NCNA 2017 Solution Slides

NCNA Judges
Problem Set Developers

- Dr. Larry Pyeatt (Chief Judge)
- Bryce Sandlund (Associate Chief Judge)
- Robert Hochberg
- Bowen Yu
- Bruce Elenbogen
- Ivor Page
- Antonio Molina
- Menghui Wang
- Andrew Morgan
- ECNA 2017 Developers (IsaHasa and Sheba’s Amoeba’s), specifically John Bonomo and Bob Roos
- The Kattis Team, specifically Greg Hamerly and Fredrik Niemela
- NWERC and SWERC, to which these slides were modeled off of
Problem

Given a vertical stack of Zebras (Z’s) and Ocelots (O’s), determine how many steps until they all turn into Z’s, given that at each step, the lowest O turns into a Z, and all Z’s below it turn into O’s.
### Problem

Given a vertical stack of Zebras (Z’s) and Ocelots (O’s), determine how many steps until they all turn into Z’s, given that at each step, the lowest O turns into a Z, and all Z’s below it turn into O’s.

### Solution

- Interpret each O as a 1 and each Z as a 0. Then the operation is just “subtract 1” in binary.
Problem

Given a vertical stack of Zebras (Z’s) and Ocelots (O’s), determine how many steps until they all turn into Z’s, given that at each step, the lowest O turns into a Z, and all Z’s below it turn into O’s.

Solution

- Interpret each O as a 1 and each Z as a 0. Then the operation is just “subtract 1” in binary.
- The answer is the decimal value of the given binary string.
Problem
Given a vertical stack of Zebras (Z’s) and Ocelots (O’s), determine how many steps until they all turn into Z’s, given that at each step, the lowest O turns into a Z, and all Z’s below it turn into O’s.

Solution
- Interpret each O as a 1 and each Z as a 0. Then the operation is just “subtract 1” in binary.
- The answer is the decimal value of the given binary string.

Pitfalls
- Simulation gets TLE.

Problem Author: Robert Hochberg
H - Zebras and Ocelots

Problem
Given a vertical stack of Zebras (Z’s) and Ocelots (O’s), determine how many steps until they all turn into Z’s, given that at each step, the lowest O turns into a Z, and all Z’s below it turn into O’s.

Solution
- Interpret each O as a 1 and each Z as a 0. Then the operation is just “subtract 1” in binary.
- The answer is the decimal value of the given binary string.

Pitfalls
- Simulation gets TLE.
- Need to use 64-bit integers.
H - Zebras and Ocelots

Problem

Given a vertical stack of Zebras (Z’s) and Ocelots (O’s), determine how many steps until they all turn into Z’s, given that at each step, the lowest O turns into a Z, and all Z’s below it turn into O’s.

Solution

- Interpret each O as a 1 and each Z as a 0. Then the operation is just “subtract 1” in binary.
- The answer is the decimal value of the given binary string.

Pitfalls

- Simulation gets TLE.
- Need to use 64-bit integers.

Statistics: 824 submissions, 112 accepted.
Problem

Find the total time to pick up all letters of a phrase, running around a circular disk.
Problem
Find the total time to pick up all letters of a phrase, running around a circular disk.

Solution
1. Figure out circumference of circle.
Problem

Find the total time to pick up all letters of a phrase, running around a circular disk.

Solution

1. Figure out circumference of circle.
2. Shortest path between two letters follows the shorter distance around the circle.
Problem
Find the total time to pick up all letters of a phrase, running around a circular disk.

Solution
1. Figure out circumference of circle.
2. Shortest path between two letters follows the shorter distance around the circle.
3. Calculate time to travel between all consecutive pairs of letters in aphorism.
Problem
Find the total time to pick up all letters of a phrase, running around a circular disk.

Solution
1. Figure out circumference of circle.
2. Shortest path between two letters follows the shorter distance around the circle.
3. Calculate time to travel between all consecutive pairs of letters in aphorism.

Statistics: 170 submissions, 114 accepted.
Problem

Count the number of amoebas contained entirely within one another in a 2D grid.

Figure: Two Petri dishes, each with four amoebas.
G - Sheba’s Amoebas

Problem
Count the number of amoebas contained entirely within one another in a 2D grid.

Solution
Run a modified flood fill:
Problem
Count the number of amoebas contained entirely within one another in a 2D grid.

Solution
Run a modified flood fill:

1. Iterate over every pixel of the grid.
Problem
Count the number of amoebas contained entirely within one another in a 2D grid.

Solution
Run a modified flood fill:
1. Iterate over every pixel of the grid.
2. If the pixel is black, run DFS from this point, recursively marking all black neighbors as visited.
Problem
Count the number of amoebas contained entirely within one another in a 2D grid.

Solution
Run a modified flood fill:

1. Iterate over every pixel of the grid.
2. If the pixel is black, run DFS from this point, recursively marking all black neighbors as visited.
3. Answer is the number of times DFS is restarted.
Problem

Count the number of amoebas contained entirely within one another in a 2D grid.

