Chapter 4

25. We use a modified version of Kruskal's MST algorithm. We initially have n connected components, where each component C_i is simply the point p_i. Whenever we make a connection between components C_i and C_j, we create a new node v that serves as the parent node to both connected components, that is, v is the parent to the root node v_i of C_i and to the root node v_j of C_j. We associate with the node v the height h_v, with value the length of the edge joining the two components together. Because Kruskal's algorithm selects the shortest edge between two unconnected components, and these components were already connected when they were selected, the distance between these components must be greater than the length of the edge that Kruskal's algorithm selected to join each of the subcomponents together in the first place. In other words, we have both $h_v \geq h_{v_i}$ and $h_v \geq h_{v_j}$.

    Because the least common ancestor between p_i and p_j is v, and the height of v is the length of the edge that connected the two components containing p_i and p_j, Kruskal's algorithm guarantees that this height is at most the distance between p_i and p_j, since they were connected by the shortest edge that joins the two components togeth If this shortest edge is between p_i and p_j, than the height is equal to the distance. Otherwise, the algorithm found a shorter edge to connect the two components, so the height is less than the distance. In any case, we have $\tau(p_i, p_j) \leq d(p_i, p_j)$, so our τ is consistent.

    Suppose for the sake of contradiction that there exists another hierarchical metric τ' such that $\tau'(p_i, p_j) > \tau(p_i, p_j)$ for every set of points p_i and p_j. Let T' be the tree that τ' operates on, and v' be the common ancestor of p_i and p_j in T'. By the assumption, $\tau'(p_i, p_j) > \tau(p_i, p_j) \Rightarrow h_{v'} > h_v$.

    Consider the path between p_i and p_j in a minimal spanning tree. Because p_i and p_j are on opposite sides of v', there must be some node p' not in the subtree of p_i. Let p be the node in the path immediately before p', so p is in the subtree of p_i. Because the two nodes p_i and p_j are connected, the distance between p' and p is at least equal to the height of the common ancestor v'. By the assumption, this height is strictly greater than $\tau(p_i, p_j)$. But because Kruskal's algorithm merged the two connected components containing p_i and p_j after connecting those subcomponents, the length of the edge connecting the two components is greater than the distance connecting the subcomponents. But this contradicts the assumption, since this implies that each edge in the path is greater than the distance between the two connected components $\tau(p_i, p_j)$. Thus, we have that there cannot exist another hierarchical metric such that $\tau'(p_i, p_j) > \tau(p_i, p_j)$.

    Because Kruskal's algorithm runs in polynomial time and we are doing a polynomial number of operations for each of the iterations of Kruskal's algorithm, our algorithm also runs in polynomial time.

28. We start by building two spanning trees. The first spanning tree has at least k edges owned by X, and the second spanning tree has at least $n - k + 1$ edges owned by Y, or equivalently, the second spanning tree has at most k edges owned by X.

    To build the first tree, we assign a weight of 0 to each of the edges owned by X, and constant positive weight c to each of the edges owned by Y. Any minimal spanning algorithm will use as many of the 0 cost edges as possible. Similarly, we build the second tree by assining a weight of 0 to the edges owned by Y, and a constant positive weight to each of the edges owned by X. Building these trees takes $O(|E| \log |V|)$ with Kruskal's algorithm. Note that if we cannot even build these trees, then there cannot be a suitable tree, as we can't construct a tree of exactly k edges owned by X if there does not exist a tree of at least k edges owned by X.

    If our first tree has exactly k edges owned by X, then we can simply return this tree, as it has k edges owned by X and $n - k + 1$ edges owned by Y. Similarly, if the second tree has exactly $n - k + 1$ edges owned by Y, then we can simply return this tree.

    We define a procedure for transforming the second tree into the first tree. If T_1 is the same as T_2, then we're done, and there must be exactly k edges owned by X. Otherwise, consider an edge e in T_1 but not in T_2. Suppose we add this edge into T_2. Since T_2 is a tree, adding this edge creates some cycle in T_2. Note that there must be an edge e' in this cycle that is present in T_2 but not T_1. If every edge of this cycle is present in T_1, then since T_1 had the original edge e, there must be a cycle in T_1, which is a contradiction. To break the cycle in T_2, we simply remove the edge e'. Thus, at the end of this iteration, T_2 adds an edge of T_1 that it did not start with, and removes an edge of T_2 that is not present in T_1. Performing this procedure once takes polynomial time, as long as we remember which elements are different between T_1 and T_2.

