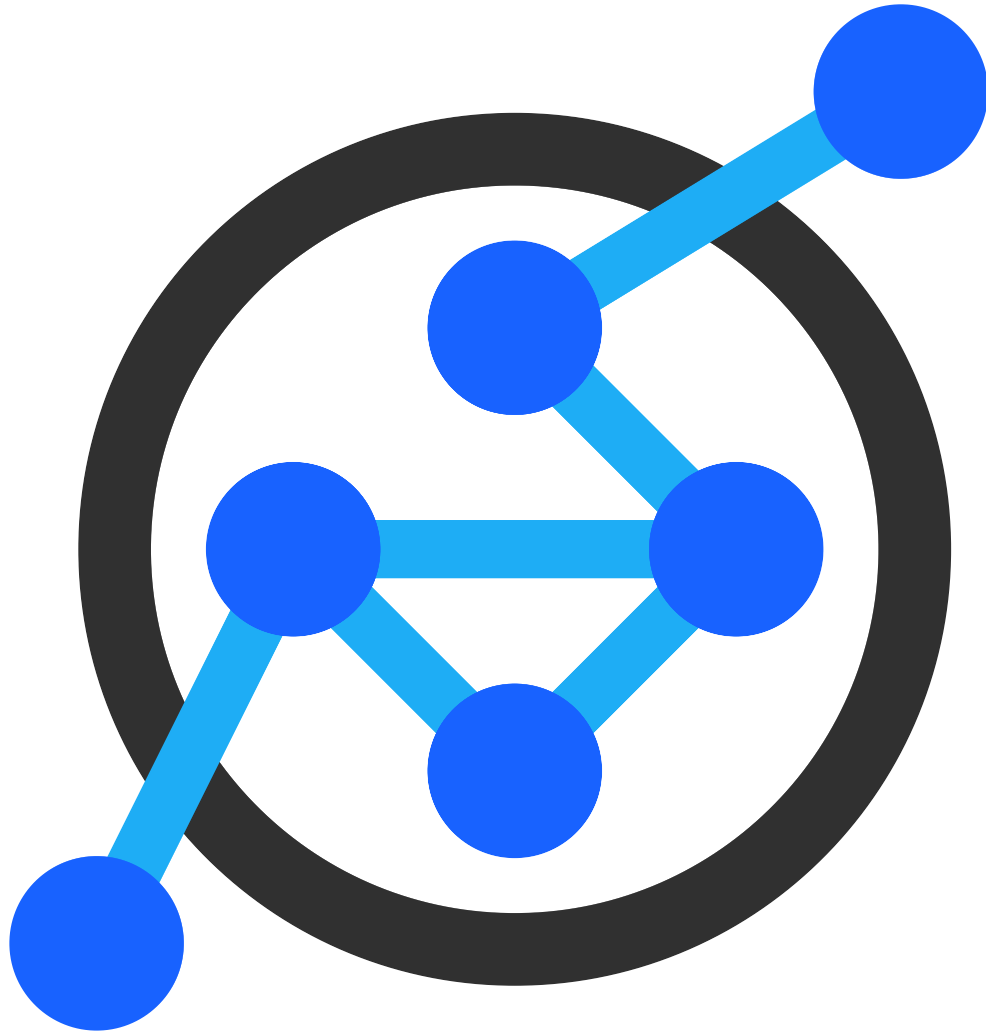
First Round SOI 2023/2024

Solution Booklet



Swiss Olympiad in Informatics

15 September – 30 November 2023

Harvest Day

Task Idea	Johannes Kapfhammer
Task Preparation	Anna Khanova
Description English	Anna Khanova
Description German	Priska Steinebrunner
Description French	Théo von Düring
Solution	Anna Khanova, Charlotte Knierim
Correction	Charlotte Knierim

Subtask 1: Is it even possible? (17 points)

To determine if it's possible to ensure each basket has an equal number of items, you need to check if the total number of items is divisible by the number of baskets. If the total number of items % number of baskets is 0, it's possible; otherwise, it's not.

Subtask 2: One large basket (18 points)

Since all farmers bring exactly one item, except for one who might bring more, the solution is simple. There is a guarantee, that it's possible to distribute equally. The minimum number of moves would be the difference between the number of items the special farmer brought and the average number of items per basket.

Subtask 3: Everyone brings almost the same amount (19 points)

Given the maximum and minimum number of items differ by at most 2, you can calculate the total number of items and divide it by the number of baskets to find the average. Since the difference is small, the number of moves will be minimal and can be calculated by distributing the excess items from the baskets with the most items to the one with the least.

Subtask 4: General case (21 points)

For the general case, you need a more complex algorithm. Calculate the average number of items per basket after summing all items. Then, for each basket, calculate the difference from the average. The total number of moves will be the sum of these differences (ignoring signs, as you're interested in the absolute value).

Subtask 5: A lot of harvest (25 points)

This is similar to Subtask 4 but requires handling large numbers (up to one billion). This can be achieved by using 64-bit numbers: e.g. *long long* type in C++.

Exhibition

Task Idea	Johannes Kapfhammer
Task Preparation	Jan Schär
Description English	Jan Schär
Description German	Jan Schär
Description French	Théo von Düring
Solution	Jan Schär

In this task, we are given the values of N paintings, each painted by one of K painters. For each number i of paintings from 2 to N, we should compute the maximum total value of a selection of i paintings, with the constraint that we cannot have exactly one painting from any painter.

Subtask 1: A single painter (7 points)

With a single painter, we always have at least two paintings from the painter, so we can ignore the one-painting constraint. For each i from 2 to N, we need to find the sum of the i biggest values.

We can do that by sorting the paintings decreasingly, and then accumulating the sum of the i biggest values in a variable, while collecting each sum in a list. The value at index i of the list is the sum of the i biggest values. At the end, we skip the first two elements of the list, because we need at least two paintings.

```
1  def parse():
2      k, n = map(int, input().split())
3      rows = []
4      for i in range(k):
5          line = list(map(int, input().split()))
6          rows.append(line[1:])
7      return n, rows
8
9  def solve(n, rows):
10     ans = [0]
11     sum = 0
12     for value in sorted(rows[0], reverse=True):
13         sum += value
14         ans.append(sum)
15     return ans[2:]
16
17 for case in range(int(input())):
18     print(f"Case #{case}:", " ".join(map(str, solve(*parse()))))
```

Running time is $O(N \log N)$ due to the sorting, space usage is $O(N)$.

