CMSC 27230: Problem Set 3

Owen Fahey

January 31, 2023

Problem 1

a) $T(n) = 8T(\frac{n}{4}) + 5n$

$T(n) = 64T(\frac{n}{16}) + 15n$

$T(n) = 512T(\frac{n}{64}) + 35n$

$T(n) = 4096T(\frac{n}{256}) + 75n$

$T(n) = 8^k T(\frac{n}{4^k}) + 5n\left(2^k - 1\right)$

Let $k = log_4(n)$ so that $\frac{n}{4^k} = 1$

$T(n) = 8^{log_4(n)} \cdot T(1) + 5n(2^{log_4(n)} - 1)$

If $T(1) = 1$:

$T(n) = 8^{log_4(n)} + 5n(2^{log_4(n)} - 1)$

$T(n) = n^{3/2} + 5n(\sqrt{n} - 1)$

$T(n) = 6n^{3/2} - 5n$

b) $T(n) = 4T(\frac{n}{2}) + 3n^2$

$T(n) = 16T(\frac{n}{4}) + 6n^2$

$T(n) = 64T(\frac{n}{8}) + 9n^2$

$T(n) = 4^k T(\frac{n}{2^k}) + 3kn^2$

Let $k = log_2(n)$ so that $\frac{n}{2^k} = 1$

$T(n) = 4^{log_2(n)} \cdot T(1) + 3log_2(n) \cdot n^2$

If $T(1) = 1$:

$T(n) = n^2 + 3n^2 log_2(n)$

c) $T(n) = 2T(\frac{n}{8}) + 6n - 1$

$T(n) = 4T(\frac{n}{64}) + \frac{15}{2}n - 3$

$T(n) = 8T(\frac{n}{512}) + \frac{63}{8}n - 7$

$T(n) = 2^k T(\frac{n}{8^k}) + (8 - 8(\frac{1}{4})^k)n - (2^k - 1)$ *

Let $k = log_8(n)$ so that $\frac{n}{8^k} = 1$

$T(n) = 2^{log_8(n)} \cdot T(1) + (8 - 8(\frac{1}{4})^{log_8(n)})n - 2^{log_8(n)} + 1$

If $T(1) = 1$:

$T(n) = n^{1/3} + (8 - 8n^{-\frac{2}{3}})n - n^{1/3} + 1$

$T(n) = -8n^{\frac{1}{3}} + 8n + 1$

$T(n) = -8n^{\frac{1}{3}} + 8n + 1$

* I used the geometric series formula to conclude $\sum_{i=0}^{k-1} 2^i 6 \cdot \frac{1}{8^i} = (8 - 8(\frac{1}{4})^k)$

d) $T(n) = 6T(\frac{n}{2}) - 8T(\frac{n}{4}) + n + 9log_2(n)$

We can attack this problem by using substitution.

$T(n) - 2T(\frac{n}{2}) = 4T(\frac{n}{2}) - 8T(\frac{n}{4}) + n + 9log_2(n)$

Let $S(n) = T(n) - 2T(\frac{n}{2})$

$S(n) = 4S(\frac{n}{2}) + n + 9log_2(n)$

$S(n) = 16S(\frac{n}{4}) + 3n + 45log_2(n) - 36$

$S(n) = 64S(\frac{n}{8}) + 7n + 189log_2(n) - 324$

$S(n) = 4^k S(\frac{n}{2^k}) + (2^k - 1)n + \sum_{i=0}^{k-1} 9 * 4^i log_2(\frac{n}{2^i})$

$S(n) = 4^k S(\frac{n}{2^k}) + (2^k - 1)n + 3\log_2(n)\left(4^k - 1\right) - 12(k-1)4^{k-1} + 4\left(4^{k-1} - 1\right)$

$S(n) = 4^k S(\frac{n}{2^k}) + (2^k - 1)n + 3\log_2(n)\left(4^k - 1\right) + (4 - 3k)4^k - 4$

