May 01, 2024

New Zealand Informatics Competition 2024

Round One

Editorial by Zalan V, Anatol C, Phoebe Z, Jonathon S

Contents

Introduction

The solutions to this round of NZIC problems are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

Resources

Ever wondered what the error messages mean?

https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Ever wondered why your submission scored zero?

https://www.nzoi.org.nz/nzic/resources/why-did-i-score-zero.pdf

See our list of other useful resources here:

https://www.nzoi.org.nz/nzic/resources

Cardboard Boxes

Problem authored by Iván G
https://train.nzoi.org.nz/problems/1371

Full Solution

The observation required to solve this problem is that we can stack as many unique boxes on top of each other as we want, we would just need to sort them. Thus, to solve this problem we need to find the number of unique boxes. Since each box has a width of at most $10,000$ we can use an array, or alternatively a hash table (known as a set in python) to find the number of unique box widths.

Python Solution

```
N = int(input())
boxes = []
for i in range(N):
    boxes.append(input())
unique_boxes = set(boxes)
print(len(unique_boxes))
```

Betty The Cat 2

Problem authored by Phoebe Z
https://train.nzoi.org.nz/problems/1391

This solution makes heavy use of the floor and ceiling operators. $\lfloor x \rfloor$ represents x rounded down to the nearest integer. $\lceil x \rceil$ represents x rounded up to the nearest integer.

Subtask 1

In this subtask, we can always meet the quota exactly. The naive way of doing so is to just get K boxes, each containing one cookie. However, we can use less boxes by taking as many boxes of B cookies as possible.

Specifically, the optimal solution is to get $\lfloor \frac{K}{B} \rfloor$ boxes of B cookies and then add as many boxes of 1 cookie as needed.

Python Solution

```
A, B, K = map(int, input().split())
b_boxes = K // B
a_boxes = K - B * b_boxes
print(0, a_boxes + b_boxes)
```

Subtask 2

In this subtask, since there is really only one kind of cookie box we can get, we don't have that many options for how many boxes to buy. The optimal solution is to buy either $\lfloor \frac{K}{B} \rfloor$ or $\lceil \frac{K}{B} \rceil$, boxes. as anything higher/lower we can simply remove/add a cookie box to achieve a strictly better solution. Therefore we can check both $\lfloor \frac{K}{B} \rfloor$ and $\lceil \frac{K}{B} \rceil$ and take the solution with a smaller difference to the target K – if they have the same difference, we take $\lfloor \frac{K}{B} \rfloor$ as it requires a smaller number of cookie boxes.

Python Solution

```
A, B, K = map(int, input().split())
take_floor = (K % B, K // B)
take_ceil = (B - (K % B), K // B + 1)
if take_floor[0] <= take_ceil[0]:
    print(take_floor[0], take_floor[1])
else:
    print(take_ceil[0], take_ceil[1])
```

Subtask 3

Note that in an optimal solution, we will never need more than K boxes of A cookies. Buying K boxes will put us exactly at or above the quota so buying more than K boxes will always place us further from the quota than buying exactly K.

Since K is small in this subtask, we can iterate some variable (call it i) from 0 to K and find the best solution possible when buying i boxes of A cookies, and then keep track of the best overall solution.

If we are buying i boxes of A cookies then there are two distinct scenarios. Either $A \cdot i \geq K$ and we are already above the quote. In this case buying any amount of boxes of B cookies would make the solution strictly worse, so we should get none. In the other case, we can use the same trick as in the last subtask to find the best amount of boxes of B cookies to buy. The optimal solution is to buy either $\lfloor \frac{K-A \cdot I}{B} \rfloor$ or $\lceil \frac{K-A \cdot I}{B} \rceil$ boxes of B cookies, whichever gives the better result.

Python Solution

The code for this solution can be found at the end of this document .

Full Solution

Without loss of generality, assume that $A \leq B$. (If that is not the case, simply swap A and B). Note that if we have a solution with B boxes of A cookies, then we can replace that with A boxes of B cookies. Note that in this new solution Betty still eats the same amount of cookies (because $B \cdot A = A \cdot B$). Additionaly, if $A \neq B$, this decreases the total number of boxes (otherwise the number of boxes stays the same). So, this new solution is as good or better than the old solution. All of this means that an optimal solution where we buy less than B boxes of A cookies.