Solution

Run a modified flood fill:

1. Iterate over every pixel of the grid.
2. If the pixel is black, run DFS from this point, recursively marking all black neighbors as visited.
3. Answer is the number of times DFS is restarted.

Statistics: 143 submissions, 77 accepted.
C - Urban Design

Problem

Given a set of infinite lines in the 2D plane and queries that consist of two regions defined by points within these regions, determine if these regions should get different or same designations, given that regions immediately across a line from one another get different designations.
Think of the regions as lines are added into the plane one-by-one.
Think of the regions as lines are added into the plane one-by-one.
Think of the regions as lines are added into the plane one-by-one.
Think of the regions as lines are added into the plane one-by-one.
Think of the regions as lines are added into the plane one-by-one.
Think of the regions as lines are added into the plane one-by-one. As you pass through a line, the designation of the region changes.
Solution

- Draw a line segment between the two given query points.
Solution

- Draw a line segment between the two given query points.
- If the number of infinite lines this segment intersects with is even, then the answer is “same”, and if it is odd, then the answer is “different.”
Solution

- Draw a line segment between the two given query points.
- If the number of infinite lines this segment intersects with is even, then the answer is “same”, and if it is odd, then the answer is “different.”

Problem

Given the distance between every pair of nodes in a weighted tree, recover the tree.

Solution

Observation: The smallest distance must be a tree edge.
So must the next smallest.
So must the next, if it does not create a cycle.
This is Kruskal's minimum spanning tree algorithm. The answer is the MST of the distance matrix.

Time complexity: $O(n^2 \log n)$. An $O(n^3)$ algorithm should TLE.

Statistics: 131 submissions, 17 accepted.

Problem Author: Bryce Sandlund
Problem
Given the distance between every pair of nodes in a weighted tree, recover the tree.

Solution
- Observation: The smallest distance must be a tree edge.
Problem

Given the distance between every pair of nodes in a weighted tree, recover the tree.

Solution

- Observation: The smallest distance must be a tree edge.
- So must the next smallest.
Problem
Given the distance between every pair of nodes in a weighted tree, recover the tree.

Solution
- Observation: The smallest distance must be a tree edge.
- So must the next smallest.
- So must the next, if it does not create a cycle.
Problem
Given the distance between every pair of nodes in a weighted tree, recover the tree.

Solution

- Observation: The smallest distance must be a tree edge.
- So must the next smallest.
- So must the next, if it does not create a cycle.
- This is Kruskal’s minimum spanning tree algorithm. The answer is the MST of the distance matrix.
Problem
Given the distance between every pair of nodes in a weighted tree, recover the tree.

Solution
- Observation: The smallest distance must be a tree edge.
- So must the next smallest.
- So must the next, if it does not create a cycle.
- This is Kruskal’s minimum spanning tree algorithm. The answer is the MST of the distance matrix.
- Time complexity: $O(n^2 \log n)$. An $O(n^3)$ algorithm should TLE.
Problem
Given the distance between every pair of nodes in a weighted tree, recover the tree.

Solution
- Observation: The smallest distance must be a tree edge.
- So must the next smallest.
- So must the next, if it does not create a cycle.
- This is Kruskal’s minimum spanning tree algorithm. The answer is the MST of the distance matrix.
- Time complexity: $O(n^2 \log n)$. An $O(n^3)$ algorithm should TLE.

Statistics: 131 submissions, 17 accepted.
Problem
Given a set of pokemon and where they appear in the 2D plane, determine the shortest distance required to collect all unique pokemon.

Solution
This is the traveling salesman problem with the twist that at a particular vertex there may be multiple pokemon that can be collected.

TSP DP on locations:
\[ O\left(n^2 2^n\right) \approx 400 \text{ million iterations}, \text{ still gets AC.} \]

Faster solution is to do subset DP on the set of pokemon collected.

\[ DP(i,S) := \text{minimum distance to visit pokemon in set } S, \text{ ending at location } i. \]

\[ DP(i,S) = \min_j \left( DP(j,S\{\text{pokemon at location } i}\}) + \text{dist}(j,i) \right). \]

Time complexity: \( O(n^2 2^D) \), where \( D \) is the number of distinct pokemon.
Problem
Given a set of pokemon and where they appear in the 2D plane, determine the shortest distance required to collect all unique pokemon.

Solution
- This is the traveling salesman problem with the twist that at a particular vertex there may be multiple pokemon that can be collected.
Problem
Given a set of pokemon and where they appear in the 2D plane, determine the shortest distance required to collect all unique pokemon.

Solution
- This is the traveling salesman problem with the twist that at a particular vertex there may be multiple pokemon that can be collected.
- TSP DP on locations: \(O(n^22^n) \approx 400\) million iterations, still gets AC.
Problem
Given a set of pokemon and where they appear in the 2D plane, determine the shortest distance required to collect all unique pokemon.

Solution
- This is the traveling salesman problem with the twist that at a particular vertex there may be multiple pokemon that can be collected.
- TSP DP on locations: $O(n^22^n) \approx 400$ million iterations, still gets AC.
- Faster solution is to do subset DP on the set of pokemon collected.
Problem
Given a set of pokemon and where they appear in the 2D plane, determine the shortest distance required to collect all unique pokemon.