Subtask 2: Two painters (23 points)

In this subtask, we have two painters. We can solve this by case distinction: Either we have paintings only from the first painter, only from the second, or from both. We can try out all three cases, and then take the maximum from all three for each output number. In each case, we always take the most expensive two paintings from each selected painter, and then take all remaining paintings and sort by decreasing value.

```
1  def solve(n, rows):
2      ans = [0] * (n + 1)
3      for painters in [[0], [1], [0, 1]]:
4          sum = 0
5          count = 0
```

```
6         paintings = []
7         for painter in painters:
8             sorted_row = sorted(rows[painter], reverse=True)
9             sum += sorted_row[0] + sorted_row[1]
10            count += 2
11            paintings += sorted_row[2:]
12        ans[count] = max(ans[count], sum)
13        for value in sorted(paintings, reverse=True):
14            sum += value
15            count += 1
16            ans[count] = max(ans[count], sum)
17    return ans[2:]
```

Subtask 3: More painters (31 points)

Here, we have up to 1000 painters. We cannot do an explicit case distinction over all subsets of painters, because the number of cases is exponential in the number of painters. But instead, we can use dynamic programming: When we know the result for a subset of painters, we can extend the result with one additional painter.

The intermediate result stores at index i the value of the most expensive exhibition with i paintings from painters seen so far, or -1 if such an exhibition is not possible. Initially, only an exhibition with no paintings is possible.

For each painter, we then first compute the sums for just this painter, as in subtask 1. Then we update the intermediate result. Here we need to be careful to do this in an order where we don't update values and then read them again later in the same update, because then we could be double counting paintings. This is why we update in reversed order, starting with the last entry. For each entry, we compute the new maximum, which is either the old value (we don't take paintings from this painter), or the sum of taking any number ≥ 2 of paintings from the painter, plus the previous answer for that many fewer paintings.

```
1 def solve(n, rows):
2     ans = [-1] * (n + 1)
3     ans[0] = 0
4     for row in rows:
5         sum = 0
6         sums = [0]
7         for value in sorted(row, reverse=True):
8             sum += value
9             sums.append(sum)
10        for i in reversed(range(1, n + 1)):
11            for j in range(max(0, i - len(row)), i - 1):
12                if ans[j] != -1:
13                    ans[i] = max(ans[i], ans[j] + sums[i - j])
14    return ans[2:]
```

Running time is $O(N^2)$, space usage is $O(N)$.

Subtask 4: A lot of painters (39 points)

In the last subtask, there are so many paintings that the $O(N^2)$ running time of the DP solution is too slow. (At least it is intended to be – fast DP solutions were unfortunately able to squeeze through the time limit.)

We can go faster with a greedy solution. Bundle the two best paintings of each painter as a pair and give it its average value. All remaining paintings are left single. Sort all pairs and singles by decreasing value, with pairs before singles of equal value.

Because the pair of each painter will be sorted before all his singles, any prefix of this sorted sequence is a valid selection. The selection is also optimal for its size. For example, the sequence could look like this (A, B, C are the painters):

```
|AA| |A| |BB| |CC| |DD| |D| |B| |A|
```

We can now greedily add pairs and singles from this sequence. However, this does not work for sizes where the cut is in the middle of a pair, like in this example:

```
                    |
|AA| |A| |BB| |C|C| |DD| |D| |B| |A|
                    |
```

We cannot take just one painting from the pair at the cut, so instead we do a case distinction:

1. Either we take the next best single from a painter whose pair we already have (`|B|` in the example).
2. Or, we drop the lowest single before the cut, and add the pair at the cut. (In the example, we remove `|A|` and add `|CC|`.)
3. Finally, if there is another pair directly before the pair at the cut, we can remove that pair and instead add the best triple among the remaining painters, i.e., the maximal sum of the three best paintings. (In the example, we remove `|BB|` and add `|DD|` and `|D|`.)

Why are these all the cases we need to check? First, if we do not remove any paintings before the cut, then the only way to reach the target number is to add a single painting, and we take the best possible (case 1). Otherwise, we remove at least one painting.

It does not make sense to remove a pair that is not directly next to the pair at the cut. If this were the case, we would have at least four selected paintings after the removed pair. We can improve the solution by adding back the removed pair, and then either removing two singles after it if there are at least two, or otherwise removing a pair with no singles (which must exist because we have at least two pairs and at most one single).

If we remove a pair directly next to the pair at the cut, then it does not make sense to additionally remove singles before the removed pair. We can improve the solution by removing the lowest group (either a pair or single), and then adding back a removed pair or single.

It does not make sense to remove more than one single, because we could improve the solution by removing the lowest group (either a pair or single), and adding one or two removed singles back.

If we do remove exactly one single, then it's optimal to remove the lowest single, and add the pair at the cut (case 2).

Finally, if we remove the pair directly before the pair at the cut, then we must be adding three paintings. This only makes sense if these three are the pair and best single of the same painter. If we instead add three singles, or a pair and single from different painters, then we could improve the solution by adding back the removed pair, and removing either two singles or the pair. That means, it's optimal to take the best triple (case 3).

Implementation

For implementing the case distinction, we need to be able to efficiently look up various things. Case 2 is easy, we just remember the last single we added in `last_single`.

For case 1, we need to find the best single which we have not taken yet, from a painter whose pair we already have. We do this by keeping all such singles (not just the best) in a priority queue (`restheap`). Because Python only has a min heap, and we need a max heap, we store values negated.

For case 3, we prepare a sorted list of the sum of the best three paintings from each painter. We need to skip triples where we already have taken the pair, so we remember which pairs we have taken (`painter_taken`), and remove those triples when they reach the top of the triples list.

```python
import heapq

def solve(n, rows):
    ans = [0] * (n + 1)

    firsttwo = []
    firstthree = []
    rest = []
    for i, row in enumerate(rows):
        row = sorted(row, reverse=True)
        firsttwo.append((row[0] + row[1], i))
        if len(row) >= 3:
            firstthree.append((row[0] + row[1] + row[2], i))
        rest.append(row[2:])
    firsttwo.sort()
    firstthree.sort()

    count = 0
    sum = 0

    painter_taken = [False] * len(rows)
    restheap = []

    last_single = -1
    prev_pair = -1

    while len(firsttwo) != 0 or len(restheap) != 0:
        if len(firsttwo) != 0 and (len(restheap) == 0 or
                firsttwo[-1][0] >= -restheap[0] * 2):
            # Case 1
            if len(restheap) != 0:
                ans[count + 1] = sum + -restheap[0]

            # Case 3
            while (len(firstthree) != 0 and
                    painter_taken[firstthree[-1][1]]):
                firstthree.pop()
            if prev_pair != -1 and len(firstthree) != 0:
                ans[count + 1] = max(ans[count + 1],
                    sum - prev_pair + firstthree[-1][0])

            prev_pair, painter_i = firsttwo.pop()
            count += 2
            sum += prev_pair
            for val in rest[painter_i]:
                heapq.heappush(restheap, -val)
            painter_taken[painter_i] = True

            # Case 2
            if last_single != -1:
                ans[count - 1] = max(ans[count - 1], sum - last_single)
        else:
            count += 1
            last_single = -heapq.heappop(restheap)
            sum += last_single
            prev_pair = -1
        ans[count] = sum

    return ans[2:]
```

Running time is $\mathcal{O}(N \log N)$ due to the priority queue, space usage is $\mathcal{O}(N)$.