Now, we can use nearly the exact same solution as the previous subtask except that we must swap A and B if $A > B$ and we only need to iterate the value of i up to B.

Python Solution

The code for this solution can be found at the end of this document .

C++ Solution

The code for this solution can be found at the end of this document .

Extra fun

Try the leveled up version https://train.nzoi.org.nz/problems/1414 (does not require linear diophantine equation) (requires more than one line)

Supply Scheduling

Problem authored by Iván G

https://train.nzoi.org.nz/problems/1377

Subtask 1

In this subtask there are only two possible answers. The first scenario is when $A[1] - A[0] = B[1] - B[0]$ in which case if the two zoos receive their first delivery on the same day, the second deliveries will also arrive on the same day and so the answer is 2. Otherwise, we can still arrange for the first deliveries to arrive on the same day, but then the next deliveries will be on separate days and so food must be delivered on 3 distinct days.

Python Solution

```
N = int(input())
a_days = list(map(int, input().split()))
b_days = list(map(int, input().split()))

if a_days[1] - a_days[0] == b_days[1] - b_days[0]:
  print(2)
else:
  print(3)
```

Subtask 2

In this subtask, note that opening the Basilisk Biosphere more than 2000 days after the Aardvark Asylum will always result in no delivery days lining up. Similarly, opening the Biosphere more than 2000 days before the Asylum will have the same result. This means the number of days that we must offset the opening of the Biosphere (relative) to the ayslum is between −2000 and 2000, giving us only 4001 possible values. We note that a negative offset is possible because we can delay the opening days of both zoos by as much as needed so the first delivery occurs after the opening day.

We can simply iterate over each of these offsets, and for each of them, count the number of days that line up. Since both arrays are sorted, we can iterate through both arays together to find the number of days that line up with a certain offset in $O(N)$.

If no days line up then food must be delivered on $2 \cdot N$ different days. For each pair of days that line up, this removes one delivery day so the answer is simply $2 \cdot N$ minus the maximum number of days we can make line up.

```
N = int(input())
a_days = list(map(int, input().split()))
b_days = list(map(int, input().split()))

max_matching = 0
for offset in range(-2000, 2001):
    a_index = 0
    b_index = 0
    matched = 0
    while a_index < N and b_index < N:
        if b_days[b_index] < a_days[a_index] + offset:
            b_index += 1
        elif b_days[b_index] == a_days[a_index] + offset:
            a_index += 1
            b_index += 1
            matched += 1
```

```
        else:
            a_index += 1

    max_matching = max(max_matching, matched)

print(2 * N - max_matching)
```

Subtask 3

Note that in any testcase we can always make any two delivery days line up. If the asylum opens on some day a_x and the biosphere opens on some day b_y then by opening the biosphere $a_x - b_y$ days after the asylum, those two deliveries will be on the same day.

Importantly, this means the only offsets that are worth trying are those of the form $a_x - b_y$ since any other offsets would result in zero days lining up.

Remember from subtask two that we can check each offset in $O(N)$. Here, the number of offsets to check is just $O(N^2)$ (since it is every pair-wise difference between the two arrays and the number of pairs is N^2). This means that we can now find the maximum number of lined up days in $O(N^3)$ giving us an $O(N^3)$ solution which passes this subtask since $N \leq 100$.

```
N = int(input())
a_days = list(map(int, input().split()))
b_days = list(map(int, input().split()))

differences = []
for a_day in a_days:
    for b_day in b_days:
        differences.append(b_day - a_day)

max_matching = 0
for offset in differences:
    a_index = 0
    b_index = 0
    matched = 0
    while a_index < N and b_index < N:
        if b_days[b_index] < a_days[a_index] + offset:
            b_index += 1
        elif b_days[b_index] == a_days[a_index] + offset:
            a_index += 1
            b_index += 1
            matched += 1
        else:
            a_index += 1

    max_matching = max(max_matching, matched)

print(2 * N - max_matching)
```

Full

We can reinforce the argument from the previous subtask to discover a better solution. Two opening days a_x and b_y will line up if and only if the biosphere opens $a_x - b_y$ days after the asylum. This means that for some offset n the number of delivery days that line up is the number of pairs where $a_x - b_y = n$.