Solution
- This is the traveling salesman problem with the twist that at a particular vertex there may be multiple pokemon that can be collected.
- TSP DP on locations: $O(n^2 2^n) \approx 400$ million iterations, still gets AC.
- Faster solution is to do subset DP on the set of pokemon collected.
- $DP(i, S) :=$ minimum distance to visit pokemon in set $S$, ending at location $i$. 

Problem Author: Bruce Elenbogen
Problem
Given a set of pokemon and where they appear in the 2D plane, determine the shortest distance required to collect all unique pokemon.

Solution
- This is the traveling salesman problem with the twist that at a particular vertex there may be multiple pokemon that can be collected.
- TSP DP on locations: $O(n^22^n) \approx 400$ million iterations, still gets AC.
- Faster solution is to do subset DP on the set of pokemon collected.
- $DP(i, S) := \text{minimum distance to visit pokemon in set } S, \text{ ending at location } i$.
- $DP(i, S) = \min_j DP(j, S \setminus \{\text{pokemon at location } i\}) + dist(j, i)$. 
Problem
Given a set of pokemon and where they appear in the 2D plane, determine the shortest distance required to collect all unique pokemon.

Solution
- This is the traveling salesman problem with the twist that at a particular vertex there may be multiple pokemon that can be collected.
- TSP DP on locations: $O(n^2 2^n) \approx 400$ million iterations, still gets AC.
- Faster solution is to do subset DP on the set of pokemon collected.
- $DP(i, S) :=$ minimum distance to visit pokemon in set $S$, ending at location $i$.
- $DP(i, S) = \min_j DP(j, S \setminus \{\text{pokemon at location } i\}) + \text{dist}(j, i)$.
- Time complexity: $O(n^2 2^D)$, where $D$ is the number of distinct pokemon.
Statistics: 73 submissions, 7 accepted.
Problem

Given a set of is-a and has-a relationships, answer is-a and has-a queries, defined as follows:

1. A is-a B if and only if there is a path of is-a relationships from A to B
2. A has-a B if and only if there is a path of is-a and has-a relationships from A to B that includes at least one has-a relationship.
Can be modeled as a graph with two types of edges. Ex, Sample Input 1:
Solution

- Can answer each query via careful DFS: $O(nm)$.
Solution

- Can answer each query via careful DFS: \( O(nm) \).
- Alternatively, can preprocess all relationships via clever application of Floyd-Warshall. The algorithm is as follows:

```java
for (int k = 0; k < D; ++k) {
    for (int i = 0; i < D; ++i) {
        for (int j = 0; j < D; ++j) {
            is_a[i][j] = is_a[i][j] || (is_a[i][k] && is_a[k][j]);
            has_a[i][j] = has_a[i][j] || (has_a[i][k] && has_a[k][j]);
            has_a[i][j] = has_a[i][j] || (is_a[i][k] && has_a[k][j]);
            has_a[i][j] = has_a[i][j] || (has_a[i][k] && is_a[k][j]);
        }
    }
}
```

Time complexity: \( O(D^3 + m) \), where \( D \) is the number of distinct classes, which is at most 500.

Statistics: 215 submissions, 4 accepted.

Problem Author: John Bonomo
Solution

- Can answer each query via careful DFS: $O(nm)$.

- Alternatively, can preprocess all relationships via clever application of Floyd-Warshall. The algorithm is as follows:
  
  ```java
  for (int k = 0; k < D; ++k) {
      for (int i = 0; i < D; ++i) {
          for (int j = 0; j < D; ++j) {
              is_a[i][j] = is_a[i][j] || (is_a[i][k] && is_a[k][j]);
              has_a[i][j] = has_a[i][j] || (has_a[i][k] && has_a[k][j]);
              has_a[i][j] = has_a[i][j] || (is_a[i][k] && has_a[k][j]);
              has_a[i][j] = has_a[i][j] || (has_a[i][k] && is_a[k][j]);
          }
      }
  }
  
  Time complexity: $O(D^3 + m)$, where $D$ is the number of distinct classes, which is at most 500.
Solution

- Can answer each query via careful DFS: $O(nm)$.
- Alternatively, can preprocess all relationships via clever application of Floyd-Warshall. The algorithm is as follows:

```java
for (int k = 0; k < D; ++k) {
    for (int i = 0; i < D; ++i) {
        for (int j = 0; j < D; ++j) {
            is_a[i][j] = is_a[i][j] || (is_a[i][k] && is_a[k][j]);
            has_a[i][j] = has_a[i][j] || (has_a[i][k] && has_a[k][j]);
            has_a[i][j] = has_a[i][j] || (is_a[i][k] && has_a[k][j]);
            has_a[i][j] = has_a[i][j] || (has_a[i][k] && is_a[k][j]);
        }
    }
}
```

- Time complexity: $O(D^3 + m)$, where $D$ is the number of distinct classes, which is at most 500.

Statistics: 215 submissions, 4 accepted.
Problem

Balance a chemical equation.
Make a system of equations. Each coefficient is an unknown and for every unique atom, we get an equation, since the number of atoms of each type is preserved through the chemical reaction.
Solution

- Make a system of equations. Each coefficient is an unknown and for every unique atom, we get an equation, since the number of atoms of each type is preserved through the chemical reaction.

- Define matrix $A$ where $A_{ij} =$ number of atoms of type $i$ in molecule $j$. 

Example: Sample Input 1:

$$\begin{align*}
H_2O + CO_2 &\rightarrow O_2 + C_6H_{12}O_6 \\
yields &\begin{bmatrix}
H & 0 & 0 & -12 \\
O & 1 & 2 & -2 \\
C & 0 & 1 & -6
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{bmatrix} = 
\begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}
\end{align*}$$
Solution

- Make a system of equations. Each coefficient is an unknown and for every unique atom, we get an equation, since the number of atoms of each type is preserved through the chemical reaction.
- Define matrix $A$ where $A_{ij} =$ number of atoms of type $i$ in molecule $j$.
- Ex, Sample Input 1:

$$H_2O + CO_2 \rightarrow O_2 + C_6H_{12}O_6$$

yields

$$
\begin{bmatrix}
2 & 0 & 0 & -12 \\
1 & 2 & -2 & -6 \\
0 & 1 & 0 & -6 \\
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
\end{bmatrix}
=
\begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
$$

Problem Author: Larry Pyeatt
Solution

Put $A$ in reduced row-echelon form, ex:

\[
A_{rref} = \begin{bmatrix}
H & 1 & 0 & 0 & -6 \\
O & 0 & 1 & 0 & -6 \\
C & 0 & 0 & 1 & -6
\end{bmatrix}
\]

This requires us to work over a field. Input is small so floating point error is negligible, therefore we can use doubles.
Solution

- Put $A$ in reduced row-echelon form, ex:

$$A_{\text{rref}} = \begin{bmatrix}
H & 1 & 0 & 0 & -6 \\
O & 0 & 1 & 0 & -6 \\
C & 0 & 0 & 1 & -6
\end{bmatrix}$$

This requires us to work over a field. Input is small so floating point error is negligible, therefore we can use doubles.

- The problem description guarantees there will be exactly one free variable, since there is a unique minimal solution.
Solution

- Put $A$ in reduced row-echelon form, ex:

$$A_{rref} = \begin{bmatrix} H & | & 1 & 0 & 0 & -6 \\ \hline O & | & 0 & 1 & 0 & -6 \\ C & | & 0 & 0 & 1 & -6 \end{bmatrix}$$

This requires us to work over a field. Input is small so floating point error is negligible, therefore we can use doubles.

- The problem description guarantees there will be exactly one free variable, since there is a unique minimal solution.

- Set this free variable to the smallest positive value that yields an integer solution.
Solution

- Put $A$ in reduced row-echelon form, ex:

$$A_{rref} = \begin{bmatrix} H & 1 & 0 & 0 & -6 \\ O & 0 & 1 & 0 & -6 \\ C & 0 & 0 & 1 & -6 \end{bmatrix}$$

This requires us to work over a field. Input is small so floating point error is negligible, therefore we can use doubles.

- The problem description guarantees there will be exactly one free variable, since there is a unique minimal solution.

- Set this free variable to the smallest positive value that yields an integer solution.

Statistics: 5 submissions, 2 accepted.
Problem

Given pairs of integers $t_i$ and $h_i$ representing a gold store, determine the maximum number of gold stores that can be visited, if store $i$ takes $t_i$ time to visit and needs to be visited prior to time $h_i$.

Solution

Observation: Given a set of stores to visit, the best order in which to visit them is in increasing order of $h_i$.

Observation: If we prefer stores with smaller $t_i$, we leave more room for other stores to be visited.

Greedy algorithm: Sort stores by increasing $t_i$. Maintain a feasible solution $F$. Add store $i$ to $F$ if doing so does not destroy feasibility.

Checking feasibility in $O(\log n)$ time may require a lazy segment tree or balanced binary search tree. Advanced data structures can be avoided if you are clever.

Problem Author: Bryce Sandlund
Problem