Let $k = log_2(n)$ so that $\frac{n}{2^k} = 1$

$S(n) = 4^{log_2(n)}S(1) + (2^{log_2(n)} - 1)n + 3\log_2(n)\left(4^{log_2(n)} - 1\right) + (4 - 3log_2(n))4^{log_2(n)} - 4$

Let $S(1) = c_1$:

$S(n) = c_1 n^2 + (n-1)n + 3\log_2(n)(n^2 - 1) + (4 - 3log_2(n))n^2 - 4$

$S(n) = (c_1 + 1)n^2 - n + 3n^2\log_2(n) - 3log_2(n) + 4n^2 - 3n^2 log_2(n) - 4$

$S(n) = (c_1 + 5)n^2 - n - 3log_2(n) - 4$

We will figure out what c_1 is later so let's just replace $c_1 + 5$ with c

$S(n) = cn^2 - n - 3log_2(n) - 4$

Now we plug this back into the original formula for $S(n)$:

Let $cn^2 - n - 3log_2(n) - 4 = T(n) - 2T(\frac{n}{2})$

$T(n) = 2T(\frac{n}{2}) + cn^2 - n - 3log_2(n) - 4$

We know how to solve this!

$T(n) = 2T(\frac{n}{2}) + cn^2 - n - 3log_2(n) - 4$

$T(n) = 4T(\frac{n}{4}) + \frac{3c}{2}n^2 - 2n - 9log_2(n) - 6$

$T(n) = 8T(\frac{n}{8}) + \frac{7c}{4}n^2 - 3n - 21log_2(\frac{n}{2}) + 2$

$T(n) = 2^kT(\frac{n}{2^k}) + \sum_{i=0}^{k-1} 2^i c(\frac{n}{2^i})^2 - kn - \sum_{i=0}^{k-1} 2^i log_2(\frac{n}{2^i}) + \sum_{i=0}^{k-1} 2^i \cdot -4$

$T(n) = 2^kT(\frac{n}{2^k}) + 2^{1-k}(2^k - 1)cn^2 - kn - 3(2^k - 1)log_2(n) + 3(2^k k - 2(2^k - 1)) - 4(2^k - 1)$

$T(n) = 2^kT(\frac{n}{2^k}) + 2^{1-k}(2^k - 1)cn^2 - kn - 3(2^k - 1)log_2(n) + 2^k(3k - 10) + 10$

Almost there! Let $k = log_2(n)$ so that $\frac{n}{2^k} = 1$

$T(n) = 2^{log_2(n)}T(1) + 2^{1-log_2(n)}(2^{log_2(n)} - 1)cn^2 - nlog_2(n) - 3(2^{log_2(n)} - 1)log_2(n) + 2^{log_2(n)}(3log_2(n) - 10) + 10$

Let $T(1) = 1$:

$T(n) = n + 2n^{-1}(n - 1)cn^2 - nlog_2(n) - 3(n - 1)log_2(n) + n(3log_2(n) - 10) + 10$

$T(n) = 2cn^2 - (2c + 9)n + (3 - n)log_2(n) + 10$

But wait! We also have to do $T(2) = 2$. Let $k = log_2(n) - 1$ so that $\frac{n}{2^k} = 2$

$T(n) = 2^{log_2(n)}T(1) + 2^{1-log_2(n)}(2^{log_2(n)} - 1)cn^2 - nlog_2(n) - 3(2^{log_2(n)} - 1)log_2(n) + 2^{log_2(n)}(3log_2(n) - 10) + 10$

$T(n) = 2^{log_2(n)-1}T(2) + 2^{1-(log_2(n)-1)}(2^{log_2(n)-1} - 1)cn^2 - (log_2(n) - 1)n - 3(2^{log_2(n)-1} - 1)log_2(n) + 2^{log_2(n)-1}(3(log_2(n) - 1) - 10) + 10$