Since $N \leq 1000$ we can calculate every possible value of $a_x - b_y$ and count up how much each value occurs. Whichever value occurs the most will be the optimal offset and furthermore, the number of

times it occurs is the number of days that will line up if we use this offset, which gives us our final answer.

The difference of all these pairs can be calculated and counted up in $O(N^2)$ using a dictionary in python (or an unordered map or hash map in other languages) giving us an $O(N^2)$ solution.

Python Solution

```
import collections

N = int(input())
a_days = list(map(int, input().split()))
b_days = list(map(int, input().split()))
difference_counts = collections.defaultdict(int)

for day_a in a_days:
    for day_b in b_days:
        difference_counts[day_b - day_a] += 1

print(2 * N - max(difference_counts.values()))
```

C++ Solution

```
#include <bits/stdc++.h>

using namespace std;

int N, A[1000], B[1000];

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) cin >> A[i];
  for (int i = 0; i < N; i++) cin >> B[i];

  unordered_map<int, int> counts;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      counts[A[i] - B[j]]++;
    }
  }

  int res = 0;
  for (auto p: counts) res = max(res, p.second);

  cout << 2 * N - res << endl;
}
```

Pinecones

Problem authored by Jonathon S
https://train.nzoi.org.nz/problems/1375

Subtask 1

The maximum possible total height of throws for a query can be found using a greedy algorithm. For each throw, simply throw the pinecone that gives you the maximum throw height at that point. Since the constraints are small enough, we can simulate the throws one by one and find the maximum height for each throw by using a `for` loop.

Note that we don't need modulo here, since the constraints of this subtask guarantee that the answer is less than $1,000,000,007$.

Time complexity: $O(Q \cdot T \cdot N)$

Subtask 2

In this subtask, there is only one query and the pinecone throwing heights and total number of throws is less than 10^5. We can sort the pinecones, and repeatedly throw the pinecones of maximum throw height. Alternatively, we can use a `priority_queue` in C++ (or the `heapq` module in Python) to find the maximum height efficiently without having to sort the pinecones.

Note that in this subtask we need to use modulo, as the answer can be larger than $1,000,000,007$.

Time complexity: $O(Q \cdot T + N \log N)$

Python Solution

```
MOD = 1000000007

N, Q = map(int, input().split())
heights = sorted(map(int, input().split()))
T = int(input())

total = 0
index = N - 1
current_height = heights[-1]
while T > 0 and heights[0] > 0:
    if heights[index] == current_height:
        total = (total + current_height) % MOD
        heights[index] -= 1
        T -= 1
        if index > 0:
            index -= 1
        else:
            index = N - 1
            current_height = heights[-1]
    else:
        index = N - 1
        current_height = heights[-1]

print(total)
```

Subtask 3

We can no longer simulate each throw in this subtask. However, since $\max(H_i) \leq 10^5$, we can create three lists, A, B and num, which are defined as follows for all $1 \leq i \leq \max(H_i)$:

- A_i = the number of throws with height $\geq i$
- B_i = the sum of heights of throws with height $\geq i$
- num_i = the number of pinecones with initial height $\geq i$

We can do this in $O(\max(H_i))$ time by looping from $\max(H_i)$ down to 1, and performing $A_i = A_{i+1} + \text{num}_i$ and $B_i = B_{i+1} + \text{num}_i \cdot i$. Note that num can be calculated easily in $O(\max(H_i))$ time, and A, B and num are also known as prefix sums.

You can think of each pinecone i as a column of H_i cells, where each cell represents a throw, and the height of a cell represents the height of the throw. Then, sort all columns in non-increasing order of height. Each height will have a row of cells. In fact, for all $1 \leq h \leq \max(H_i)$, height h will have exactly num_h cells. The answer for the j-th scenario is the sum of heights of the highest T_j cells.

For each scenario j, we can now binary search for the minimum height x such that $A_x \leq T_j$ in $O(\log(\max(H_i)))$ time. The answer is then $B_x + \left(T_j - A_x\right) \cdot (x-1)$. To think about it intuitively, we have used A_x throws to get a combined height of B_x, so $T_j - A_x$ is the number of throws we have remaining. Since we have used all throws with height $\geq x$, the best we can do with our remaining throws is to throw them all to a height of $x-1$, hence adding $\left(T_j - A_x\right) \cdot (x-1)$ to our answer.

