Q1

a)

Order the clowns in the order of increasing heights and breaks ties arbitrarily with heapsort (content of CSC263 on CLRS page 151). This order is the optimal order of clowns that will minimize the strong-man's effort.

b)

Step 1: we define an order O' has an inversion if clown i with height $h_i$ is ordered before clown j with height $h_j$ but clown i is taller than clown j, $h_i > h_j$.

Step 2: there is an optimal order that has no inversion. Proof:

If order O' has an inversion, then there is a pair of clowns i and j such that clown j is ordered immediately after clown i and $h_i > h_j$. This is because, suppose O' has an inversion such that clown m is ordered somewhere before clown n and has $h_m > h_n$. Then advance the ordered clowns from m to n once at a time, there would be a point where the height of the clown starts to decrease. That's where a consecutive pair of clowns forms an inversion.

After switching the order of consecutive clowns i and j we get one less inversion.

The new order O requires total effort no larger than that of O'. Proof: since we only switched clown i (suppose O'[a] is clown i and thus O'[a + 1] is clown j) and clown j from order O' to get order O, but did not add or remove any clowns, then if clown m $\in O'[1...k]$ for $k \in [1, a-1] \bigcup [a+2, n]$, then clown m $\in O[1...k]$ for $k \in [1, a-1] \bigcup [a+2, n]$, and vise versa. Therefore, $H_k^{O'} = (\sum_{m=1}^{k} h_{O'[m]}) - (k-1) * f = (\sum_{m=1}^{k} h_{O[m]}) - (k-1) * f = H_k^O$ for $k \in [1, a-1] \bigcup [a+2, n]$.

$\sum_{k=1}^{n} H_k^{O'} - \sum_{k=1}^{n} H_k^O$

$= \sum_{k=1}^{a-1} H_k^{O'} - \sum_{k=1}^{a-1} H_k^O + H_a^{O'} + H_{a+1}^{O'} - H_a^O - H_{a+1}^O + \sum_{k=a+2}^{n} H_k^{O'} - \sum_{k=a+2}^{n} H_k^O$

$= H_a^{O'} + H_{a+1}^{O'} - H_a^O - H_{a+1}^O$ (as proved above, $H_k^{O'} = H_k^O$ for $k \in [1, a-1] \bigcup [a+2, n]$.)

$= \sum_{m=1}^{a} h_{O'[m]} - (a-1) * f + \sum_{m=1}^{a+1} h_{O'[m]} - (a+1-1) * f - (\sum_{m=1}^{a} h_{O[m]} - (a-1) * f +$
$\sum_{m=1}^{a+1} h_{O[m]} - (a+1-1) * f)$

$= \sum_{m=1}^{a} h_{O'[m]} + \sum_{m=1}^{a+1} h_{O'[m]} - \sum_{m=1}^{a} h_{O[m]} - \sum_{m=1}^{a+1} h_{O[m]}$

$= 2H_{a-1}^{O'} - 2H_{a-1}^O + 2h_{O'[a]} + h_{O'[a+1]} - 2h_{O[a]} - h_{O[a+1]}$

$= 2h_i + h_j - 2h_j - h_i$

Since $h_i > h_j$, then assume $h_i = h_j + p$, $p$ is a positive integer.

$= 2(h_j + p) + h_j - 2h_j - (h_j + p)$

$= p > 0$

Since The effort made by the strong-man is proportional to the cumulative height and $\sum_{k=1}^{n} H_k^{O'} > \sum_{k=1}^{n} H_k^O$, then the new order O requires total effort no larger than that of O'.

Step 3: All orders with no inversion require the same total effort. Proof:
If two different orders both have no inversions, they may not be exactly the same, but they can only differ in the order in which clowns with the same height are ordered. In both orders, because they do not have any inversions, the clowns with same heights are ordered consecutively. The different orders of some consecutive numbers that have the same value do not effect the cumulative sum. Therefore, any two orders with no inversions produce the same cumulative height.

