G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

Proof: The opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that this path is removed, so nothing is gained.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

Strategy: It follows that such paths only come into play once there are no other options remaining. We can thus use the parity of the total number of goods to determine which player has to tank the net loss of each such path, and we can then remove all such paths from the game.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

Strategy: It follows that such paths only come into play once there are no other options remaining. We can thus use the parity of the total number of goods to determine which player has to tank the net loss of each such path, and we can then remove all such paths from the game.

Furthermore, we can do the same whenever the last value of a path is larger than the previous, since those two goods will eventually reduce to a path of length two.

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

Strategy: We may thus combine the first three goods into one good with value $a - b + c$.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

Strategy: We may thus combine the first three goods into one good with value $a - b + c$.

We can furthermore do the same whenever any value besides the last value of a path is larger than the previous, since the path will eventually reduce for that value to become the second value of the path.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

Strategy: We may thus combine the first three goods into one good with value $a - b + c$.

We can furthermore do the same whenever any value besides the last value of a path is larger than the previous, since the path will eventually reduce for that value to become the second value of the path.

Combined with Observation 2, we may thus assume that no value in any path is larger than the previous!

Simpler case: Consider if each rooted tree is a path.

Observations 2 & 3: We may assume that no value in any path is larger than the previous.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observations 2 & 3: We may assume that no value in any path is larger than the previous.

Strategy: It is now always possible to grab the good with the highest value, so we can apply the greedy strategy!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observations 2 & 3: We may assume that no value in any path is larger than the previous.

Strategy: It is now always possible to grab the good with the highest value, so we can apply the greedy strategy!

Observation 4: Since there is never a reason to grab anything other than the good with the highest value, we can combine all the paths into a single path by sorting all the values from largest to smallest!

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

The entire tree thus reduces to the same path, but with the root added to the front.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

The entire tree thus reduces to the same path, but with the root added to the front.

Even if this path is no longer sorted from largest to smallest, it can again be reduced until it is.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

The entire tree thus reduces to the same path, but with the root added to the front.

Even if this path is no longer sorted from largest to smallest, it can again be reduced until it is.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest!

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Then merge these paths into one by sorting all combined values from largest to smallest, and add the value a to the front of the path.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Then merge these paths into one by sorting all combined values from largest to smallest, and add the value a to the front of the path.

Finally, while the first value of the path is smaller than the second, apply Observations 2 and 3 to reduce the path.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Then merge these paths into one by sorting all combined values from largest to smallest, and add the value a to the front of the path.

Finally, while the first value of the path is smaller than the second, apply Observations 2 and 3 to reduce the path.

Running time: Since every value may need to get sorted n times, the total running time is $\Omega(n^2 \log n)$, which is too slow!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

The fact that the heaps get smaller due to the reduction step only reduces the number of times a value moves into a larger heap.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

The fact that the heaps get smaller due to the reduction step only reduces the number of times a value moves into a larger heap.

Amortized, we thus have to move a value to a larger heap at most $n \log_2 n$ times.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

The fact that the heaps get smaller due to the reduction step only reduces the number of times a value moves into a larger heap.

Amortized, we thus have to move a value to a larger heap at most $n \log_2 n$ times.

Since heap operations cost $\mathcal{O}(\log n)$ time, the total running time is $\mathcal{O}(n(\log n)^2)$, which is fast enough!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a heap data structure, and apply small-to-large merging to the heaps.

Running time: $\mathcal{O}(n(\log n)^2)$.

Even faster: With a custom heap implementation, you can insert all k elements of a heap of size k into a larger heap of size n in $\mathcal{O}(n \log(n/k))$ time.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a heap data structure, and apply small-to-large merging to the heaps.

Running time: $\mathcal{O}(n(\log n)^2)$.

Even faster: With a custom heap implementation, you can insert all k elements of a heap of size k into a larger heap of size n in $\mathcal{O}(n \log(n/k))$ time.

Combined with small-to-large merging, this results in a total running time of $\mathcal{O}(n \log n)$!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a heap data structure, and apply small-to-large merging to the heaps.

Running time: $\mathcal{O}(n(\log n)^2)$.

Even faster: With a custom heap implementation, you can insert all k elements of a heap of size k into a larger heap of size n in $\mathcal{O}(n \log(n/k))$ time.

Combined with small-to-large merging, this results in a total running time of $\mathcal{O}(n \log n)$!

However, in practice, the difference turns out to be immeasurable. In fact, only reading the input already already takes up half the time of the $\mathcal{O}(n(\log n)^2)$ solution!

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.
With at least three values a, b, c with $b > a$, you can reduce those three values to a single value of $a - b + c$. A special case is needed for two values.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.
With at least three values a, b, c with $b > a$, you can reduce those three values to a single value of $a - b + c$. A special case is needed for two values.

Running time: Must use heaps and small-to-large merging to obtain $\mathcal{O}(n \log^2 n)$.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.
With at least three values a, b, c with $b > a$, you can reduce those three values to a single value of $a - b + c$. A special case is needed for two values.

Running time: Must use heaps and small-to-large merging to obtain $\mathcal{O}(n \log^2 n)$.

Statistics: 4 submissions, 0 accepted, 2 unknown