If you're wondering, it's guaranteed that there are at least $T_j - A_x$ throws with height $x-1$. Assume that $T_j - A_x >$ the number of throws with height $x-1$. Then, $A_{x-1} < T_j$ would be true, so $x-1$ (or less) would actually be the result of the binary search, thus we have reached a contradiction.

Time complexity: $O(N + \max(H_i) + Q \cdot \log(\max(H_i)))$

Python Solution

The code for this solution can be found at the end of this document.

Full

Solution 1

For full points, H_i can be up to 10^9, so we can no longer store the prefix sums for each different height. Instead, we can use coordinate compression to store values in the prefix sum only for the initial heights given to us. We will first add a "dummy" pinecone with height 0 into H, then sort H in non-decreasing order. We'll also slightly modify our three lists to be as follows:

- A_i = the number of throws with height $> H_i$
- B_i = the sum of heights of throws with height $> H_i$
- num_i = the number of pinecones with initial height $> H_i$

We can now calculate them for all $0 \leq i \leq N$. Note that the 0-height pinecone has index 0, and the actual pinecones start at index 1. We now have:

- $\text{num}_i = N - i$
- $A_i = A_{i+1} + \text{num}_i \cdot \left(H_{i+1} - H_i\right)$
- $B_i = B_{i+1} + \text{num}_i \cdot \left(\frac{H_{i+1}(H_{i+1}+1)}{2} - \frac{H_i(H_i+1)}{2}\right)$

The expression inside brackets in the formula for B_i calculates the sum of $H_i + 1, H_i + 2, ..., H_{i+1}$. You can also use the arithmetic sum formula to achieve a simpler expression.

Going back to the cell analogy in Subtask 3, we will have multiple rows with num_x cells each after our binary search. Let `full_rows` be the number of full rows we can use, and `extra_throws` be the number of remaining throws we have after we use the full rows. We will use the same definition as Subtask 3 for x.

- `full_rows` = $\left\lfloor \frac{T_j - A_x}{\text{num}_x} \right\rfloor$

- `extra_throws` = $\left(T_j - A_x\right) - \text{num}_x \cdot \text{full\_rows}$

The answer is then $B_x + \text{num}_x \cdot \left(\frac{H_x(H_x+1)}{2} - \frac{(H_x-\text{full\_rows})(H_x-\text{full\_rows}+1)}{2}\right) + \text{extra\_throws} \cdot (H_x - \text{full\_rows})$. The arithmetic sum formula can again be used for simplification.

Time complexity: $O((N+Q)\log N)$

Python Solution

The code for this solution can be found at the end of this document.

Solution 2 (Briefly)

You may have noticed that the order that we process the scenarios does not matter, as long as we output the answers in the correct order. It turns out that we can solve the problem efficiently if we process the scenarios in non-decreasing order of number of throws, and keep track of the sum of throw heights. This type of solution is called an *offline* solution.

Using the row/cell analogy in Subtask 3, the idea is to start from the highest row, then keep moving downwards until we run out of throws. We then add the heights of all the rows we reached to the sum, which will be stored as the answer for the current scenario. Then, move to the next scenario, but start from the row we just finished at. We will repeat this until all scenarios have been processed.

The amount of maths required is similar to Solution 1. However, this solution may be easier to implement.

Time complexity: $O((N+Q)\log N)$

C++ Solution

The code for this solution can be found at the end of this document.

The Grand Tree

Problem authored by Iván G
https://train.nzoi.org.nz/problems/1330

Subtask 1

In this subtask, the tastiness we are trying to achieve (M) is either 1 or 2.

Note that we must always collect apple 0 which will have a non-zero tastiness (t_0). This means the minimum non-zero tastiness we can ever achieve is the t_0.

If $t_0 \geq M$ then just by picking the first apple we solve the problem and so the answer is t_0.

In this subtask, the only other case possible is when $M = 2$ and $t_0 = 1$, in this case we just need to get one more tastiness after picking apple 0. The best option will always be to pick the apple connected to apple 0 with the minimum tastiness and so the answer is just $t_0 +$ this minimum tastiness. (Note that this minimum is guaranteed to exist).