Given pairs of integers $t_i$ and $h_i$ representing a gold store, determine the maximum number of gold stores that can be visited, if store $i$ takes $t_i$ time to visit and needs to be visited prior to time $h_i$.

Solution

- Observation: Given a set of stores to visit, the best order in which to visit them is in increasing order of $h_i$. 
Problem

Given pairs of integers $t_i$ and $h_i$ representing a gold store, determine the maximum number of gold stores that can be visited, if store $i$ takes $t_i$ time to visit and needs to be visited prior to time $h_i$.

Solution

- Observation: Given a set of stores to visit, the best order in which to visit them is in increasing order of $h_i$.
- Observation: If we prefer stores with smaller $t_i$, we leave more room for other stores to be visited.
Problem

Given pairs of integers \( t_i \) and \( h_i \) representing a gold store, determine the maximum number of gold stores that can be visited, if store \( i \) takes \( t_i \) time to visit and needs to be visited prior to time \( h_i \).

Solution

- Observation: Given a set of stores to visit, the best order in which to visit them is in increasing order of \( h_i \).
- Observation: If we prefer stores with smaller \( t_i \), we leave more room for other stores to be visited.
- Greedy algorithm: Sort stores by increasing \( t_i \). Maintain a feasible solution \( F \). Add store \( i \) to \( F \) if doing so does not destroy feasibility.
Problem

Given pairs of integers $t_i$ and $h_i$ representing a gold store, determine the maximum number of gold stores that can be visited, if store $i$ takes $t_i$ time to visit and needs to be visited prior to time $h_i$.

Solution

- Observation: Given a set of stores to visit, the best order in which to visit them is in increasing order of $h_i$.
- Observation: If we prefer stores with smaller $t_i$, we leave more room for other stores to be visited.
- Greedy algorithm: Sort stores by increasing $t_i$. Maintain a feasible solution $F$. Add store $i$ to $F$ if doing so does not destroy feasibility.
- Checking feasibility in $O(\log n)$ time may require a lazy segment tree or balanced binary search tree.
Problem

Given pairs of integers $t_i$ and $h_i$ representing a gold store, determine the maximum number of gold stores that can be visited, if store $i$ takes $t_i$ time to visit and needs to be visited prior to time $h_i$.

Solution

- Observation: Given a set of stores to visit, the best order in which to visit them is in increasing order of $h_i$.
- Observation: If we prefer stores with smaller $t_i$, we leave more room for other stores to be visited.
- Greedy algorithm: Sort stores by increasing $t_i$. Maintain a feasible solution $F$. Add store $i$ to $F$ if doing so does not destroy feasibility.
- Checking feasibility in $O(\log n)$ time may require a lazy segment tree or balanced binary search tree.
- Advanced data structures can be avoided if you are clever.
\textit{O(n \log n)} Solution with c++ set

1. Maintain $F$ as a set of pairs $(h_i, t_i)$
**F - Atlantis**

**$O(n \log n)$ Solution with c++ set**

1. Maintain $F$ as a set of pairs $(h_i, t_i)$
2. To see if store $i$ can be added to $F$, we iterate down the tree starting at the first store scheduled to end before $h_i$, removing the pairs and keeping track of the sum of $t_j$’s of the removed intervals.

If for any $j$, $\sum t_j + t_i \leq h_i$, we can add store $i$. Insert $(h_i, \sum t_j + t_i)$ back into the tree.

If we get to the beginning of the set, we cannot add store $i$. Insert $(h_i, \sum t_j)$ back into the tree.

If we add store $i$, inserting $(h_i, \sum t_j + t_i)$ into the tree is equivalent to scheduling store $i$ right before $h_i$ and pushing everything else earlier to make room.

If we do not add store $i$, we will not be able to add any store $i'$ before time $h_i$, so where stores before $h_i$ are scheduled in $F$ is no longer relevant.
1. Maintain $F$ as a set of pairs $(h_i, t_i)$

2. To see if store $i$ can be added to $F$, we iterate down the tree starting at the first store scheduled to end before $h_i$, removing the pairs and keeping track of the sum of $t_j$’s of the removed intervals.

3. If for any $j$, $\sum_j t_j + t_i \leq h_i$, we can add store $i$. Insert $(h_i, \sum_j t_j + t_i)$ back into the tree.
**O(n log n) Solution with c++ set**

1. Maintain $F$ as a set of pairs $(h_i, t_i)$

2. To see if store $i$ can be added to $F$, we iterate down the tree starting at the first store scheduled to end before $h_i$, removing the pairs and keeping track of the sum of $t_j$’s of the removed intervals.

3. If for any $j$, $\sum_j t_j + t_i \leq h_i$, we can add store $i$. Insert $(h_i, \sum_j t_j + t_i)$ back into the tree.