Cruise Trip

Task Idea	Benjamin Faltin
Task Preparation	Théo von Düring
Description English	Théo von Düring
Description German	Benjamin Schmid
Description French	Théo von Düring
Solution	Théo von Düring, Johannes Kapfhammer

In this task, we are given a grid with N rows and M columns representing a section of the Nile river with some islands scattered everywhere. We then need to find the length of the longest path going from the highest to the lowest row without backtracking or moving upwards. We also know that there always exists a valid path going from top to bottom.

Subtask 1: A first trip (20 points)

In this subtask, we can realize that we have very few possible configurations for each row since $M = 2$. Each row must be in one of 3 configurations: ".." or ".#" or "#.". The configuration "##" can't happen because it would block all paths and we wouldn't get any solution.

Since we cannot go upstream the longest path going from the top to the current row only depends on the longest path going from the top to the left column and to the right column of the row right above the current one. By splitting the 3 cases we can solve the problem quickly.

```python
def parse_dim():
    return map(int, input().split())

def parse_grid(n):
    for _ in range(n):
        yield list(map(lambda x: {"#":0, ".":1}[x], input())) # 1: free, 0: occupied

def solve(n, m):
    col1 = 0
    col2 = 0
    for row in parse_grid(n):
        if row[0] and row[1]: # both space are free
            col1, col2 = max(col1+1, col2+2), max(col1+2, col2+1)
        elif row[0]:
            col1 += 1
            col2 = -1_000_000_000
        else:
            col1 = -1_000_000_000
            col2 += 1
    return max(col1, col2)

for case in range(int(input())):
    print(f"Case #{case}:", solve(*parse_dim()))
```

Running time is $\mathcal{O}(N)$, space usage is $\mathcal{O}(1)$

Subtask 2: Heading to Asyut (25 points)

Here, the grid is a bit wider with a max width of $M = 20$ cells. We can try to generalise our previous strategy by finding the length of the longest path going through all the cells of each row. To do so, we can find all the ways to get to a cell on the current row from the previous row and keep track of the maximum length amoung these different possible paths.

```python
def solve(n, m):
    prev_row = [0] * m
    curr_row = [-1_000_000_000] * m
    for row in parse_grid(n):
        for i in range(m):
            if (not row[i]) or (prev_row[i] < 0):
                continue

            for j in range(i, m): # update values after index
                if not row[j]:
                    break
                curr_row[j] = max(curr_row[j], prev_row[i]+j-i+1)

            for j in range(i-1, -1, -1): # update values before index
                if not row[j]:
                    break
                curr_row[j] = max(curr_row[j], prev_row[i]+i-j+1)
        prev_row, curr_row = curr_row, prev_row
        curr_row = [-1_000_000_000] * m

    return max(prev_row)
```

Running time is $O(N * M^2)$, space usage is $O(M)$

Subtask 3: Problems begin (25 points)

Until now we have solved the problem for small values of M and we can easily see that our last solution isn't viable for large values of M in the same order of magnitude than N. Before tackling the general problem we can try to simplify the task by removing one direction. Now we can only move from left to right and from top to bottom.

In the previous solution, we had a lot of unnecessary calculations when going from the i-th to the $i + 1$-th row. Instead of calculating all the paths going to one cell from the last row, we know that, if the longest path goes through one cell, then it must pass either through the cell directly to it's left or to it's top. Therefore, knowing the maximum path length going through the cell above and to the left is enough to get the longest path going through the current cell.

Our solution will be able to solve the problem for each cell by reading each row from left to right and for each cell taking the maximum of the value in the cell above plus 1 and to the left plus 1.

```python
def solve(n, m):
    curr_row = [0] * m

    for row in parse_grid(n):
        for i in range(m): # Left to right
            if not row[i]:
                curr_row[i] = -1_000_000_000
                continue

            if i > 0:
                curr_row[i] = max(curr_row[i], curr_row[i-1])
            curr_row[i] += 1

    return max(curr_row)
```

Running time is $O(N * M)$, space usage is $O(M)$

Subtask 4: Getting to Cairo (30 points)

In the last subtask, we finally look at the general solution. To solve it, we can adapt the solution for subtask 3. In subtask 3 we only went from left to right, now we are going in both directions. A key observation is that, like before, the value of a cell only depends on it's neighbours' value.

However, since we cannot backtrack, the path can go through each row in only one direction. The boat cannot go right to left then left to right on the same row.

We will implement the same solution as subtask 3 except we will solve for both directions seperately and merge the results for each row.

```python
def solve(n, m):
    prev_row = [0] * m
    curr_row_lr = [0] * m
    curr_row_rl = [0] * m

    for row in parse_grid(n):
        for i in range(m): # Left to right
            if not row[i]:
                curr_row_lr[i] = -1_000_000_000
                continue

            curr_row_lr[i] = prev_row[i]
            if i > 0:
                curr_row_lr[i] = max(curr_row_lr[i], curr_row_lr[i-1])
            curr_row_lr[i] += 1

        for i in range(m-1, -1, -1): # Right to left
            if not row[i]:
                curr_row_rl[i] = -1_000_000_000
                continue

            curr_row_rl[i] = prev_row[i]
            if i < m-1:
                curr_row_rl[i] = max(curr_row_rl[i], curr_row_rl[i+1])
            curr_row_rl[i] += 1

        for i in range(m): # merging both directions' results
            prev_row[i] = max(curr_row_lr[i], curr_row_rl[i])

    return max(prev_row)
```

Running time is $O(N * M)$, space usage is $O(M)$

Calendar

Task Idea	Christopher Burckhardt
Task Preparation	Anna Khanova
Description English	Anna Khanova, Charlotte Knierim
Description German	Charlotte Knierim
Description French	Théo von Düring
Solution	Anna Khanova, Charlotte Knierim
Correction	Charlotte Knierim

Subtask 1: Tough start (16 points)

Conditions: k=0 (no camp planning), all IMOI events start on the first day.