Python Solution

```
N, M  = map(int, input().split())
t = list(map(int, input().split()))

if t[0] >= M:
  print(t[0])
else:
  minimum_connected_apple = 2001
  for i in range(N-1):
    x, y = map(int, input().split())
    if x == 0: minimum_connected_apple = min(minimum_connected_apple, t[y])
    if y == 0: minimum_connected_apple = min(minimum_connected_apple, t[x])
  print(t[0] + minimum_connected_apple)
```

Subtask 2

There are many solutions possible for this subtask. Here we will present two.

Brute Force Solution

The fact that N is so small allows us to use very inneficient approaches. Note that the problem can be boiled down to finding a subset of the available apples with minimum total tastiness K such that this $K \geq M$ and such that those apples in the subset can be picked without knocking off any other apples.

There are 2^N possible subsets of apples (actually 2^{N-1} since you must always pick apple 0) so we can actually iterate over all possible subsets to find the answer, we just need to be able to check that some subset is valid.

There are many ways of checking if a subset is valid (and efficiency doesn't matter too much) but the simplest is as follows: A subset is valid if and only if, for each apple in the subset except for apple 0, its parent[1] is also in the subset. This works because for every apple i in our subset that we want to pick:

- If its parent is not in the subset, then by picking apple i we will knock its parent apple of its branch.
- If its parent is in the subset, it means we can get to that apple without knocking any apples off and so we can reach apple i by following the branch from its parent to itself.

[1] If you are not familiar with graph theory, you can think of the parent of apple i as the apple that must be picked before picking apple i.

With this, we can solve the full problem in $O(2^N \cdot N)$

Python Solution

Note that this solution is naively implementing the above solution and is actuall $O(2^N \cdot N^2)$. As an extra challenge, can you figure out why, and make it $O(2^N \cdot N)$.

The code for this solution can be found at the end of this document.

Recursive Solution (Briefly)

For any apple i we can calculate the set T_i of all the possible total tastiness values that can be achieved by picking apple i, along with any other apple in apple i's subtree as follows:

- Initialize T_i as the set containing only t_i, the tastiness of apple i.
- For every child c of apple i:
  - Recursively calculate T_c.
  - Add $a + b$ to T_i for all a in T_c and all b already in T_i.

Using this, we can calculate T_0. Then, the answer is simply the smallest value in T_0 greater than or equal to M.

This time complexity of this solution is difficult to estimate but it is roughly $O\left(N^3 \cdot \max(t)^2\right)$ although in reality it is much faster for small N.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 3

In this subtask, the entire tree forms a line. This means that at any point, there is at most one apple that we are able to pick. We must pick apple 0 first, and then we can pick apple 1, and then we can pick apple 2, and so on... .This leads to a very simple greedy solution, we simply pick the only apple available until the total tastiness reaches M.

Python Solution

```
N, M  = map(int, input().split())
t = list(map(int, input().split()))

res = 0
for i in range(N):
  res += t[i]
  if res >= M:
    break

print(res)
```

Full Solution

You may notice that this problem seems very similar to the classic knapsack problem. In fact it is! Consider the following similar knapsack problem:

- Given an integer M and an array t of N integers, find the smallest value x larger than or equal to M such that x is the sum of some subset of t.

This can be solved using dynamic programming. Let's define the function dp such that $\text{dp}(i, m)$ is the smallest integer y larger than or equal to m such that y is the sum of some subset of $t[i...]$[2].

[2] $t[i...]$ denotes the suffix of array t that excludes the first i elements of t. Can be denoted by `t[i:]` in python.

By definition, the solution to the knapsack problem is $\mathrm{dp}(0, M)$.

This function has a few trivial base cases.

- $\mathrm{dp}(i, m) = 0$ if $m \leq 0$ since 0 can always be formed by adding up zero elements.
- $\mathrm{dp}(i, m) = \infty$ if $m > 0$ and $i \geq N$ since the array `t[i:]` is empty and so we cannot form positive elements.

For all other cases, we can define the function recursively. Specifically consider the case where we want to calculate $\mathrm{dp}(i, m)$. The value of this function is the sum of some subset of $t[i...]$. Element i is either in this subset or it isn't.