4. If we get to the beginning of the set, we cannot add store $i$. Insert $(h_i, \sum_j t_j)$ back into the tree.
### $O(n \log n)$ Solution with C++ set

1. Maintain $F$ as a set of pairs $(h_i, t_i)$

2. To see if store $i$ can be added to $F$, we iterate down the tree starting at the first store scheduled to end before $h_i$, removing the pairs and keeping track of the sum of $t_j$’s of the removed intervals.

3. If for any $j$, $\sum_j t_j + t_i \leq h_i$, we can add store $i$. Insert $(h_i, \sum_j t_j + t_i)$ back into the tree.

4. If we get to the beginning of the set, we cannot add store $i$. Insert $(h_i, \sum_j t_j)$ back into the tree.

If we add store $i$, inserting $(h_i, \sum_j t_j + t_i)$ into the tree is equivalent to scheduling store $i$ right before $h_i$ and pushing everything else earlier to make room.
### $O(n \log n)$ Solution with c++ set

1. Maintain $F$ as a set of pairs $(h_i, t_i)$

2. To see if store $i$ can be added to $F$, we iterate down the tree starting at the first store scheduled to end before $h_i$, removing the pairs and keeping track of the sum of $t_j$’s of the removed intervals.

3. If for any $j$, $\sum_j t_j + t_i \leq h_i$, we can add store $i$. Insert $(h_i, \sum_j t_j + t_i)$ back into the tree.

4. If we get to the beginning of the set, we cannot add store $i$. Insert $(h_i, \sum_j t_j)$ back into the tree.

If we add store $i$, inserting $(h_i, \sum_j t_j + t_i)$ into the tree is equivalent to scheduling store $i$ right before $h_i$ and pushing everything else earlier to make room.

If we do not add store $i$, we will not be able to add any store $i'$, $i' > i$ before time $h_i$, so where stores before $h_i$ are scheduled in $F$ is no longer relevant.
**F - Atlantis**

---

**$O(n \log n)$ Solution with c++ set**

- Time complexity of checking feasibility in this approach: $O(\log n \cdot (\# \text{ of intervals removed from the tree} + 1))$. 
\(O(n \log n)\) Solution with c++ set

- Time complexity of checking feasibility in this approach: \(O(\log n \cdot (\# \text{ of intervals removed from the tree } + 1))\).
- We only insert one interval per feasibility check, therefore the cost of deletes amortize amongst the adds.
**O(n log n) Solution with c++ set**

- Time complexity of checking feasibility in this approach: \(O(\log n \cdot (\# \text{ of intervals removed from the tree } + 1))\).
- We only insert one interval per feasibility check, therefore the cost of deletes amortize amongst the adds.
- Overall time complexity: \(O(n \log n)\).
Simpler $O(n \log n)$ Solution using Priority Queues

1. Sort the stores by increasing $h_i$. 

2. Maintain a priority queue of keys by $t_i$, largest $t_i$ on top.

3. Add store $i$ to $F$.

4. Pop from the priority queue until $\sum_{j \in F} t_j \leq h_i$.

Without loss of generality assume each $t_i$ is distinct. Let $\text{add}(i)$ be a true or false value denoting whether store $i$ is added in this strategy. Then $\text{add}(i) = \sum_j s_j \cdot t_j$. If $t_j < t_i$ and $t_j + t_i \leq h_i$.

This is the same condition as the originally proposed greedy algorithm.

Time Complexity: $O(n \log n)$.

Statistics: 72 submissions, 0 accepted.
Simpler $O(n \log n)$ Solution using Priority Queues

1. Sort the stores by increasing $h_i$.
2. Maintain a priority queue of $F$ keyed by $t_i$, largest $t_i$ on top.
Simpler $O(n \log n)$ Solution using Priority Queues

1. Sort the stores by increasing $h_i$.
2. Maintain a priority queue of $F$ keyed by $t_i$, largest $t_i$ on top.
3. Add store $i$ to $F$.

Time Complexity: $O(n \log n)$. 

Statistics: 72 submissions, 0 accepted.
Simpler $O(n \log n)$ Solution using Priority Queues

1. Sort the stores by increasing $h_i$.
2. Maintain a priority queue of $F$ keyed by $t_i$, largest $t_i$ on top.
3. Add store $i$ to $F$.
4. Pop from the priority queue until $\sum_{j \in F} t_j \leq h_i$.
**F - Atlantis**

**Simpler $O(n \log n)$ Solution using Priority Queues**

1. Sort the stores by increasing $h_i$.
2. Maintain a priority queue of $F$ keyed by $t_i$, largest $t_i$ on top.
3. Add store $i$ to $F$.
4. Pop from the priority queue until $\sum_{j \in F} t_j \leq h_i$.

Without loss of generality assume each $t_i$ is distinct. Let $add(i)$ be a true or false value denoting whether store $i$ is added in this strategy. Then

$$add(i) = \sum_{j \text{ s.t. } t_j < t_i \text{ and } add(j)} t_j + t_i \leq h_i.$$
F - Atlantis

Simpler $O(n \log n)$ Solution using Priority Queues

1. Sort the stores by increasing $h_i$.
2. Maintain a priority queue of $F$ keyed by $t_i$, largest $t_i$ on top.
3. Add store $i$ to $F$.