$T(n) = \frac{n}{2} \cdot 2 + \frac{4}{n}(\frac{n}{2} - 1)cn^2 - (log_2(n) - 1)n - 3(\frac{n}{2} - 1)log_2(n) + \frac{n}{2}(3(log_2(n) - 1) - 10) + 10$

$T(n) = 2cn^2 - (4c + \frac{9}{2})n + (3 - n)log_2(n) + 10$

Now we need to find c

$4c + \frac{9}{2} = 2c + 9$

$c = \frac{9}{4}$

Final Answer:

$T(n) = \frac{9}{2}n^2 - \frac{27}{2}n + (3 - n)\log_2 n + 10$

Problem 2.a

Solution: The statement is true.

Proof of Solution

Claim: A tree will cease to be a tree if a new node is connected to two or more existing nodes

Proof: Consider the following properties of trees: 1) they cannot contain cycles and 2) there is a path between any two vertices. Assume it possible to connect a new node to two existing nodes without creating a cycle. Let's say V_0 is being adding to a tree and V_1, V_2 are nodes already in the tree. We know that some path, P, already exists between V_1 and V_2. If we connect V_0 to both V_1 and V_2 then we can travel from V_0 to V_1 then along P and finally from V_2 to V_0. This clearly constitutes a cycle. Therefore, our assumption is wrong.

Claim: All minimum spanning trees of a graph will have $|V - 1|$ edges

Proof: This can be proved fairly easily with induction.

> Assume $P(n) : |E| = n - 1$ for a tree with $|V| = n$ where $n \in \mathbb{N}$.
>
> $P(1) = 0$ since an edge requires at least 2 nodes.
>
> $P(n + 1) = n$. Exactly one edge connects a new $(n + 1)^{th}$ node. Connecting a new node by any more than one edge would mean the tree would cease to be a tree. Hence, $P(n+1) = P(n) + 1 = n - 1 + 1 = n$.

Claim: If we modify the edge lengths of G by changing the length of each edge $e \in E(G)$ from l_e to $2l_e + 3$ then in the resulting graph, T will still be a minimum spanning tree.

Proof: Let us call the graph before the update G_1 and the graph after the update G_2. Let $T1$ be the some minimum spanning tree of G_1 and $T2$ be the same tree after the update. We know all minimum spanning trees of a graph will have the same value. Hence, we know that the sum of all the edges of all minimum spanning trees of G_1 is equal to:

$$\sum_{e \in E(T_1)} l_e$$

Furthermore, the sum of all the edges of all minimum spanning trees of G_1 will have the following value after the update (in terms of T_1):

$$\sum_{e \in E(T_1)} (2l_e + 3)$$

Assume there exists some T_2' that is a minimum tree spanning tree of G_2 but not a minimum spanning tree of G_1. In order for this to be true, the following criteria would have to hold:

$$1) \sum_{e \in E(T_2')} l_e < \sum_{e \in E(T_1)} (2l_e + 3)$$

$$2) \sum_{e \in E(T_2')} \frac{l_e - 3}{2} > \sum_{e \in E(T_1)} l_e$$

We can rewrite these equations to arrive at a contradiction. Notice we are relying on the fact that T_1 and T_2' have the same number of edges since they have same number of nodes.

Rewriting equation 1:

$$\sum_{e \in E(T_2')} l_e < \sum_{e \in E(T_1)} (2l_e + 3)$$

$$\sum_{e \in E(T_2')} l_e < 2 \sum_{e \in E(T_1)} l_e + \sum_{1}^{|V-1|} 3$$

Rewriting equation 2:

$$\sum_{e \in E(T_2')} \frac{l_e - 3}{2} > \sum_{e \in E(T_1)} l_e$$

$$\sum_{e \in E(T_2')} l_e - 3 > 2 \sum_{e \in E(T_1)} l_e$$

$$\sum_{e \in E(T_2')} l_e > 2 \sum_{e \in E(T_1)} l_e + \sum_{1}^{|V-1|} 3$$

We have reached a contradiction. Thus our assumption that there exists some T_2' that is a minimum tree spanning tree of G_2 but not a minimum spanning tree of G_1 is false. We know from Theorem 5.6 in the notes that a minimum spanning tree exists for any graph G. Thus, the only possibility is that T_2 is a minimum spanning tree of G_2.[1] $\square$

Problem 2.b

Solution: The statement is false. P may no longer be the shortest graph.