Step 4: Our greedy algorithm from (a) orders the clowns in the order of increasing heights and breaks ties arbitrarily, therefore it does not have any inversions. Since there is an optimal order that has no inversions (step 2) and all orders with no inversion require the same total effort (step 3), then the order produced by the greedy algorithm from (a) require the same total effort as some optimal solutions. Therefore the algorithm will always yield a minimum effort solution.

c)
Since the algorithm only used heapsort to order the clowns, the worst case time complexity of this algorithm is the same as that of heapsort, which is $O(n \log n)$ (CLRS page 153).

d)
```
tallButLowEffort(L[1, ..., n], H)
1   L'[1, ..., n] = heapSort(L[1, ..., n]) // Order the clowns in the order of increasing
                                               heights
2   optimalSequence = an empty linked list
3   H' = H - f // Max height the clowns can fit in minus the size of the head of the clown
                  on the top
4   for (int i = n; i > 0; i -= 1)
5       if (H' - h_{L'[i]} > 0)
6           H' = H' - h_{L'[i]}
7           optimalSequence.add(L'[i])
8   reverse optimalSequence // So that the shorter clowns go first
9   return optimalSequence
```

e)
Part 1: the final solution makes the tallest tower. Proof:
By contradiction, suppose the sequence L'[1, ..., m] (m is some integer such that $0 \leq m \leq n$) produced by tallButLowEffort(L[1, ..., n], H) is not the optimal solution and that there

is an optimal solution O[1, ..., k] (k is some integer such that $0 \leq k \leq n$) such that O[1, ..., k] makes a taller tower within the constraint.
Would like to show that L'[1, ..., m] $\subseteq$ O[1, ..., k] by induction
Statement $S(i)$: for $i \in \mathbb{N}$ and $0 \leq i \leq m$, L'[m - i] $\in$ O[1, ..., k].

Base case: would like to show that S(i) holds when i = 0
Suppose, by contradiction, that L'[m - 0] $\notin$ O[1, ..., k]. Since, by algorithm from d), L'[m] is the tallest clown that has shoulder height $h_{L'[m]} \leq H' = H - f$, then any clowns taller than L'[m] are not in O[1, ..., k]. Then O[1, ..., k] = {j $|h_{L[j]} < h_{L'[m]}$ }. Since the shoulder heights $h_i - f$ of the clowns are all unique powers of 2, then $(\sum_{j \text{ such that } h_{L[j]} < h_{L'[m]}} h_{L[j]}) < h_{L'[m]}$, which contradicts the statement that O[1, ..., k] is the optimal solution. Therefore L'[m] $\in$ O[1, ..., k]

Inductive step:
Given some $i > 0$, assume S(p) holds for all p with $0 \leq p < i \leq m$. Would like to show that S(i) holds.
Suppose, by contradiction, that L'[m - i] $\notin$ O[1, ..., k]. Since, by assumption, S(p) holds for all p with $0 \leq p < i \leq m$, and by the algorithm from d), all clowns that are taller than L'[m - i] and could fit in the height restriction are already in O[1, ..., k]. Let T = { L'[m - 0], L'[m - 1], ..., L'[m - i + 1] } denote this set. Then O[1, ..., k] - T = {j $|h_{L[j]} < h_{L'[m-i]}$ }. Since the shoulder heights $h_i - f$ of the clowns are all unique powers of 2, then $(\sum_{j \text{ such that } h_{L[j]} < h_{L'[m-i]}} h_{L[j]}) < h_{L'[m-i]}$, which means $\sum_{j=1}^{k} h_{O[j]} < \sum_{j=1}^{m} h_{L'[j]}$, which contradicts the statement that O[1, ..., k] is the optimal solution. Therefore L'[m - i] $\in$ O[1, ..., k], and S(i) holds. Therefore, L'[1, ..., m] $\subseteq$ O[1, ..., k].

Since, by assumption, O[1, ..., k] makes a taller tower than L'[1, ..., m] within the constraint, and L'[1, ..., m] $\subseteq$ O[1, ..., k], then there exist a number j such that j $\in$ O[1, ..., k] and j $\notin$ L'[1, ..., m]. But this is not possible as the algorithm from d) iterate through all the clowns ordered from max height to min height and puts any clown that would fit the height constraint into the list.

Part 2: Among all such sequences, our solution minimizes the strong-man's effort.
As proved in part (b), having the shortest clowns go first would minimize the strong man's effort.

f)
The worst case time complexity of the heapsort on line1 is $O(n \log n)$. The time complexity of the for loop on line 4 takes O(n), and reversing the linked list on line 7, as mentioned in CSC148, takes O(n). Therefore the worst case time complexity of this algorithm is $O(n \log n)$

g)
Our greedy algorithm of (a) does not guarantee to provide a solution that minimizes the strong-man's effort. Counter example: suppose k = 2 there are only two clowns. Clown 1 has height 1 and girth 10, while clown 2 has height 2 and girth 1. Our greedy algorithm of (a) would have clown 1 go first , resulting in cumulative area $10 * 1 + 10 * (1 + 2) = 40$. However, if clown 2 goes first the cumulative area is only $1 * 2 + 10 * (2 + 1) = 32$.

