Please note that these solutions are intended to provide you with a basis to check your own answers. In order to get full credit on the exam, you must show all work and fully explain your answers. These solutions do not necessarily provide a full explanation. If you are confused about these answers, make sure to review the lecture slides or come to TA hours.

Problem 1
Solution:

1. $O(\log n)$, because at worst you will only need to downheap the height of the heap.

2. $O(n^2 \log n)$, because Kruskal's algorithm sorts the edges which requires $O(|E| \log |E|)$, which equals $O(n^2 \log n^2)$, which simplifies to $O(n^2 \log n)$ by the properties of logarithms.

3. $O(n)$, because you may need to iterate through the entire linked list.

4. $O(n)$, using radix sort.

5. $O(n \log n)$. You could do a breadth-first search starting from a random node and decorate each node as it is added to the tree. This visits every node and edge, so it is $O(|V| + |E|)$, or $O(n \log n)$

6. Since each node is visited once in a pre-order traversal, the runtime is $O(n)$.

Problem 2

Solution:

Using what we know about the order of binary search trees we can avoid searching sections of the tree that aren't within the k1, k2 range. If a node is less than our k1 key then we know all nodes to the left of this node will also be smaller and therefore don't need to be visited. Also, if a node value is higher than k2 we know we don't have to visit any children to the right of that node. Using this we can traverse the tree with a modified in-order traversal in which we only visit the left child if the current node is greater than k1 and only visit the right child if the current node is less than k2.

```
function traverseRange(tree, k1, k2)
""" Transforms: Tree, int, int -> array of TreeNodes
    Purpose: calls the traversalHelper to find and
    add nodes with keys in a given range (k1<k2) to
    the 'results' array
"""
    result = []
    if root is not null:
        traversalHelper(tree.root(), k1, k2, result)
    return result


function traversalHelper(current, k1, k2, result)
""" Transforms: TreeNode, int, int, array -> nothing
    Purpose: checks left and right children for range,
    traversing the graph in modified in-order.
    'result' may be modified
"""

    leftChild = current.leftChild()
    rightChild = current.rightChild()
    if leftChild is not null and current.key > k1:
        traversalHelper(leftChild, k1, k2, result)
    if current.key >= k1 and current.key <= k2:
        result.add(current.key)
    if rightChild is not null and current.key < k2:
        traversalHelper(rightChild, k1, k2, result)
```

This algorithm runs in $O(h + s)$ because you may need to travel down the entire height (h) of the tree, and you must visit every node (s) between k1 and k2 exactly once.

Problem 3

Solution:

We can simply run Kruskal's algorithm, but add the edges in F into our spanning tree before running Kruskal's algorithm. This way, all edges in F will be in our final tree T, and we are still adding all possible edges of minimum weight because Kruskal's will sort the remaining edges.

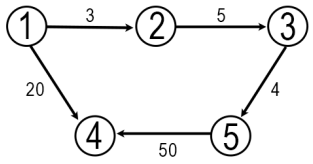
```
function modifiedKruskal(G):
  """ Transforms: graph G -> array ST
      Purpose: finds the lightest spanning tree such that all edges
      in F are included. Returns array of the edges in the spanning
      tree.
  """

  for vertices v in G:
      makeCloud(v)
  ST=[] //spanning tree
  for all edges (u,v) in G where inF(e) is true:
      add(u,v) to ST
      merge clouds containing u and v
  for all edges (u,v) in G where inF(e) is false, sorted by weight:
      if u and v are not in same cloud:
          add (u,v) to ST
          merge clouds
  return ST
```

This algorithm is $O(|E| \log |E|)$ because the slowest operation is sorting the edges not in F, which is $O(|E| \log |E|)$.

Alternatively, you could set all the edges in F to $-\infty$ and run Kruskal's normally, since would automatically choose them first.

Problem 4
Solution:

- Dijkstra's guarantees that when a vertex is dequeued from the minimum cost priority queue, it is decorated with the minimum cost to reach it. A maximum cost priority queue does not have the same guarantee since there may be unvisited nodes which would yield longer paths. Take the following example where 1 is the source and 4 is the target.



  We would terminate once 4 is dequeued from the priority queue and it is decorated with a cost of 20. However, the maximal path would go through 1, 2, 3, and 5.