Consider the following figures. In Figure 1, the shortest path from s to t is $\{(s, a)(a, t)\}$. However, in Figure 2, the shortest path from s to t is $\{(s, t)\}$.

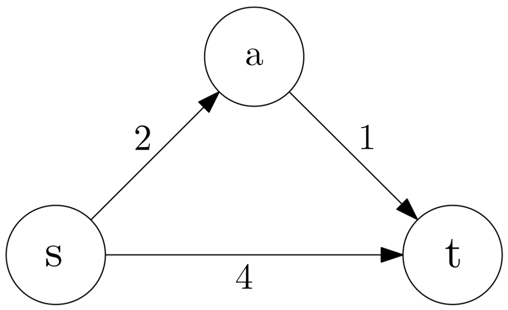


Figure 1: Initial G

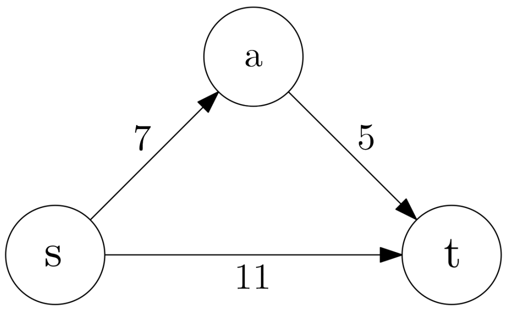


Figure 2: Updated G

[1]Another way to do this proof is to show that the orderings of the edges by length do not change so Kruskal's algorithm produces the same result.

Problem 3

Solution: We can use the following recursive algorithm:

The algorithm takes a set of samples, P, that are potentially diseased and test them. If all samples are clean, the process terminates. Otherwise match against the following cases:

a) If $|P| = 1$, note if p_1 is diseased then terminate

b) Else, let $k = \lfloor \frac{|P|}{2} \rfloor$ call the algorithm on $\{p_1, ..., p_k\}$ and $\{p_k, ..., p_{|P|}\}$

Pass S into the algorithm

Explanation of Correctness:

The only way to know if a sample is diseased is to test it on its own. This algorithm is correct because it ends up testing all diseased samples alone. This can be seen in that fact that it only terminates when it tests a sample alone or when every sample in some subset is clean.

Run time explanation:

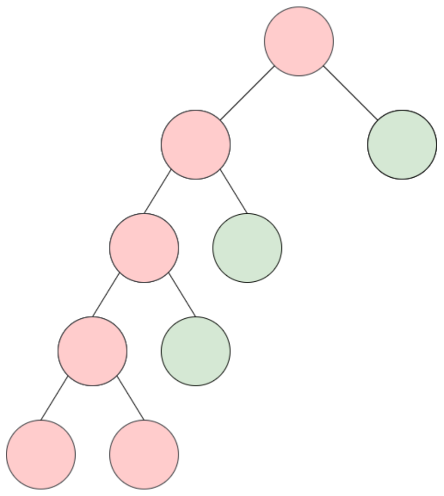


Figure 3: Best Case for $k = 2$, $n = 16$

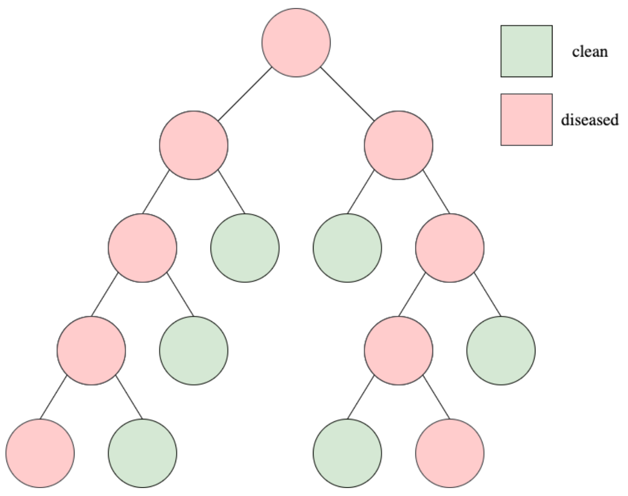


Figure 4: Worst Case for $k = 2$, $n = 16$

The run time of this algorithm is $O(klogn + 1)$.

We know the height of the tree will always be equal to $log_2(n)$ since we are repeatedly dividing the size of the subsets we are testing by (at least) half.

There will always be k paths from the root to a diseased leaf. Each diseased node on this path has at most 1 sibling node that is not diseased. Hence, there will be at most $2klogn$ nodes in the tree (this is quite a generous upper bound). Finally, we know that if $k = 0$, we still have to perform 1 initial test.

Hence, we can say that upper bound on the number of tests is $2klogn + 1$ steps. Therefore, this algorithm has a run time of $O(klogn + 1)$.

Problem 4

Solution: We can modify the binary search algorithm to solve this problem:

Use a recursive algorithm with inputs A, B, i and j that always return a missing element a_k. If a_x is in A and not in B, a_x must be between the i^{th} element and j^{th} element of A

Take $k = \lfloor \frac{i+j}{2} \rfloor$. Match to one of the following cases:

a) If $a_k = b_k$ then repeat the algorithm with $i' = k$ and $j = j$

b) If $a_k < b_k$, consider the following cases:

i) If $k = 1$ or $a_{k-1} = b_{k-1}$, return a_k

ii) Else, repeat the algorithm with $i' = i$ and $j' = k$

c) If $a_k > b_k$, consider the following cases:

i) If $k = n$ or $a_{k+1} = b_{k+1}$, return a_k

ii) Else, repeat the algorithm with $i' = k$ and $j' = j$

Call the algorithm on A and B setting $i = 0$ and $j = n + 2$

Explanation of correctness

Here are some key facts:

1) If B aligns with A up to n, then a_{n+1} must be missing.

2) If $a_x < b_x$, then some there exists some $y \leq x$, such that a_y is missing.
Reasoning: Assume this claim is false. Then $\{a_1, ..., a_x\} \subseteq \{b_1, ..., b_{x-1}\}$. However, $|\{a_1, ..., a_x\}| > |\{b_1, ..., b_{x-1}\}|$. Hence, our assumption is false.

3) If $a_x > b_x$, then some there exists some $y \geq x$, such that a_y is missing.
Reasoning: Assume this claim is false. Then $\{a_x, ..., a_{n+1}\} \subseteq \{b_{x+1}, ..., b_n\}$. However, $|\{a_x, ..., a_{n+1}\}| > |\{b_{x+1}, ..., b_n|$. Hence, our assumption is false.

4) If $a_x < b_x$ and ($x = 1$ or $a_{x-1} = b_{x-1}$) then a_x is missing.
Reasoning: Assume $a_x < b_x$ and $x = 1$. Any b_y equal to a_x would have to come before b_x but b_x is the first element, so this assumption is false. Assume $a_x < b_x$ and $a_{x-1} = b_{x-1}$. Any b_y equal to a_x would have to come before b_x but after b_{x-1} since $b_x > a_x = b_y > a_{x-1} = b_{x-1}$. However, b_x and b_{x-1} are next to each other and neither equal to a_i. Therefore, our assumption is false.