Q2
a)
Define helper functions:

upperTangent (leftSequence, rightSequence, $p_i$, $p_j$):
    // while the line is not yet the upper tangent line
    while $(p_j - p_i) \times (p_{j+1} - p_i) > 0$ or $(p_i - p_j) \times (p_{i-1} - p_j) > 0$
        // While part of the right arena still above the line, move the point up
        while $(p_j - p_i) \times (p_{j+1} - p_i) > 0$:
            j $\leftarrow$ j + 1, L $\leftarrow$ L' line joining $p_i$ and new $p_j$
        // While part of the left arena still above the line, move the point up
        while $(p_i - p_j) \times (p_{i-1} - p_j) > 0$:
            i $\leftarrow$ i - 1, L $\leftarrow$ L' line joining new $p_i$ and $p_j$
    return $p_i, p_j$

lowerTangent (leftSequence, rightSequence, $p_i$, $p_j$)
    // while the line is not yet the lower tangent line
    while $(p_j - p_i) \times (p_{j-1} - p_i) < 0$ or $(p_i - p_j) \times (p_{i+1} - p_j) < 0$
        // While part of the right arena still below the line, move the point down
        while $(p_j - p_i) \times (p_{j-1} - p_i) < 0$
            j $\leftarrow$ j - 1, L $\leftarrow$ L' line joining $p_i$ and new $p_j$
        // While part of the left arena still below the line, move the point down
        while $(p_i - p_j) \times (p_{i+1} - p_j) < 0$
            i $\leftarrow$ i + 1, L $\leftarrow$ L' line joining new $p_i$ and $p_j$
    return $p_i, p_j$

Let P $= [p_1 = (x_1, y_1), p_2 = (x_2, y_2), ..., p_n = (x_n, y_n)]$ be the sequence of points where the n stakes lays.
circularSequence(P$= [p_1, ..., p_n]$)
    if n $= 2$

return P = $[p_1, p_2]$

else if n = 3

if $(p_2 - p_1) \times (p_3 - p_1) > 0$

return circular sequence $[p_1, p_3, p_2]$

else

return circular sequence $[p_1, p_2, p_3]$

else

P = heapSort(P) based on the x values

m = $\lfloor \frac{length(P)}{2} \rfloor$

leftSequence = circularSequence($[p_1, p_2, ..., p_m]$)

rightSequence = circularSequence($[p_{m+1}, p_{m+2}, ..., p_n]$)

iterate through leftSequence and pick a point with the largest x value $p_i$

iterate through rightSequence and pick a point with the smallest x value $p_j$

upTanLeftPoint, upTanRightPoint = upperTangent(leftSequence, rightSequence, $p_i$, $p_j$)

lowTanLeftPoint, lowTanRightPoint = lowerTangent(leftSequence, rightSequence, $p_i$, $p_j$)

combinedSequence = [ ]

iterate through leftSequence, append lowTanLeftPoint, points between lowTanLeftPoint and upTantLeftPoint, upTantLeftPoint to combinedSequence.

iterate through rightSequence, append upTantRightPoint, points between upTanRightPoint and lowTantRightPoint, lowTantRightPoint to combinedSequence.

return combinedSequence

b)

Correctness: Proof by induction.

Statement S(n): the algorithm determines the clockwise circular sequence of n stakes that anchor the rope, n is some positive integer.

Base case: n = 1: the problem is not defined when n = 1 therefore trivially true.

n = 2: when there are 2 stakes, either order of the two points can be clockwise

n = 3, pick a random point as the vertex and calculate sine(the angle formed by the vertex and two other points) by doing cross product would give us the orientation of the angle and thus determine which order is the clockwise order.

Inductive Step: given some n > 3 would like to show that P(n) holds under the assumption (inductive hypothesis) that P(m) holds for all m with $1 \leq m < n$

Since the algorithm circularSequence recursively calls circularSequence on two sub circular sequences of size $\frac{n}{2}$. By the inductive hypothesis, these calls will correctly return circular sequences of stakes that anchor sub-arenas. When combining the two sub-arenas, circularSequence calls upperTangent and lowerTangent to determine an upper tangent line and a lower tangent line. The two tangent lines are where the rope

c)
Base case takes constant time. Since each time we recursively calls circularSequence on two sub circular sequences of the same size, and have to iterate through the each sub arrays 4 times to combine them together, therefore $T(n) = 2T(\frac{n}{2}) + 4n$. Based ons the Master Theorem, the time complexity of the algorithm is $O(n \log n)$

d)
In the worst case we have nested triangles. Base on part (c), for each layer of triangle the time complexity is $O(n \log n)$; we have $\frac{n}{3}$ triangles, therefore the time complexity of the algorithm is $O(n^2 \log n)$

e)
Define helper function:

// This function determines if points $p_1, p_2$ are on the same side of the line defined by
// points $l_1, l_2$
sameSideOfLine($l_1, l_2, p_1, p_2$):
    if $((l_2 - l_1) \times (p_1 - l_1)) * ((l_2 - l_1) \times (p_2 - l_1)) < 0$
        return false
    else
        return true

onTheLine($l_1, l_2, p$):
    if $((l_2 - l_1) \times (p - l_1)) = 0$ // $p$ is on the line defined by $l_1, l_2$
        dotProduct_p_l $= (l_2 - l_1) \cdot (p - l_1)$
        dotProduct_l_l $= (l_2 - l_1) \cdot (l_2 - l_1)$
        if dotProduct_p_l $< 0$ or dotProduct_p_l $>$ dotProduct_l_l: // $p$ is not on the line
                                                                    // segment defined by $l_1, l_2$
            return false
        else
            return true
    else

return false

Suppose C is the circular sequence output from (a), and p = (u, v) is the landing point.
checkInsideArena(C, p)
if k = 3
if sameSideOfLine($c_1, c_2, c_3, p$) and sameSideOfLine($c_3, c_2, c_1, p$) and sameSideOfLine($c_1, c_3, c_2, p$):
// if the point is on the same side of each vertex with respect to the line defined
// by the other two vertices, then the point is inside of the triangle defined by
// the three points
return true
else if onTheLine($c_1, c_2, p$) or onTheLine($c_3, c_2, p$) or onTheLine($c_1, c_3, p$):
// Count the point being on the rope as inside the arena
return true
else
return false
else
pick any point $c_i$ and its opposite point $c_{i+\frac{k}{2}}$ in the sequence.
if onTheLine($c_i, c_{i+\frac{k}{2}}, p$):
return true
if sameSideOfLine($c_i, c_{i+\frac{k}{2}}, c_{i+\frac{k}{4}}, p$):
let C' be circular sequence $[c_i, c_{i+1}, ..., c_{i+\frac{k}{2}}]$
return checkInsideArena(C', p)
else
let C' be circular sequence C - $[c_i, c_{i+1}, ..., c_{i+\frac{k}{2}}]$
return checkInsideArena(C', p)

Q3
a)
Initialize a fully connected weighted undirected graph G=(V, E) such that each vertex is either a lion or a tamer. And each edge connecting vertex i and j represents a plank connecting lion/tamer i with lion/tamer j, and the weight on that edge represents the length $l(i, j)$. Use an adjacency-matrix M to represent G.
Algorithm:

Step 1: Remove planks directly connecting two lions.
for $1 \leq i \leq m$
for $1 \leq j \leq m$

if i $\neq$ j
    set M[i, j] = 0

Step 2: For each lion, search through all planks connecting to tamers. Only keep the plank with the smallest length.
for $1 \leq j \leq m$
    keep track of what the current min dist is and is between which tamer
    for $m + 1 \leq j \leq n$
        if $l(i, j) <$ min_dist, $l(i, j')$:
            set M[i, j'] = 0 and update min_dist = $l(i, j)$ and tamer = $j$
        else:
            set M[i, j] = 0

Step 3:
Define sub-graph G'=(V', E') such that each vertex is a tamer. And each edge connecting vertex i and j represents a plank connecting tamer i with tamer j. Apply Kruskal's algorithm (content of CSC263 on CLRS page 624) to find the minimum spanning tree and update matrix M accordingly.
And the edge set { (i, j)|M[i, j] = 1} is the the subset of planks that meets the three conditions and minimizes the total length.

b)
Proof for correctness:
Step 1 removed all planks between any two lions. Therefore, with the remaining planks, no lions are directly connected to each other by a plank, which satisfies the first condition.

Since lions need to be connected directly to exactly one tamer and lions do not connect to other lions, it is not possible to create a path between two tamers that has lions on the path. So by making a minimum spinning tree on the subgraph that only had the tamers guarantees that there is a path of planks connecting any two tamers and the total length of the path is minimized. Condition 3 is satisfied.

Finding each lion a tamer that is closest to the lion guarantees that every lion is directly connected by a plank to exactly one tamer, which satisfies the second condition, and the total length of the planks combined is minimized.