- Perform a modified topsort on the graph. Set the cost of every node to $-\infty$ and the start node to 0. Push all sources onto the stack. While the stack is not empty, pop a node and examine all of its neighboring nodes. If the cost of the neighbor is less than the that of the node plus the weight of the edge between them, update the cost of the neighbor to this higher value. Delete the edge between the node and its neighbor. If the neighbor is now a source, push it on to the stack. Once the target node is popped from the stack, we know that its cost will not change since it must be a source to have been pushed on the stack. Thus, we can immediately return the target node's cost.

  The pseudocode looks something like this:

```
function longestPath(G, s, t):
    """ Transforms: graph G, start node s, target node t ->
                length of longest path between s and t
        Purpose: calculates the longest path between two tree nodes.
    """

    stack = Stack()
    for v in G.vertices():
        v.cost = -infinity
        if v is source:
            stack.push(v)
    s.cost = 0
    while stack not empty:
        v = stack.pop()
        if v == t:
            return v.cost
        for each edge (v, w):
            if w.cost < v.cost + weight(v, w):
                w.cost = v.cost + weight(v, w)
            delete edge (v, w)
            if w is source:
                stack.push(w)
    return infinity
```

Problem 5
Solution:
Runtime for Algorithm A:
$T(|V|) = T(|V|/2) + O(|V|^2)$
Recall the master theorem: $a = 1$, $b = 2$, $d = 2$
$a < b^d$
so $T(|V|)$ is $O(|V|^2)$

(remember, we'll give you the master theorem for the actual exam if you need it!)

Runtime for Algorithm B:
$O(|E| \log(|E|)$ to sort the edges using merge or quick sort

If the graph is fully connected, algorithm B has a runtime of $|V|^2 \log(|V|^2)$, which simplifies to $O(|V|^2 \log(|V|))$. So Algorithm A is faster in this case. But if the graph has much fewer edges (suppose $|E|$ is roughly equal to $|V|$), then $O(|E| \log |E|)$ is much less than $O(|V|^2)$, so algorithm B would be better in this case.

Problem 6

Solution:

The key to solving this problem efficiently is determining what our subproblems are. The question of whether a given string can be segmented into dictionary-recognized words can be split into the problem of whether the string has a combination of *substrings* that can be segmented into dictionary-recognized words.

Our first instinct may be to approach this problem recursively. The idea is to consider each possible prefix and search for it in the dictionary. If the prefix is present in the dictionary, we recur for the rest of the string (the suffix). If the recursive call for the suffix returns true, we return true. Otherwise we try the next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.

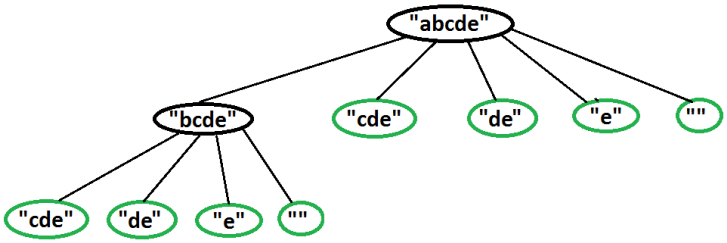
The pseudocode for the recursive solution is as follows:

Note that in our pseudocode, we are using python-like indexing for slicing strings: `string[0:i]` *indicates the substring of* `string` *up to and including the* $i - 1^{th}$ *character.*

```
# Input: string to be segmented, dict (hashset) of recognized words
# Ouput: boolean
# Purpose: returns whether or not a string can be segmented
# into a space-separated sequence of dictionary words

function word_break(string, dict):
    size = len(string)
    if size is 0:
        return true

    # For each possible prefix
    for i = 1 to i <= size:
        if dict.contains(string[0:i]) and word_break(string[i:size], dict):
            return true

    return false
```

While the recursive implementation is straightforward, it is not the most efficient solution. The issue is that we are computing the results of several subproblems more than once. Ideally, we would save ourselves from doing repeated work by storing the results of these subproblems.

For example, part of the recursion tree for the input string “abcde” is illustrated below:



The overlapping subproblems are circled in green. Since there are overlapping subproblems, we can implement a more efficient algorithm by storing the intermediate results of the subproblems.