Solution: Since all events start on the first day, Mouse Binna should plan this day first to maximize her productivity score. She starts with a productivity score of n-1 and it decreases each day. Therefore, she should plan the day with the most events when her productivity is highest.

Subtask 2: Hard Times (12 points)

Conditions: k=0, IMOI events can start on any day.

Solution: In this case, Binna should prioritize days with the highest number of events. She should start with the day that has the most events and work her way down to the day with the fewest. This ensures that her higher productivity scores are used on days with more events.

Implementation details: Create an array a of length n with $a[i]$ representing the number of events on day i. For each segment $[l;r]$ increment $a[i]$ for all $i \in [l;r]$. Time complexity here is $O(n^2)$. Next, assign productivity scores starting from $n-1$ to the day with the most events, decreasing for subsequent days. Sorting for this can also be achieved in $O(n^2)$ time complexity.

Subtask 3: Overwhelmed (21 points)

Conditions: Stofl helps with camp planning, a longer month, and many events.

Solution: The strategy remains similar to Subtask 2. However it requires more careful implementation.

Implementation details: Use *lazy incrementations* for efficiency. For each segment $[l;r]$ adjust the array as $a[l]+=1$ and $a[r+1]-=1$. Process all segments and then cumulatively add the value of the previous array element to the next ($a[i]+=a[i-1]$). This approach maintains the same result as Subtask 2 but reduces the time complexity to $O(n)$. After forming this array, assign productivity scores from $n-1$ downwards, using an efficient sorting algorithm (like *std::sort* in C++) resulting in a time complexity of $O(n \log n)$.

Subtask 4: No help (18 points)

Conditions: Binna plans the camp, shorter month, fewer events.

Solution: For this subtask there are many possible solutions. One approach is to reuse the solution from the previous subtask and implement brute-force for determining the camp's start date. The time complexity for this method is $O(n^2 \log n)$

Subtask 5: Lot's of work (33 points)

Conditions: Longer months, many events, camp planning required.

Key observations:

- If there are x days with i events and the highest productivity for these days is P, the productivity scores will range from P to $P - x + 1$.
- Consider the camp is from day l to day r. Let x, y be the numbers of events on days $l, r + 1$ correspondingly. If we shift the camp one day later (from day $l + 1$ to $r + 1$), the event count on day l changes to $x - 1$ and on day $r + 1$ to $y + 1$, affecting the number of days with $x - 1$, x, y and $y - 1$ events. All other counts remain unchanged.

Solution: Calculate answer for camp planning starts at day 0 and save intermediate data. For each number of events i, record the number of days with i events and what is the highest/lowest productivity for the days with i events. The main difficulty of this task is to carefully recalculate all this data in no worse then $O(\log n)$ on each shift of the camp start. Final time complexity is: $O(n \log n)$

Pyramids

Task Idea	Johannes Kapfhammer
Task Preparation	Andrei Feodorov
Description English	Andrei Feodorov
Description German	Jan Schär
Description French	Théo von Düring
Solution	Yaël Arn, Andrej Ševera

Summary

In this task, Stofl needs our help to reorder a pile of N stones so that they are sorted in descending order of size, forming a pyramid. Each stone has a distinct size, denoted by 0 to $N-1$, and the stones initially lie on the first stable patch. You get the number of additional stable patches where Stofl can deposit stones in the input. As the stones are heavy, Stofl can only carry the topmost stone of a stable patch to the top of another stable patch. You have to output a sequence of moves sorting the pile.

Subtask 1: Being the apprentice (14 points)

In the first subtask, you don't need to sort the stones but get the moves you have to do in the input. To simulate the process, we need a data structure that supports three types of operations:

1. Push a new element to the top of a pile
2. Get the element on top of a pile
3. Delete the element on the top of a pile

A stack can perform all these operations, so we can see a pile as a stack, and a move as popping from the first stack and pushing to the second.

```python
for t in range(int(input())):
n, k, m = map(int, input().split())
pyramids = [input().split()[1:] for i in range(k)]
for i in range(m):
s, e = map(int, input().split())
# simulate the moving of the stone
pyramids[e].append(pyramids[s].pop())
print(f"Case #{t}:")
for i in range(k):
print(len(pyramids[i]), *pyramids[i])
```

Subtask 2: Pyramid of Ahmose (16 points)

In this subtask, there are $K = 3$ stable patches and $1 \leq N \leq 10$ stones, and you need to sort the stones in less than 27000 moves. Any sorting algorithm can be adapted to work on 3 stacks, but the easiest to implement would be insertion sort. For this, we would first move all stones onto the third stable patch, and then insert stones repeatedly by moving all greater stones to stable patch number two, moving the new stone to the first stable patch, and then moving the greater stones back to the first stable patch.

Subtask 3: Pyramid of Menkaure (30 points)

In this subtask, there are $K = 3$ stable patches and $1 \leq N \leq 500$ stones, and you need to sort the stones in less than 4500 moves to get a full score. We will detail two different solutions to the problem as they both have interesting underlying ideas.

Mergesort solution

By looking at the different sorting algorithms, we can notice that mergesort seems to be a good choice for this problem. One crucial observation for implementing all solutions correctly is that when you move a sequence to another stack, its order is reversed. If we implement mergesort naively, first splitting the array in half, moving both halves to different stacks, and then recursively sorting both stacks in reverse order and merging again at the end, we get a number of moves that is approximately equal to $2n \log n$, which is worth around 15 points.

To optimize this, we can search for wasteful moves we make. In the naive version, one call to our function consists of three parts:

1. move the two halves of the pile to the other stable patches
2. make recursive calls
3. merge the sorted piles back onto the initial stable patch

Of these three steps, the one that seems the most wasteful is the first one, as we need to merge and make recursive calls in the logic of mergesort. For merging, we still need the piles of stones to be on the other stable patches, so we need to find a way to sort not only in such a way that the sorted sequence ends up on the initial stable patch but also make it possible to end up on any other stable patch.