Therefore, the algorithm described in (a) satisfies the three conditions and selects the subset of planks that minimizes the total length.

Worst case time complexity:
Step 1 has a for loop nested inside of another for loop and the cost is $m(m-1) \in O(m^2)$
Step 2 has a for loop nested inside of another for loop and the cost is $m(n-m) \in O(n^2)$ (since there are more tamers than lions $n > m$).
Step 3 had Kruskal's algorithm run on G'=(V', E'). According to what we learned in CSC263 and on CLRS page 633, the running time of Kruskal's algorithm is $O(E' \log V')$. Since each vertex represents a tamer, |V'| = n - m, and since it is a complete graph, $|E'| = \frac{(n-m)(n-m-1)}{2}$. Since there are more tamers than lions $n > m$, then the cost is $O(n^2 \log n)$. Therefore the worst case complexity is $O(n^2 \log n)$.

Q4
a)

```
def if_safe(L[a_1, a_2, ..., a_p], F)
    if p = 2: // The base case is when there are only 2 people since according to
                  the definition, a segment of rope is the rope between any 2 acrobats
        if f_1 * q_1 + f_2 * q_2 > F // Define q_k = 1 if i_k is hanger, -1 otherwise for k ∈ [1, 2]
            return false
        else
            return true
    else
        m = ⌊p/2⌋
        left = if_safe(L[a_1, ..., a_m])
        if !left: // If the left half is not safe then terminates the program and reports it
            return false
        right = if_safe(L[a_m, ..., a_p])
        if !right: // If the right half is not safe then terminates the program and reports it
            return false
        left_max = 0
        iterate from a_m to a_1 by index i
            if ∑_{z=i}^{m} f_z * q_z > left_max: // Define q_z = 1 if i_z is hanger, -1 otherwise
                left_max = ∑_{z=i}^{m} f_z * q_z
        right_max = 0
        iterate from a_m to a_p by index i
            if ∑_{z=m}^{i} f_z * q_z > right_max:
                right_max = ∑_{z=i}^{m} f_z * q_z
        if left_max + right_max > F: // Some middle segment is not safe
            return false
```

```
    else
        return true
```

b)
Correctness: Proof by induction.
Define P(k): the algorithm is correct when there are k acrobats, k is some positive integer.

Base case: k = 1, 1 acrobat does not form a segment, therefore trivially true.
k = 2, when there are 2 acrobats, we calculate total force applied by these two people and compare it with F, therefore correct.

Inductive Step: given some k > 2, would like to show that P(k) holds under the assumption (induction hypothesis) that P(m) holds for all m with $2 \leq m < k$.
Since the algorithm if_safe recursively calls if_safe on two arrays (left subarray and right subarray) of size $\frac{k}{2}$. By the induction hypothesis, these calls will correctly determine if the left subarray and the right subarray are safe. If they are not, the program reports it and terminates. If they are safe, the program looks for an array in the middle that has elements from both left and right subarrays and that has the largest net force. If the rope segment with this array of acrobats is safe then the whole rope is safe. Therefore the algorithm is correct.

Time complexity:
Since each time we recursively calls if_safe on two subarrays of the same size, and iterate through the array itself to check if some array in the middle that has elements from both left and right subarrays is safe, then $T(n) = 2T(\frac{n}{2}) + n$. Based on the Master Theorem, the time complexity of the algorithm is $O(n \log n)$

c)
Greedy algorithm:

Initialize an undirected graph G = (V, E) such that each vertex represents an acrobat. If one acrobat knows another acrobat, then there is an edge between them. The maximum vertex degree in the graph is the maximum number of acrobats that any acrobat knows.

Helper function:

```
// This function looks for the lowest number color that has not yet been used by a
// specific vertex's adjacent neighbors.
firstColorAvaliable(L):
    initialize an empty array colorIndex of length m + n + 1
```

```
    for color in L:
        colorIndex[color] = 1
    for color in colorIndex:
        if colorIndex[color] == 0
        return color

color(adjacency-list representation of G = (V, E)):
    colorMapping = map()
    for v in V:
        adjacentColors = [ ] // collects the colors that are being used by vertex v's adjacent
                                neighbors.
        for v' in adjacency-list of v:
            if v' in colorMapping:
                adjacentColors.append(colorMapping(v'))
        colorMapping[v] = firstColorAvaliable(adjacentColors) // assign the lowest number
                                                                 color that has not yet been used by v's
                                                                 adjacent neighbors.
```