5) If $a_x > b_x$, and ($a_{x+1} = b_{x+1}$ or $x = n$) then a_x is missing. (*Reasoning*: Same as above)

Hence, we know that while $a_x \geq b_x$, there exists some $y \geq x$, such that a_y does not exist. Likewise if $a_x < b_x$, there exists some $y \leq x$, such that a_y does not exist. These facts allow us to perform a modified binary search where with each step check if the current a_k is missing according to the conditions above. If a_k does not meet these conditions, we know which half must contain some missing a_y. Hence, in the worst case, we repeatedly half the range that must contain a missing element until we are left with just one necessarily missing element.

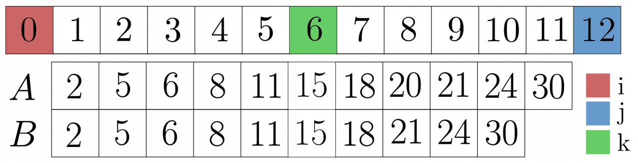
Run time analysis

The run time of this algorithm is $O(logn)$.

Each time the algorithm recurses, either the process terminates or the range of indexes being searched is reduced by (approximately) half until only one element remains. Hence in the worst case, $log_2(n)$ recurses are needed so the algorithm is called $log_2(n) + 1$ times. During each call, the algorithm checks and compares the current value of a_k and b_k and potentially does the same for EITHER a_{k+1} and b_{k+1} OR a_{k-1} and b_{k-1}. If 2 checks and a comparison is taken to be 3 steps then the algorithm takes at most 6 steps per call. In other words, algorithm makes at most $6log_2n + 6$ steps. This is $O(logn)$.

Running Algorithm on Example:

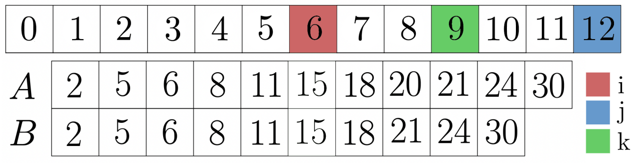
We start with $n = 10$ so we call the algorithm with $i = 0$ and $j = 12$. This results in $k = 6$:



	🟥 0	1	2	3	4	5	🟩 6	7	8	9	10	11	🟦 12
A		2	5	6	8	11	15	18	20	21	24	30	
B		2	5	6	8	11	15	18	21	24	30		

🟥 i
🟦 j
🟩 k

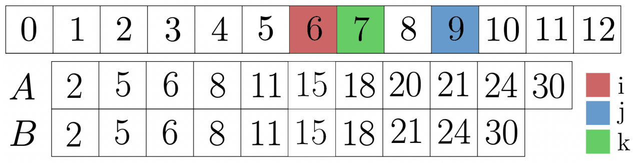
We identify $a_6 = b_6$. Hence, we call the algorithm with $i' = 6$ and $j = 12$. This results in $k = 9$:



	0	1	2	3	4	5	🟥 6	7	8	🟩 9	10	11	🟦 12
A		2	5	6	8	11	15	18	20	21	24	30	
B		2	5	6	8	11	15	18	21	24	30		

🟥 i
🟦 j
🟩 k

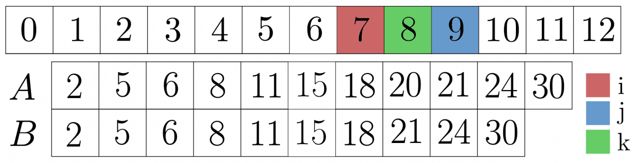
We identify $a_9 < b_9$. We identify that $k \neq 1$ and $a_8 \neq b_8$ Hence, we call the algorithm with $i' = 6$ and $j = 9$ This results in $k = 7$:



	0	1	2	3	4	5	🟥 6	🟩 7	8	🟦 9	10	11	12
A		2	5	6	8	11	15	18	20	21	24	30	
B		2	5	6	8	11	15	18	21	24	30		

🟥 i
🟦 j
🟩 k

We identify $a_7 = b_7$. Hence, we call the algorithm with $i' = 7$ and $j = 9$. This results in $k = 8$:



	0	1	2	3	4	5	6	🟥 7	🟩 8	🟦 9	10	11	12
A		2	5	6	8	11	15	18	20	21	24	30	
B		2	5	6	8	11	15	18	21	24	30		

🟥 i
🟦 j
🟩 k

We identify $a_8 < b_8$. We identify that $a_8 = b_8$. Hence, we return 20.