To achieve this, we define a function $move(s, e, n, ord)$, which sorts the first n elements in order (ascending/descending) ord of the stable patch s such that the sorted sequence ends up on stable patch e. For merging, we need two stacks sorted in reverse order on the two patches which are not the patches on which the sequence should end. So for a call of $move(s, e, n, ord)$, we can recurse to $move(s, i, n/2, !ord)$ for both $i \neq e$.

```
1  #include "bits/stdc++.h"
2
3  using namespace std;
4
5  vector<vector<int>> pyramids;
6  vector<pair<int, int>> moves;
7
8  void makeMove(int s, int e) {
9          if (s == e) return;
10         moves.push_back({s, e});
11         pyramids[e].push_back(pyramids[s].back());
12         pyramids[s].pop_back();
13 }
14
15 // sort the first n stones of stable patch s in order ord so that they end up on pile e
16 void move(int s, int e, int n, bool ord) {
17         if (n <= 1) { // the base case when n == 1
18                 makeMove(s, e);
19                 return;
20         }
21         if ((e + 1) % 3 == s) { // if we have s == e for one stack, then do it last
22                 move(s, (e + 2) % 3, (n + 1)/2, !ord); // recurse to floor(n/2) and ceil(n/2)
23                 move(s, (e + 1) % 3, n / 2, !ord);
24         } else {
25                 move(s, (e + 1) % 3, n / 2, !ord); // recurse to floor(n/2) and ceil(n/2)
26                 move(s, (e + 2) % 3, (n + 1)/2, !ord);
27         }
28         int counterE1 = 0, counterE2 = 0;
```

```
29         for (int i = 0; i < n; ++i) { // merge both back together
30                 if (counterE1 >= n / 2) {
31                         makeMove((e + 2) % 3, e);
32                         counterE2++;
33                 } else if (counterE2 >= (n + 1)/2) {
34                         makeMove((e + 1) % 3, e);
35                         counterE1++;
36                 } else if ((pyramids[(e + 1) % 3].back() < pyramids[(e + 2) % 3].back()) == ord) {
37                         makeMove((e + 2) % 3, e);
38                         counterE2++;
39                 } else {
40                         makeMove((e + 1) % 3, e);
41                         counterE1++;
42                 }
43         }
44 }
45
46 signed main() {
47         ios_base::sync_with_stdio(false);
48         cin.tie(nullptr);
49         int t; cin >> t;
50         for (int i = 0; i < t; ++i) {
51                 int n, k; cin >> n >> k;
52                 pyramids = vector<vector<int>>(k);
53                 for (int j = 0; j < n; ++j) {
54                         int stone; cin >> stone;
55                         pyramids[0].push_back(stone);
56                 }
57                 move(0, 0, n, true);
58                 cout << "Case #" << i << ": " << moves.size() << "\n";
59                 for (auto [start, end]: moves) {
60                         cout << start << " " << end << "\n";
61                 }
62                 moves.clear();
63         }
64 }
```

This gives us 27 points.

To get the full score, we still need to optimize our solution. We can do this by looking at the small cases, for example $n \leq 4$, and brute-forcing the best way to sort these cases with a precomputed BFS.

```
1         // compute the best move for all positions of si stones on k stable patches
2         map<vector<vector<int>>, pair<int, int>> bfs(int si, int k) {
3             vector<vector<int>> startSt(k);
4             for (int i = 0; i < si; ++i) { // the sorted sequence is the starting position
5             startSt[0].push_back(i);
6             }
7             map<vector<vector<int>>, pair<int, int>> moveTable;
8             queue<pair<vector<vector<int>>, pair<int, int>>> q;
9             q.push({startSt, {-1, -1}});
10            while (!q.empty()) {
11                auto [pyra, lastMove] = q.front();
12                q.pop();
13                if (moveTable.find(pyra) != moveTable.end()) continue;
14                moveTable[pyra] = lastMove;
15                for (int i = 0; i < pyra.size(); ++i) { // simulate all possible moves
16                for (int j = 0; j < pyra.size(); ++j) {
17                    if (i == j || pyra[i].empty()) continue;
18                    pyra[j].push_back(pyra[i].back());
19                    pyra[i].pop_back();
20                    q.push({pyra, {i, j}});
21                    pyra[i].push_back(pyra[j].back());
22                    pyra[j].pop_back();
23                }
24                }
25            }
```

```
26            return moveTable;
27        }
```

To make use of this, we need to use coordinate compression when we come to a small case.

```
1         // ...
2         vector<pair<int, int>> moves;
3         vector<map<vector<vector<int>>, pair<int, int>>> preCompMoves;
4         constexpr int BRUTEFORCELIMIT = 4
5         // ...
6         void move(int s, int e, int n, bool ord) {
7             if (n <= BRUTEFORCELIMIT) { // use the precomputed values for small n
8             vector<int> topElements(n);
9             for (int i = 0; i < n; ++i) {
10                topElements[n - i - 1] = pyramids[s][pyramids[s].size() - i - 1];
11            }
12            vector<int> cooTE = topElements; // use coordinate compression
13            std::sort(cooTE.begin(), cooTE.end());
14            for (int i = 0; i < n; ++i) {
15                topElements[i] = std::lower_bound(cooTE.begin(), cooTE.end(), topElements[i])
16                - cooTE.begin();
17                if (ord) topElements[i] = n - topElements[i] - 1; // swap the order of the elements
18            }
19            vector<vector<int>> tmpPyra(pyramids.size()); // create dummy pyramids
20            tmpPyra[(s - e + 3) % 3] = topElements;
21            pair<int, int> nextMove = preCompMoves[n - 1][tmpPyra]; // get the optimal move
22            while (nextMove.first != -1) {
23                makeMove((nextMove.second + e) % 3, (nextMove.first + e) % 3);
24                // simulate the move
25                tmpPyra[nextMove.first].push_back(tmpPyra[nextMove.second].back());
26                tmpPyra[nextMove.second].pop_back();
27                nextMove = preCompMoves[n - 1][tmpPyra];
28            }
29            return;
30            }
31            // ...
32        }
33
34        signed main() {
35            // ...
36            int t; cin >> t;
37            for (int i = 1; i <= BRUTEFORCELIMIT; ++i) {
38                preCompMoves.push_back(bfs(i));
39            }
40            // ...
41        }
```

This finally passes for 30 points! It can be proven that we always stay under 4500 moves if we brute-force up to $n = 8$, but this was not necessary to get a full score. The complete solution can be found at `https://soi.ch/media/files/pyramids-yael_andrej_version_sub3_full.cpp`

Polyphase Merge Sort solution

Instead of using normal mergesort, one can do some research and find an algorithm called Polyphase Merge Sort. As in the other solution, it is based on the observation that the wasteful element of the standard merge sort is moving stacks before merging.