Dynamic Programming Solution

```
# Input: string to be segmented, dict (hashset) of recognized words
# Ouput: boolean
# Purpose: returns whether or not a string can be segmented
# into a space-separated sequence of dictionary words

function word_break(string, dict):
    size = len(string)
    if size is 0:
        return true

    # Make array to store results of subproblems: wordbreak[i] stores
    # if the substring string[i:size] can be segmented or not

    wordbreak = array of length size+1
    initialize all values of wordbreak as false
    set wordbreak[size] = true

    for i = size-1 to i >= 0:
        for j = i to j < size:
            if dict.contains(string[i:j+1]) and wordbreak[j+1] = true:
                wordbreak[i] = true
                break

    return wordbreak[0]
```

Problem 7

Solution:

```
def reverseList(head):
    """ Transforms node head -> nothing
        Purpose: reverses a list of nodes
    """

    prev = null
    current = head
    next = null
    while(current != null):
       next = current.next
       current.next = prev
       prev = current
       current = next
    head = prev
```

This method iterates through the list and 'reverses' the next pointers. Try hand-simulating with a short example to see how this works and end up with a reversed list! Since the method goes over each element of the list once (from the *while* loop), the run-time is linear i.e. $O(n)$

Problem 8

Solution:

Here's pseudocode for the *sort* function:

```
def sort(S):
    """ Transforms unsorted stack of ints S -> sorted stack of ints R
        Purpose: sorts a stack of integers.
    """
    Stack R = new Stack()
    while !S.isEmpty():
        temp = S.pop()
        while(!R.isEmpty() and R.peek() > temp):
            S.push(R.pop())
        R.push(temp)
    return R
```

The idea is that each element of S is taken out and placed on the 'placeholder' stack R, only if something already on top of R isn't smaller than the elemnt we're looking at. Try to hand-simulate on a small example!

Problem 9
Solution:

1. The purpose of the learning rate η is to control how much we adjust the weights when training our perceptron. Each time our current weights incorrectly predict the label of a training example, we want to adjust our weights in the right direction. Larger values of η will cause the weights to be adjusted a lot when faced with a misclassified example. Smaller values of η cause the weights to be shifted in small amounts when faced with a misclassified example.
2. In the functional paradigm, there is no notion of state. That is, each function depends only on the input data and reliably returns the same output for a given input. Nothing else in the program can affect what is going on inside of a function.

   Another defining characteristic of the functional paradigm is that you can think of a program as a composition of functions.

   One more item is that data in the functional paradigm is immutable. That is, once a variable in a function is assigned data, it can not be modified to hold different data. Because of this, we cannot use for loops in functional programming.
3. Dijkstra's will not work on a graph with negative edge weights. The key reason is because visited nodes (nodes that have been removed from the priority queue) are assumed to already know the shortest path to themselves. Dijkstra's assumes that if a node has the current lowest cost in the priority queue, then its shortest path must already be found since adding more edges to the path will only increase the cost. However, we know that this is not the case with negative edge weights.

   To explain further, imagine a graph in which we are trying to find the shortest path from node A to B. Consider the situation where nodes A and B are separated by one edge, or hop, by a low positive weight. However, imagine the case where the shortest path involves multiple hops, and one of the hops very far away from A and B is extremely negative, which results in this path being the lowest cost path. In the first iterations, the priority queue only internalizes the costs of nodes given local paths, since it updates all nodes that are one edge away from the current node that is being visited. However, since the true shortest path is multiple nodes long, the negative weight cost will not be "uncovered" until later iterations in the algorithm.
4. P is the set of all search problems (having a well-defined solution state and checkable in polynomial time) that are also **solvable** in polynomial time. NP is a superset of P, and is simply the set of all search problems, regardless of whether they are solvable in polynomial time.
5. For offline algorithms, you have all of the data that you are going to use beforehand, so can optimize your solution based on that information. For online algorithms, you are continuously getting new data, and have to handle it in the moment without knowing what data will come next. There are many examples of offline algorithms that we used, such as any of the sorting methods. Expanding stacks and queues were an online algorithm, as was the incremental hull sort algorithm.