Problem 5

Solution: This problem can be solved with the following recursive algorithm:

Initialization: Let S be the the set of teams in the tournament. Initialize some data structure (such as a list) to keep track of the number of losses each team from s_1 to s_n experiences starting each team at 0 losses. Pass S into the following recursive function.

Recursive Function: Let C be the set of team passed into the recursive function. Now, match against the following cases:

1) If $|C| > 4k + 2$, divide C into two halves, C_1 and C_2. Call the recursive function of C_1 and designate the set of people the recursive function returns as L. Call the recursive function of C_2 and designated the set of people the recursive function returns as R. Check the outcome of every game between each person in L and each person in R making sure to record each defeat. Return the set of the people who have lost less than $k + 1$ games (thus far).
2) If $|C| \leq 4k + 2$, check the outcome of every game between everybody in C making sure to record each defeat. Return the set of the people who have lost less than $k + 1$ games (thus far).

Final Step: Let F be the set of people that the recursive function returns after passing in S. For each person in F, check every game played and add up the number of loses (from scratch). Discard anyone who has lost more than k games. Return any remaining people.

The run time of this algorithm will be $O((k + 1)^2 n)$

Proof of Correctness:

There are only possible two ways in which this algorithm could be wrong:

Case 1: It returns someone who has lost more than k times

Case 2: It does not return someone who has lost less than $k + 1$ games

Claim: Case 1 cannot happen.

Proof: Notice that in the final step any team who has lost more than k games will be discarded. Hence, it is impossible for a team to be returned by the algorithm that has lost more than k games.

Claim: Case 2 cannot happen.

Proof: There is no case in which a person who has lost less than $k + 1$ would not make it F. This can be seen in that fact that the recursive function always returns people who have less lost $k + 1$ games (at that time). Clearly, any team who has lost less than $k + 1$ in total will not at intermediate step be thought to have lost $k + 1$ or more games.

Any team that makes it to F and has lost less than $k + 1$ games will be returned in the final group of people. Hence, our claim is correct.

Hence, both ways the algorithm might fail are not possible, so the algorithm is correct. □

Run time Analysis:

Claim: If $2k+2$ teams play each other, at least one the teams must lose $k+1$ times

Proof: Assume the number of teams in a game is even. We know that the total number of losses will be equal to the number of games played. This is equal to $\frac{n(n-1)}{2}$. In order to minimize the maximum number of losses any one team suffers, we can try to distribute these losses as uniformly through the population as possible. In other words, n teams will experience $\frac{n-1}{2}$. losses. This last quantity is not a whole number. However, we easily rewrite things to show that $\frac{n}{2}$ teams will experience $\frac{n-2}{2}$ losses and $\frac{n}{2}$ teams will experience $\frac{n-2}{2}+1$ losses. As evidence that this is equal to the number of total losses:

$$n\frac{n-2}{2}+n(\frac{n-2}{2}+1)=2n\frac{n-2}{2}+n=\frac{n^2-2n}{2}+\frac{n}{2}=\frac{n(n-1)}{2}$$

Let $n=2k+2$. We can conclude that in the case that minimizes the maximum number of losses any one player experiences, $k+1$ players experience $k+1$ losses. Hence, our claim is true.

Key Fact: In a group of m teams that all play each other (where $m>2k+1$), we can always find someone who has lost $k+1$ or more games. This follows from the previous claim.