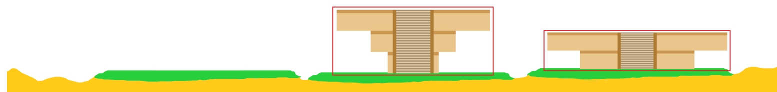
To understand the algorithm, we first need to define the term "run". In sorting a sorted subsequence, e.g. 1, 3, 10, 2, 3, 1, is called a run.

In polyphase merge sort, we will also merge two runs similarly to standard merge sort, but instead of splitting in the middle, we distribute runs unevenly. During a merge operation, we have one output stack and two input stacks. We merge the two top runs from the input stacks onto the output stack. Once one of the input stacks becomes empty, it becomes the new output stack. The idea behind polyphase merge sort is that we can distribute the runs in such a way that

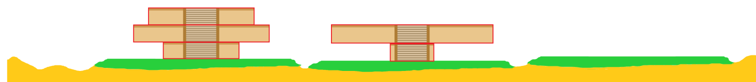
we can always continue merging until we have only one sorted run left, and don't need to move runs unnecessarily.

To find out how to distribute these runs, we can simulate the procedure backward. We also need to pay attention to the fact that the runs get reversed from ascending to descending or the other way around in one merge operation. We can ensure that the top runs of the input stacks are always sorted in the same order by alternating ascending and descending runs in the beginning. In the end, we want to have exactly one run on the first stable patch (red frame added to show runs move clearly).

In the round before, we thus need one run on each of the two other piles.



Going further, we then have one run on the first stable patch and two on another, and afterward two and three.



Continuing this, we find that these are exactly the Fibonacci numbers!

With this in mind, we can then implement this, using dummy elements to fill up until the next Fibonacci number.

```
#include "bits/stdc++.h"

using namespace std;

vector<vector<int>> pyramids;
vector<pair<int, int>> moves;

void makeMove(int s, int e) {
        if (pyramids[s].back() < 1e9) moves.push_back({s, e});
        pyramids[e].push_back(pyramids[s].back());
        pyramids[s].pop_back();
}

signed main() {
        ios_base::sync_with_stdio(false);
        cin.tie(nullptr);
```

```
17        int t; cin >> t;
18        for (int i = 0; i < t; ++i) {
19                int n, k; cin >> n >> k;
20                pyramids = vector<vector<int>>(k);
21                moves.clear();
22                for (int j = 0; j < n; ++j) {
23                        int stone; cin >> stone;
24                        pyramids[0].push_back(stone);
25                }
26                while (pyramids[0].size() < 610) { // add dummy runs
27                        pyramids[0].push_back(1e9);
28                }
29                while (pyramids[0].size() > 377) { // distribute runs initially
30                        makeMove(0, 1);
31                }
32                while (!pyramids[0].empty()) {
33                        makeMove(0, 2);
34                }
35                int outputStack = 0, inputStack1 = 1, inputStack2 = 2;
36                int inputStack1Size = 1, inputStack2Size = 1;
37                bool actDir = true;
38                while (true) {
39                        int actInpSt1S = inputStack1Size, actInpSt2S = inputStack2Size;
40                        while (actInpSt1S + actInpSt2S > 0) { // merge the two topmost runs
41                                if (actInpSt1S == 0) {
42                                        while (actInpSt2S--) makeMove(inputStack2, outputStack);
43                                        break;
44                                }
45                                if (actInpSt2S == 0) {
46                                        while (actInpSt1S--) makeMove(inputStack1, outputStack);
47                                        break;
48                                }
49                                if ((pyramids[inputStack1].back() > pyramids[inputStack2].back()) == actDir) {
50                                        makeMove(inputStack1, outputStack);
51                                        actInpSt1S--;
52                                } else {
53                                        makeMove(inputStack2, outputStack);
54                                        actInpSt2S--;
55                                }
56                        }
57                        actDir = !actDir;
58                        if (pyramids[inputStack2].empty() && pyramids[inputStack1].empty()) {
59                                break; // only one run left, we are done
60                        }
61                        // the empty stack is the new output stack
62                        if (pyramids[inputStack1].empty()) {
63                                swap(inputStack1, outputStack);
64                                inputStack1Size += inputStack2Size;
65                        }
66                        if (pyramids[inputStack2].empty()) {
67                                swap(inputStack2, outputStack);
68                                inputStack2Size += inputStack1Size;
69                        }
70                }
71                cout << "Case #" << i << ": " << moves.size() << "\n";
72                for (auto [s, e]: moves) {
73                        cout << s << " " << e << "\n";
74                }
75        }
76 }
```

This gives us 25 points.

To get more points, one has to notice that we can use runs that already exist in the input. Instead of distributing the runs greedily, we can use an unexpected technique in this setting: DP. For this, we notice that in polyphase merge sort, the number of times a run gets moved does not depend on the input. This allows us to precompute the number of times each run gets moved, and then use this information to minimize the total number of moves.

Let us define $dp[i][j]$ as the minimum number of moves using the first i stones and being at run number j.

For each state, we have three different transitions to consider:

1. We continue run j with stone i, but we need to make sure that stone i is greater / smaller than stone $i-1$, depending on run j being ascending or descending. This corresponds to $dp[i-1][j]+$ # moves elements of run j make
2. We already used i stones in the runs before, which corresponds to $dp[i][j-1]$
3. We start a new run with stone i, which corresponds to $dp[i-1][j-1]+$ # moves elements of run j make