4. Pop from the priority queue until $\sum_{j \in F} t_j \leq h_i$.

Without loss of generality assume each $t_i$ is distinct. Let $add(i)$ be a true or false value denoting whether store $i$ is added in this strategy. Then

$$
add(i) = \sum_{\text{such that } t_j < t_i \text{ and } add(j)} t_j + t_i \leq h_i.
$$

This is the same condition as the originally proposed greedy algorithm.
Simpler $O(n \log n)$ Solution using Priority Queues

1. Sort the stores by increasing $h_i$.
2. Maintain a priority queue of $F$ keyed by $t_i$, largest $t_i$ on top.
3. Add store $i$ to $F$.
4. Pop from the priority queue until $\sum_{j \in F} t_j \leq h_i$.

Without loss of generality assume each $t_i$ is distinct. Let $add(i)$ be a true or false value denoting whether store $i$ is added in this strategy. Then

$$add(i) = \sum_{j \text{ s.t. } t_j < t_i \text{ and } add(j)} t_j + t_i \leq h_i.$$ 

This is the same condition as the originally proposed greedy algorithm.

Time Complexity: $O(n \log n)$. 

Statistics: 72 submissions, 0 accepted.
Simpler $O(n \log n)$ Solution using Priority Queues

1. Sort the stores by increasing $h_i$.
2. Maintain a priority queue of $F$ keyed by $t_i$, largest $t_i$ on top.
3. Add store $i$ to $F$.
4. Pop from the priority queue until $\sum_{j \in F} t_j \leq h_i$.

Without loss of generality assume each $t_i$ is distinct. Let $add(i)$ be a true or false value denoting whether store $i$ is added in this strategy. Then

$$add(i) = \sum_{j \text{ s.t. } t_j < t_i \text{ and } add(j)} t_j + t_i \leq h_i.$$ 

This is the same condition as the originally proposed greedy algorithm.

Time Complexity: $O(n \log n)$.

Statistics: 72 submissions, 0 accepted.
Problem

Given an array $A$ of $N$ integers, determine the minimum number of changes in $A$ to make every contiguous subarray of length $K$ sum to $S$. 
D - Smooth Array

Solution

As the problem states, for the array to be $K_S$-smooth, it must contain a repeating pattern of length $K$. 

Can use dynamic programming to find the pattern that requires the minimum number of changes in $A$.

Let $DP(i, j) :=$ minimum number of changes to make the first $i$ integers of the pattern sum to $j$.

And let $cost(i, v) :=$ number of changes in $A$ to make $A_i, A_i+K, A_i+2K, \ldots$ equal to $v$.

A simple recurrence is then

$$DP(i, j) = \min_v DP(i-1, j-v) + cost(i, v)$$

There are $O(KS)$ states and each takes $O(S)$ time to evaluate, so the complexity is $O(KS^2)$. 

# of iterations: 5000

$3 = 125 \times 10^9 \Rightarrow$ TLE!

Problem Author: Bowen Yu
As the problem states, for the array to be $K_S$-smooth, it must contain a repeating pattern of length $K$.

Can use dynamic programming to find the pattern that requires the minimum number of changes in $A$. 

Let $\text{DP}(i, j) := \text{minimum number of changes to make the first } i \text{ integers of the pattern sum to } j$.

And let $\text{cost}(i, v) := \text{number of changes in } A \text{ to make } A_i, A_{i+K}, A_{i+2K}, \ldots \text{ equal to } v$.

A simple recurrence is then $\text{DP}(i, j) = \min_v \text{DP}(i-1, j-v) + \text{cost}(i, v)$.

There are $O(KS)$ states and each takes $O(S)$ time to evaluate, so the complexity is $O(KS^2)$.

# of iterations: 5000

$3 = 125 \times 10^9 \Rightarrow TLE!$
D - Smooth Array

Solution

- As the problem states, for the array to be $K_S$-smooth, it must contain a repeating pattern of length $K$.
- Can use dynamic programming to find the pattern that requires the minimum number of changes in $A$.
- Let $DP(i, j) := \text{minimum number of changes to make the first } i \text{ integers of the pattern sum to } j$. 
As the problem states, for the array to be $K_S$-smooth, it must contain a repeating pattern of length $K$.

Can use dynamic programming to find the pattern that requires the minimum number of changes in $A$.

Let $DP(i, j) :=$ minimum number of changes to make the first $i$ integers of the pattern sum to $j$.

And let $cost(i, v) :=$ number of changes in $A$ to make $A_i, A_{i+K}, A_{i+2K}, \ldots$ equal to $v$. 
As the problem states, for the array to be $K_S$-smooth, it must contain a repeating pattern of length $K$.

Can use dynamic programming to find the pattern that requires the minimum number of changes in $A$.

Let $DP(i, j) := \text{minimum number of changes to make the first } i \text{ integers of the pattern sum to } j$.

And let $\text{cost}(i, v) := \text{number of changes in } A \text{ to make } A_i, A_{i+K}, A_{i+2K}, \ldots \text{ equal to } v$.

A simple recurrence is then

$$DP(i, j) = \min_v DP(i - 1, j - v) + \text{cost}(i, v)$$
D - Smooth Array

Solution