Claim: The maximum number of teams returned from 2) in the recursive function is $2k+1$.

Proof: We know that given any C in 2), $2k+1\leq|C|\leq 4k+2$. In other word, $|C|$ is always greater than or equal to $2k+1$. At the end of 2), all teams in C will play each other and then we return all teams who did not lose $k+1$ or more times. Suppose we remove all teams who lost $k+1$ or more times one by one. As long as the group of teams remaining is greater than $2k+1$, we know from the key fact that we can find a team to remove that lost $k+1$ or more games. Hence, we know we there will be at least $|C|-(2k+1)$ teams that lost $k+1$ or more times. Hence, the maximum number of teams that will only have lost k or less games by the end of 2) is $2k+1$.

Claim: The maximum number of teams returned at any step in the recursive function is $2k+1$.

Proof: Suppose we match to 1) and have L and R. By the end of this call of the recursive function, everyone in the union of L and R will have played each other. Hence, we can use the same reasoning as in the last step to conclude that there will be at least $(|L|+|R|)-(2k+1)$ teams that will have lost $k+1$ or more games by the end of the 1). Hence, the maximum number of teams that get returned from 1) is $2k+1$. Hence, our claim is true.

Claim: If the recursive functions matches to 1) then at most $2k^2+6k+\frac{5}{2}$ steps are needed.

If the function matches to 1) then at most $2k+1$ teams play $2k+1$ other teams. Hence this many games are checked:

$$\frac{(2k+1)(2k+1)}{2}=2k^2+2k+\frac{1}{2}$$

For each game check, both involved teams need their loss counter updated. Finally at the end of the function call, the total losses of each of the (at max) $4k+2$ players is checked and compared.

Claim: The recurrence relationship is $T(n) = 2T(\frac{n}{2}) + C$ where C is the number of steps that are taken at each level. This will be equal to $|L| \cdot |R|$ (except at the base level).

Proof: If the recursive function calls itself, it will do so twice with half the number of people.

Claim: Solving this recurrence relation, we find that the recursive part of the process takes $O(k^2 n)$.

Proof:

$T(n) = 2T(\frac{n}{2}) + C$

$T(n) = 4T(\frac{n}{4}) + 3C$

$T(n) = 8T(\frac{n}{8}) + 7C$

$T(n) = 2^i T(\frac{n}{2^i}) + (2^i - 1)C$ (note: using i instead of k)

Let $i = log_2(n)$ so that $T(\frac{n}{2^i}) = 1$

$T(n) = 2^{log_2(n)}T(1) + (2^{log_2(n)} - 1)C$

$T(n) = nT(1) + (n-1)C$

Let $T(1) = 1$

Some remarks: This is not exact, but it does not change the final big O. The base case will end up being greater than 1 (specifically it will be between $2k+1$ teams and $4k+2$ teams). The number of steps will involve all these teams playing each other. The run time of this base case will be $O(k^2 n)$ which is the same as what we will momentarily conclude for the whole recursive function assuming $T(1) = 1$. Hence, it does not affect the limiting behavior of the recursive function which is what we care about.

$T(n) = n + (n-1)C$

We know that the maximum value of C is $2k^2 + 6k + \frac{5}{2}$. Plugging this in, we get:

$T(n) = n + (n-1)(2k^2 + 6k + \frac{5}{2})$

Thus the recursive part of this process takes $O(k^2 n)$ time.

Claim: The run time of final step is $O(kn)$

Proof: We know that at most $2k+1$ people will be returned by the recursive function. Hence, the the maximum value of $|F|$ is $2k+1$. For each team in F, we are checking the result of $n-1$ games, adding these games to a sum and then checking if the final sum is less than or equal to k. Hence, this section will take at most $n(2k+1)$ steps, meaning this section is $O(kn)$

Claim: The whole algorithm is $O((k+1)^2 n)$

Proof: This can be conclude by put everything and remembering that if $k = 0$, we must have $O(1)$.