As we have n^2 states, the runtime is $O(n^2)$. For simplicity, we have assumed that there are at most 233 runs in the input, which seems to be correct for all the generated inputs. To get the distribution of the runs, we need to backtrack through our DP.

```
#include "bits/stdc++.h"

using namespace std;

vector<vector<int>> pyramids;
vector<vector<int>> runSizes;
vector<pair<int, int>> moves;

const vector<int> movs{ // the precomputed number of moves for elements of a run
                 // out of space reasons, I haven't inserted the whole table
                8, 9, 8, 7, 8, 9, 8, 9, 10, 9
};

void makeMove(int s, int e) {
        if (pyramids[s].back() < 1e9) moves.push_back({s, e});
        pyramids[e].push_back(pyramids[s].back());
        pyramids[s].pop_back();
}

signed main() {
        ios_base::sync_with_stdio(false);
        cin.tie(nullptr);
        int t; cin >> t;
        for (int i = 0; i < t; ++i) {
                int n, k; cin >> n >> k;
                pyramids = runSizes = vector<vector<int>>(k);
                moves.clear();
                for (int j = 0; j < n; ++j) {
                        int stone; cin >> stone;
                        pyramids[0].push_back(stone);
                }
                // in the pair, the first integer is the cost and the second the optimal last state
                vector<vector<pair<int, int>>> dp(n + 1, vector<pair<int, int>>(233, {1e9, -1}));
                for (int j = 1; j < 233; ++j) {
                        dp[0][j] = {0, 1}; // for no stones, go one run back
                }
                dp[0][0] = {0, -1}; // no stones and no runs, base case
                for (int j = 0; j < n; ++j) {
                        for (int l = 0; l < 233; ++l) {
                                if (j == 0 || (pyramids[0][j - 1] < pyramids[0][j]) == (l % 2)) {
                                        // continue run with stone j
                                        dp[j + 1][l] = {dp[j][l].first + movs[l], 0};
                                }
                                // already use i stones
                                if (l > 0)
                                        dp[j + 1][l] = min(dp[j + 1][l], {dp[j + 1][l - 1].first, 1});
                                if (j > 0 && l > 0) // start a new run with stone j
                                        dp[j + 1][l] = min(dp[j + 1][l], {dp[j][l - 1].first + movs[l], 2});
                        }
```

```
50                  }
51                  int stoneNumber = n, runNumber = 232; // backtracking
52                  runSizes[0].resize(233);
53                  for (;;) {
54                          if (dp[stoneNumber][runNumber].second == -1) break;
55                          if (dp[stoneNumber][runNumber].second == 0) {
56                                  runSizes[0][runNumber]++;
57                                  stoneNumber--;
58                          } else if (dp[stoneNumber][runNumber].second == 1) {
59                                  runNumber--;
60                          } else if (dp[stoneNumber][runNumber].second == 2) {
61                                  runSizes[0][runNumber]++;
62                                  stoneNumber--;
63                                  runNumber--;
64                          }
65                  }
66                  while (runSizes[0].size() > 89) { // distribute runs
67                          for (int j = 0; j < runSizes[0].back(); ++j) {
68                                  makeMove(0, 1);
69                          }
70                          runSizes[1].push_back(runSizes[0].back());
71                          runSizes[0].pop_back();
72                  }
73                  int outputStack = 2, inputStack1 = 1, inputStack2 = 0;
74                  bool actDir = true;
75                  while (true) {
76                          int actInpSt1S = runSizes[inputStack1].back();
77                          int actInpSt2S = runSizes[inputStack2].back();
78                          runSizes[outputStack].push_back(actInpSt1S + actInpSt2S);
79                          runSizes[inputStack1].pop_back();
80                          runSizes[inputStack2].pop_back();
81                          while (actInpSt1S + actInpSt2S > 0) { // merge the two topmost runs
82                                  if (actInpSt1S == 0) {
83                                          while (actInpSt2S--) makeMove(inputStack2, outputStack);
84                                          break;
85                                  }
86                                  if (actInpSt2S == 0) {
87                                          while (actInpSt1S--) makeMove(inputStack1, outputStack);
88                                          break;
89                                  }
90                                  if ((pyramids[inputStack1].back() < pyramids[inputStack2].back()) == actDir) {
91                                          makeMove(inputStack1, outputStack);
92                                          actInpSt1S--;
93                                  } else {
94                                          makeMove(inputStack2, outputStack);
95                                          actInpSt2S--;
96                                  }
97                          }
98                          actDir = !actDir; // change sorting direction
99                          if (runSizes[inputStack1].empty() && runSizes[inputStack2].empty()) {
100                                 break; // only one run left, we are done
101                         }
102                         // the empty stack is the new output stack
103                         if (runSizes[inputStack1].empty()) {
104                                 swap(outputStack, inputStack1);
105                         }
106                         if (runSizes[inputStack2].empty()) {
107                                 swap(outputStack, inputStack2);
108                         }
109                 }
110                 cout << "Case #" << i << ": " << moves.size() << "\n";
111                 cerr << moves.size() << " ";
112                 for (auto [s, e]: moves) {
113                         cout << s << " " << e << "\n";
114                 }
115         }
116 }
```

This turns out to be enough to get a full score, even if cleverly constructed counterexamples could

break this approach.

Subtask 4: The pyramid complex of Dahshur (40 points)

In the last subtask, we have $K = 5$ stable patches, and up to 2500 moves to get a full score. To solve it, we can generalize our solutions from the last subtask.