- If element i is in the subset, then the result is $t_i + \mathrm{dp}(i + 1, m - t_i)$
- If element i is not in the subset, we skip this element and so the result is $\mathrm{dp}(i + 1, m)$

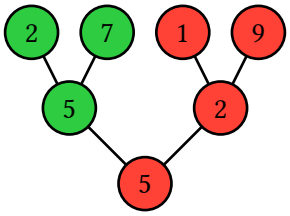
Since we want to find the smallest possible valid subset sum, we can define $\mathrm{dp}(i, m)$ as the minimum of the two values above. So,

$$\mathrm{dp}(i, m) = \begin{cases} 0 & \text{if } m \leq 0 \\ \infty & \text{if } i \geq N \\ \min(t_i + \mathrm{dp}(i + 1, m - t_i), \mathrm{dp}(i + 1, m)) & \text{otherwise} \end{cases}$$

This allows us to solve the knapsack problem by simply evaluating $\mathrm{dp}(0, M)$. Note that this would be incredibly inefficient as each call to dp makes two more calls to dp resulting in a very large amount of calls. We can make this much more efficient by caching (also called memoizing) the results of identical calls to dp. This turns it into an exponential solution into an $O(N \cdot M)$ solution.

The original problem is incredibly similar to this knapsack problem. The only difference is that if we don't pick some apple i it locks us out of picking any descendants of apple i. How can we handle this? The trick is to first flatten our tree into an array in a very specific way.

Consider the following tree:



If we flatten it using a pre-order traversal[3], we get the following array:

Notice what happened to the green subtree in our flattened tree – it became a contiguous subarry with the root element at the front. In fact, this will always be true for any subtree in the original tree (no matter the original tree) – this is a property of a pre-order traversal.

This is hugely advantageous. Remember when we were defining the "dp" function? We said that for $\mathrm{dp}(i, m)$ we can either take element i or not take it. The trick is that before, when we skipped element

[3]In a pre-order traversal, the root is visited first, followed by the child nodes. A normal depth-first search (starting at the root) is a pre-order traversal.

i, we continued from element $i+1$. In the actual problem, we want to skip all apples in element i's subtree. Now, this becomes trivial since all these elements are contiguous and occur after element i.

This means if we transform our tree into an array like above we can use this modified dp formula to solve the problem:

$$\mathrm{dp}(i,m) = \begin{cases} 0 & \text{if } m \leq 0 \\ \infty & \text{if } i \geq N \\ \min(t_i + \mathrm{dp}(i+1, m-t_i), \mathrm{dp}(\mathrm{next}_i, m)) & \text{otherwise} \end{cases}$$

Where next_i corresponds to the smallest index greater than i than is not in element i's subtree. These can be calculated while performing the pre-order traversal.

Computing one value of the dp function is $O(1)$ and there are $O(NM)$ possible sets of arguments making this a $O(NM)$ solution.

Python Solution

The code for this solution can be found at the end of this document.

C++ Solution

This solution does the "flattening" implicitly.

The code for this solution can be found at the end of this document.

Additional Example Solutions

Betty The Cat 2 Subtask 3 Python Solution

```
A, B, K = map(int, input().split())

best_diff = K
best_boxes = 0

for i in range(K+1):
  if A * i >= K:
    diff = A * i - K
    if diff < best_diff or diff == best_diff and i < best_boxes:
      best_diff = diff
      best_boxes = i
  else:
    remaining = K - A * i
    b_low = remaining // B
    b_high = remaining // B + 1

    if remaining - b_low * B <= b_high * B - remaining:
      best_b = b_low
    else:
      best_b = b_high

    diff = abs(remaining - best_b * B)
    if diff < best_diff or diff == best_diff and i + best_b < best_boxes:
      best_diff = diff
      best_boxes = i + best_b

print(best_diff, best_boxes)
```

Betty The Cat 2 Full Python Solution

```
A, B, K = map(int, input().split())

best_diff = K
best_boxes = 0

for i in range(K+1):
  if A * i >= K:
    diff = A * i - K
    if diff < best_diff or diff == best_diff and i < best_boxes:
      best_diff = diff
      best_boxes = i
  else:
    remaining = K - A * i
    b_low = remaining // B
    b_high = remaining // B + 1

    if remaining - b_low * B <= b_high * B - remaining:
      best_b = b_low
    else:
      best_b = b_high

    diff = abs(remaining - best_b * B)
    if diff < best_diff or diff == best_diff and i + best_b < best_boxes:
      best_diff = diff
```

```
      best_boxes = i + best_b

print(best_diff, best_boxes)
```