- As the problem states, for the array to be $K_S$-smooth, it must contain a repeating pattern of length $K$.
- Can use dynamic programming to find the pattern that requires the minimum number of changes in $A$.
- Let $DP(i, j) :=$ minimum number of changes to make the first $i$ integers of the pattern sum to $j$.
- And let $\text{cost}(i, v) :=$ number of changes in $A$ to make $A_i, A_{i+K}, A_{i+2K}, \ldots$ equal to $v$.
- A simple recurrence is then $DP(i, j) = \min_v DP(i - 1, j - v) + \text{cost}(i, v)$
- There are $O(KS)$ states and each takes $O(S)$ time to evaluate, so the complexity is $O(KS^2)$. 

Problem Author: Bowen Yu
As the problem states, for the array to be $K_S$-smooth, it must contain a repeating pattern of length $K$.

Can use dynamic programming to find the pattern that requires the minimum number of changes in $A$.

Let $DP(i, j) :=$ minimum number of changes to make the first $i$ integers of the pattern sum to $j$.

And let $cost(i, v) :=$ number of changes in $A$ to make $A_i, A_{i+K}, A_{i+2K}, \ldots$ equal to $v$.

A simple recurrence is then

$$DP(i, j) = \min_v DP(i - 1, j - v) + cost(i, v)$$

There are $O(KS)$ states and each takes $O(S)$ time to evaluate, so the complexity is $O(KS^2)$. # of iterations: $5000^3 = 125 \times 10^9 \Rightarrow$ TLE!
D - Smooth Array

An $O(NS)$ Solution

- Observation: there are at most $\lceil N/K \rceil$ unique values in \( \{A_i, A_{i+K}, A_{i+2K}, \ldots \} \).
D - Smooth Array

An $O(\text{NS})$ Solution

- Observation: there are at most $\lceil N/K \rceil$ unique values in $\{A_i, A_{i+K}, A_{i+2K}, \ldots\}$.
- All other values of $v$ require changing all of $A_i, A_{i+K}, A_{i+2K}, \ldots$, so the cost function for these values will be $\lceil (N - i + 1)/K \rceil$. 

Time complexity: $O(\text{KS} \cdot \lceil N/K \rceil) = O(\text{NS})$. 

Statistics: 37 submissions, 0 accepted.

Problem Author: Bowen Yu
Observation: there are at most \(\lceil N/K \rceil\) unique values in 
\(\{A_i, A_{i+K}, A_{i+2K}, \ldots\}\).

All other values of \(v\) require changing all of \(A_i, A_{i+K}, A_{i+2K}, \ldots\), so the cost function for these values will be \(\lceil (N - i + 1)/K \rceil\).

Instead of iterating all \(v\) in this second category, we can precompute the best \(v\) to minimize \(DP(i-1, j-v)\).
An $O(NS)$ Solution

- Observation: there are at most $\lceil N/K \rceil$ unique values in
$$\{A_i, A_{i+K}, A_{i+2K}, \ldots\}.$$ 

- All other values of $v$ require changing all of $A_i, A_{i+K}, A_{i+2K}, \ldots$, so the cost function for these values will be $\lceil (N - i + 1)/K \rceil$.

- Instead of iterating all $v$ in this second category, we can precompute the best $v$ to minimize $DP(i - 1, j - v)$.

- We can still try all $\lceil N/K \rceil$ values of $v$ in the first category.
An $O(NS)$ Solution

- Observation: there are at most $\lceil N/K \rceil$ unique values in $
\{A_i, A_{i+K}, A_{i+2K}, \ldots\}$.
- All other values of $v$ require changing all of $A_i, A_{i+K}, A_{i+2K}, \ldots$, so
the cost function for these values will be $\lceil (N - i + 1)/K \rceil$.
- Instead of iterating all $v$ in this second category, we can precompute
the best $v$ to minimize $DP(i - 1, j - v)$.
- We can still try all $\lceil N/K \rceil$ values of $v$ in the first category.
- The recurrence now takes $O(N/K)$ time.
An $O(NS)$ Solution

- Observation: there are at most $\lceil N/K \rceil$ unique values in
  $\{A_i, A_{i+K}, A_{i+2K}, \ldots\}$.
- All other values of $v$ require changing all of $A_i, A_{i+K}, A_{i+2K}, \ldots$, so
  the cost function for these values will be $\lceil (N - i + 1)/K \rceil$.
- Instead of iterating all $v$ in this second category, we can precompute
  the best $v$ to minimize $DP(i-1, j-v)$.
- We can still try all $\lceil N/K \rceil$ values of $v$ in the first category.
- The recurrence now takes $O(N/K)$ time.
- Time complexity: $O(KS \cdot N/K) = O(NS)$. 
**D - Smooth Array**

**An \(O(NS)\) Solution**

- Observation: there are at most \(\lceil N/K \rceil\) unique values in \(\{A_i, A_{i+K}, A_{i+2K}, \ldots\}\).
- All other values of \(v\) require changing all of \(A_i, A_{i+K}, A_{i+2K}, \ldots\), so the cost function for these values will be \(\lceil (N - i + 1)/K \rceil\).
- Instead of iterating all \(v\) in this second category, we can precompute the best \(v\) to minimize \(DP(i - 1, j - v)\).
- We can still try all \(\lceil N/K \rceil\) values of \(v\) in the first category.
- The recurrence now takes \(O(N/K)\) time.
- Time complexity: \(O(KS \cdot N/K) = O(NS)\).

Statistics: 37 submissions, 0 accepted.
Questions? Comments? Concerns? Email Bryce Sandlund: bcsandlund@uwaterloo.ca.