    Because T_1 and T_2 are spanning trees, they have $n - 1$ edges. Therefore if we repeat the process $n - 1$ times, there must be some time where $T_1 = T_2$, since we add an edge of T_1 into T_2 with every step. Note that the number of edges owned by X in T_2 cannot decrease. If we added an X edge and removed a different X edge, then the total number of X edges does not change. However, if we added an X edge and removed a Y edge, then the total number of X edges increases by 1. But since T_1 initially contains at least k edges owned by X, T_2 intially contained at most k edges owned by X, and the total number of X edges in T_2 only increments 1 at a time, there must be some point in the transformation that T_2 becomes a spanning tree with exactly k edges owned by X. Because we have to repeat the process at most $n - 1$ times before T_2 becomes identical to T_1, after at most $n - 1$ iterations of the procedure, we will find a tree with exactly k edges owned by X. Therefore, we can find a suitable spanning tree in polynomial time.

Chapter 6

4. (a) Consider the following operating costs for 4 months, where the cost of moving $M = 100$. Clearly, because the cost of moving is so high, the optimal plan

	Month 1	Month 2	Month 3	Month 4
NY	1	2	1	2
SF	2	1	2	1

consists of staying in one location for the entire time. The cost of staying in one city the whole time is $1 + 2 + 1 + 2 = 6$. However, the cost given by the proposed algorithm is NY, move, SF, move, NY, move, SF, for a total cost of $1 + 100 + 1 + 100 + 100 + 1 = 303$.

(b) Consider the following operating costs for 4 months, where the cost of moving $M = 1$. Because the cost of moving is always less than the difference in costs

	Month 1	Month 2	Month 3	Month 4
NY	100	200	100	200
SF	200	100	200	100

between the cities, and the more expensive city switches every month, it cannot be beneficial to stay in one city for more than one month. We see that the optimal cost is $100 + 1 + 100 + 1 + 100 + 1 + 100 = 403$.

(c) Suppose we have an algorithm $OPT(i, C)$ that returns the optimal cost for i months ending in city C. Let $P(i, C)$ be the cost of operating at city C for month i. At month $i + 1$, we could have either came from the other city with a cost $M + OPT(i, C')$, or remained in the same city with a cost $OPT(i, C)$. Therefore the value of $OPT(i, C)$ can be defined recursively.

$$OPT(i,C) = \begin{cases} 0 & i = 0 \\ \min(M + OPT(i-1, C'), OPT(i-1, C)) & \textit{otherwise} \end{cases}$$

We maintain a 2-D array, with one dimension corresponding to n, and one dimension corresponding to the two cities. A sample algorithm to fill out the array:

```
# M: the moving cost
# n: the number of months to schedule
# COST: Array holding the cost of each city for each month
#       i.e. It costs COST[4][C] to operate city C in month 4
PROBLEM-4(M, n, COST):
    A[0][C] = 0
    A[0][C'] = 0
    for i from 1,...,n:
        A[i][C] = min(A[i-1][C], A[i-1][C'] + M)
        A[i][C'] = min(A[i-1][C'], A[i-1][C] + M)
    return min(A[n][C], A[n][C'])
```

6. Let $OPT(i)$ be the cost of the optimal solutions on words $w_1, ..., w_i$. Let $SLACK(i, j)$ be the slack when we put all words from w_i to w_j on the same line, where $SLACK(i, j) = \infty$ if the words don't fit on the same line.

   Consider the first word w_i that appears on the last line L_k. Then the cost of this particular word break is $OPT(i - 1) + SLACK(i, n)^2$. By iterating over all possible values of i, we can arrive at the optimal solution.

```
PROBLEM-8:
    A[0] = 0
    for i from 1,...,n:
        for j from 1,...,i:
            A[i] = min(A[i], SLACK(j, i)^2 + A[j-1])
    return A[n]
```

12. It suffices to find an array of optimum costs, since the configuration of minimum total cost can be extracted from this array. Suppose that $OPT(i)$ is the cost of the best configuration on the first i servers such that the file exists on server i. Because of this stipulation that the file must be on server i, $OPT(i)$ may not be the best solution possible. Suppose that the file is placed in another location $k < i$ as well. By definition, the costs of servers 1 through k is $OPT(k)$, since k has the file. The file does not exist in each of the servers between k and i, so each of these servers must pay an access cost. Server $i-1$ pays an access cost of 1, server $i-2$ pays an access cost of 2, etc. The total access cost is then $1 + 2 + ... + (i-k-2) + (i-k-1) = \frac{(i-k-1)(i-k)}{2}$. Finally, since the cost of placing the file at server k is c_k, the total cost for placing another file at server k is $OPT(k) + \frac{(i-k-1)(i-k)}{2} + c_k$. We can then iterate over all possible values of k, until we find the value that minimizes $OPT(i)$.

```
PROBLEM-12:
    A[0] = 0
    for i from 1,...,n:
        for k from 1,...,i:
            A[i] = c_i + min(A[i], OPT(k) + (i-k-1)*(i-k)/2)
    return A[n]
```