Betty The Cat 2 Full C++ Solution

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  long long A, B, K;
  cin >> A >> B >> K;

  if (A > B) swap(A, B);
  pair<long long, long long> best = {K, 0};

  for (int a = 0; a < B; a++) {
    long long lowb = max(0ll, (K - a * A) / B);
    long long highb = lowb + 1;

    best = min(best, {abs(K - a * A - lowb * B), a + lowb});
    best = min(best, {abs(K - a * A - highb * B), a + highb});
  }

  cout << best.first << " " << best.second << endl;
}
```

Pinecones Subtask 3 Online Python Solution

```
N, Q = map(int, input().split())
H = list(map(int, input().split()))

heights = [0] * 100001

for h in H:
    heights[h] += 1

A = [0] * 100002
B = [0] * 100002

num = 0
for i in range(100000, -1, -1):
    num += heights[i]
    A[i] = A[i+1] + num
    B[i] = B[i+1] + num * i

for i in range(Q):
    t = int(input())

    lo_x = 0
    hi_x = 100001

    while lo_x != hi_x:
        m = (lo_x + hi_x) // 2

        if A[m] <= t:
            hi_x = m
```

```
        else:
            lo_x = m+1

    x = lo_x

    if x == 0:
        ans = B[x]
    else:
        ans = B[x] + (t - A[x]) * (x - 1)

    print(ans % 1000000007)
```

Pinecones Full Python Solution

```
N, Q = map(int, input().split())
H = list(map(int, input().split()))
H.append(0)
H.sort()

A = [0] * (len(H) + 1)
B = [0] * (len(H) + 1)
num = [0] * (len(H) + 1)

for i in range(len(H) - 1, 0, -1):
    num[i] = num[i+1] + 1
    A[i] = A[i+1] + num[i] * (H[i] - H[i-1])
    B[i] = B[i+1] + num[i] * (H[i] * (H[i] + 1) - H[i-1] * (H[i-1] + 1)) // 2

A[0] = A[1]
B[0] = B[1]

for i in range(Q):
    t = int(input())

    if t > A[0]:
        print(B[0] % 1000000007)
    else:
        lo_x = 0
        hi_x = len(A) - 1

        while lo_x != hi_x:
            m = (lo_x + hi_x) // 2

            if A[m] <= t:
                hi_x = m
            else:
                lo_x = m+1

        x = lo_x
        remaining = t - A[x]
        num_pinecones = num[x] + 1
        base_height = H[x-1]

        full_lines = remaining // num_pinecones
      full_lines_extra = num_pinecones * (base_height * (base_height + 1) - (base_height
- full_lines) * (base_height - full_lines + 1)) // 2
```

```
        rem = remaining % num_pinecones

        ans = B[x] + full_lines_extra + rem * (base_height - full_lines)
        print(ans % 1000000007)
```

Pinecones Full Offline C++ Solution

```
#include <bits/stdc++.h>

using namespace std;

int N, Q;
long long H[100001], T[100000], TSorted[100000];
const int MOD = 1000000007;

int main() {
    cin >> N >> Q;
    for (int i = 1; i <= N; i++) cin >> H[i];
    for (int i = 0; i < Q; i++) {
        cin >> T[i];
        TSorted[i] = T[i];
    }

    sort(H, H + N + 1);
    sort(TSorted, TSorted + Q);

    unordered_map<long long, long long> answers;

    long long currSum = 0;
    long long currThrows = 0;
    int qi = 0;
    for (int i = N; i > 0 && qi < Q; i--) {
        int numPinecones = N - i + 1;
        long long numThrows = numPinecones * (H[i] - H[i-1]);

        while (qi < Q && TSorted[qi] < currThrows + numThrows) {
            long long extra = TSorted[qi] - currThrows;
            long long fullLines = extra / numPinecones;
            long long base = H[i] - fullLines;
            long long remainingThrows = extra % numPinecones;
             long long ans = (currSum + (numPinecones * ((H[i] * (H[i] + 1) - base *
(base + 1)) / 2) % MOD) % MOD + (remainingThrows * base) % MOD) % MOD;
            answers[TSorted[qi++]] = ans;
        }

        currThrows += numThrows;
        currSum = (currSum + (numPinecones * (((H[i] * (H[i] + 1) - H[i-1] * (H[i-1] +
1)) / 2) % MOD)) % MOD) % MOD;
    }

    while (qi < Q) answers[TSorted[qi++]] = currSum;

    for (int i = 0; i < Q; i++) cout << answers[T[i]] << '\n';
}
```

The Grand Tree Subtask 2 Python Brute Force Solution

```
from itertools import combinations
N, M = map(int, input().split())
t = list(map(int, input().split()))

conn = [[] for _ in range(N)]

for i in range(N-1):
  x, y = map(int, input().split())
  conn[x].append(y)
  conn[y].append(x)

parents = [0] * N

def find_parents(node, parent = 0):
  parents[node] = parent
  for c in conn[node]:
    if c != parent:
      find_parents(c, node)

find_parents(0)

res = 10000
for size in range(1, N+1):
  for subset in combinations(range(N), size):
    tastiness = 0
    for x in subset:
      if parents[x] not in subset:
        break
      tastiness += t[x]
    else:
      if tastiness >= M:
        res = min(res, tastiness)

print(res)
```

The Grand Tree Subtask 2 Recursive Python Solution

```
from itertools import combinations
N, M = map(int, input().split())
t = list(map(int, input().split()))

conn = [[] for _ in range(N)]

for i in range(N-1):
  x, y = map(int, input().split())
  conn[x].append(y)
  conn[y].append(x)

def calculate_T(i, parent = 0):
  T_i = {t[i]}

  for c in conn[i]:
    if c == parent: continue
    T_c = calculate_T(c, i)

    T_i = T_i | set(a + b for a in T_c for b in T_i)
```

```
  return T_i

T_0 = calculate_T(0)
print(min(filter(lambda x: x >= M, T_0)))
```

The Grand Tree Full Python Solution

```
import sys
sys.setrecursionlimit(5000)

N, M = map(int, input().split())
t = list(map(int, input().split()))

conn = [[] for _ in range(N)]

for i in range(N-1):
  x, y = map(int, input().split())
  conn[x].append(y)
  conn[y].append(x)

flattened = []
next_array = [-1] * N

def flatten(node, parent = 0):
  idx = len(flattened)
  flattened.append(t[node])

  for c in conn[node]:
    if c == parent:
      continue

    flatten(c, node)

  next_array[idx] = len(flattened)

flatten(0)

cache = [[-1] * (M + 1) for _ in range(N)]

def dp(i, m):
  if m <= 0: return 0
  if i >= N: return 1000000

  if cache[i][m] == -1:
    cache[i][m] = min(dp(next_array[i], m), flattened[i] + dp(i + 1, m - flattened[i]))

  return cache[i][m]

print(dp(0, M))
```

The Grand Tree Full C++ Solution

```
#include <bits/stdc++.h>

using namespace std;

int N, M, t[100000], dp[2005][2005];
vector<int> conn[100000];
```

```
void dfs(int node, int next, int p = -1) {
    int curr_next = next;
    for (int c: conn[node]) {
        if (c != p) {
            dfs(c, curr_next, node);
            curr_next = c;
        }
    }

    for (int m = 0; m <= M; m++) {
        dp[node][m] = min(dp[next][m], t[node] + (m > t[node] ? dp[curr_next][m -
t[node]] : 0));
    }
}

int main() {
    cin >> N >> M;
    for (int i = 0; i < N; i++) cin >> t[i];
    for (int i = 0; i < N-1; i++) {
        int x, y;
        cin >> x >> y;
        conn[x].push_back(y);
        conn[y].push_back(x);
    }

    for (int m = 1; m <= 2000; m++) dp[N][m] = 5000;

    dfs(0, N);

    cout << dp[0][M] << endl;
}
```

Big O Complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million (100,000,000) operations per second. For example, if the problem specifies a time limit of 1 second and an input of N as large as 100,000, then you know that an $O(N^2)$ algorithm might be too slow for large N since $100{,}000^2 = 10{,}000{,}000{,}000$, or 10 billion operations.

Time Complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a + b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry i corresponds to, but these will take the same amount of computation regardless of N. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a constant number of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O(N^2)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb in contests, $O(N^2)$ algorithms are only useful for input sizes of $N \lesssim 10{,}000$. Usually, if you have a nested loop in your program (loop inside a loop) then your solution is $O(N^2)$ if both these loops run about N times.