Mergesort solution

Instead of splitting in two, we split in $k - 1$ sections and merge them back in the same way as in the last subtask.

```
#include "bits/stdc++.h"

using namespace std;

vector<vector<int>> pyramids;
vector<pair<int, int>> moves;
map<int, vector<map<vector<vector<int>>, pair<int, int>>>> preCompMoves;
constexpr int BRUTEFORCELIMIT = 4;
int K = 0;

void makeMove(int s, int e) {
        if (s == e) return;
        moves.push_back({s, e});
        pyramids[e].push_back(pyramids[s].back());
        pyramids[s].pop_back();
}

// compute the best move for all positions of si stones on k stable patches
map<vector<vector<int>>, pair<int, int>> bfs(int si, int k) {
        vector<vector<int>> startSt(k);
        for (int i = 0; i < si; ++i) { // the sorted sequence is the starting position
                startSt[0].push_back(i);
        }
        map<vector<vector<int>>, pair<int, int>> moveTable;
        queue<pair<vector<vector<int>>, pair<int, int>>> q;
        q.push({startSt, {-1, -1}});
        while (!q.empty()) {
                auto [pyra, lastMove] = q.front();
                q.pop();
                if (moveTable.find(pyra) != moveTable.end()) continue;
                moveTable[pyra] = lastMove;
                for (int i = 0; i < pyra.size(); ++i) { // simulate all possible moves
                        for (int j = 0; j < pyra.size(); ++j) {
                                if (i == j || pyra[i].empty()) continue;
                                pyra[j].push_back(pyra[i].back());
                                pyra[i].pop_back();
                                q.push({pyra, {i, j}});
                                pyra[i].push_back(pyra[j].back());
                                pyra[j].pop_back();
                        }
                }
        }
        return moveTable;
}

// sort the first n stones of stable patch s in order ord so that they end up on pile e
void move(int s, int e, int n, bool ord) {
        if (n <= BRUTEFORCELIMIT) { // use the precomputed values for small n
                vector<int> topElements(n);
                for (int i = 0; i < n; ++i) {
                        topElements[n - i - 1] = pyramids[s][pyramids[s].size() - i - 1];
                }
                vector<int> cooTE = topElements; // use coordinate compression
```

```
54                  std::sort(cooTE.begin(), cooTE.end());
55                  for (int i = 0; i < n; ++i) {
56                          topElements[i] = std::lower_bound(cooTE.begin(), cooTE.end(),topElements[i])
57                                                  - cooTE.begin();
58                          // swap the order of the elements
59                          if (ord) topElements[i] = n - topElements[i] - 1;
60                  }
61                  vector<vector<int>> tmpPyra(pyramids.size()); // create dummy pyramids
62                  tmpPyra[(s - e + K) % K] = topElements;
63                  pair<int,int> nextMove = preCompMoves[K][n - 1][tmpPyra]; // get the optimal move
64                  while (nextMove.first != -1) {
65                          makeMove((nextMove.second + e) % K, (nextMove.first + e) % K);
66                          // simulate the move
67                          tmpPyra[nextMove.first].push_back(tmpPyra[nextMove.second].back());
68                          tmpPyra[nextMove.second].pop_back();
69                          nextMove = preCompMoves[K][n - 1][tmpPyra];
70                  }
71                  return;
72          }
73          int split = (n + K - 2)/(K - 1);
74          int numGreat = n % (K - 1);
75          vector<int> counters(K);
76          for (int i = 0; i < K; ++i) { // recurse to ceil(n/k) and floor(n/k)
77                  if (i == e || i == s) continue;
78                  move(s, i, split, !ord);
79                  counters[i] = split;
80                  if (--numGreat == 0) split--;
81          }
82          if (e != s) { // do the in-place sorting last
83                  move(s, s, split, !ord);
84                  counters[s] = split;
85          }
86          for (;;) {  // merge both back together
87                  auto minIn = e;
88                  for (int i = 0; i < K; ++i) {
89                          if (i == e) continue;
90                          if (counters[i] > 0 &&
91                              (minIn == e || (pyramids[i].back() > pyramids[minIn].back()) == ord))
92                                  minIn = i;
93                  }
94                  if (minIn == e) break;
95                  makeMove(minIn, e);
96                  counters[minIn]--;
97          }
98  }
99
100 signed main() {
101         int t; cin >> t;
102         for (int i = 0; i < t; ++i) {
103                 int n; cin >> n >> K;
104                 // precompute optimal moves if not done already
105                 if (preCompMoves.find(K) == preCompMoves.end()) {
106                         for (int j = 1; j <= BRUTEFORCELIMIT; ++j) {
107                                 preCompMoves[K].push_back(bfs(j, K));
108                         }
109                 }
110                 pyramids = vector<vector<int>>(K);
111                 for (int j = 0; j < n; ++j) {
112                         int stone;
113                         cin >> stone;
114                         pyramids[0].push_back(stone);
115                 }
116                 move(0, 0, n, true);
117                 cout << "Case #" << i << ": " << moves.size() << "\n";
118                 for (auto [start, end]: moves) {
119                         cout << start << " " << end << "\n";
120                 }
121                 moves.clear();
```

```
122         }
123 }
```

This gives us 37 points, and to get a full score, we need to make some more optimizations like stopping early if the input is already sorted. A full solution can be found at `https://soi.ch/media/files/pyramids_yael_andrej_version_sub4_full.cpp`

Polyphase Merge Sort solution

We can't use Fibonacci numbers anymore, but again need to simulate the process backward. This is a bit more complicated, but can still be done by hand, again assuming a maximum of 349 runs in the input. For more infos on the numbers you get with this simulation see Fibonacci numbers of higher order.

```
1  #include "bits/stdc++.h"
2
3  using namespace std;
4
5  vector<vector<int>> pyramids;
6  vector<vector<int>> runSizes;
7  vector<pair<int, int>> moves;
8
9  const vector<int> movs{ // the precomputed number of moves for elements of a run
10                 // out of space reasons, I haven't inserted the whole table
11                 4, 5, 4, 5, 6, 5, 4, 5,
12 };
13
14 void makeMove(int s, int e) {
15         if (pyramids[s].back() < 1e9) moves.push_back({s, e});
16         pyramids[e].push_back(pyramids[s].back());
17         pyramids[s].pop_back();
18 }
19
20 signed main() {
21         ios_base::sync_with_stdio(false);
22         cin.tie(nullptr);
23         int t; cin >> t;
24         for (int i = 0; i < t; ++i) {
25                 int n, k; cin >> n >> k;
26                 pyramids = runSizes = vector<vector<int>>(k);
27                 moves.clear();
28                 for (int j = 0; j < n; ++j) {
29                         int stone; cin >> stone;
30                         pyramids[0].push_back(stone);
31                 }
32                 // in the pair, the first integer is the cost and the second the optimal last state
33                 vector<vector<pair<int, int>>> dp(n + 1, vector<pair<int, int>>(349, {1e9, -1}));
34                 for (int j = 1; j < 349; ++j) {
35                         dp[0][j] = {0, 1}; // for no stones, go one run back
36                 }
37                 dp[0][0] = {0, -1}; // no stones and no runs, base case
38                 for (int j = 0; j < n; ++j) {
39                         for (int l = 0; l < 349; ++l) {
40                                 if (j == 0 || (pyramids[0][j - 1] < pyramids[0][j]) == (l % 2)) {
41                                         // continue run with stone j
42                                         dp[j + 1][l] = {dp[j][l].first + movs[l], 0};
43                                 }
44                                 // already use i stones
45                                 if (l > 0)
46                                         dp[j + 1][l] = min(dp[j + 1][l], {dp[j + 1][l - 1].first, 1});
47                                 if (j > 0 && l > 0) // start a new run with stone j
48                                         dp[j + 1][l] = min(dp[j + 1][l], {dp[j][l - 1].first + movs[l], 2});
49                         }
50                 }
51                 int stoneNumber = n, runNumber = 348; // backtracking
52                 runSizes[0].resize(349);
```

```
53                  for (;;) {
54                          if (dp[stoneNumber][runNumber].second == -1) break;
55                          if (dp[stoneNumber][runNumber].second == 0) {
56                                  runSizes[0][runNumber]++;
57                                  stoneNumber--;
58                          } else if (dp[stoneNumber][runNumber].second == 1) {
59                                  runNumber--;
60                          } else if (dp[stoneNumber][runNumber].second == 2) {
61                                  runSizes[0][runNumber]++;
62                                  stoneNumber--;
63                                  runNumber--;
64                          }
65                  }
66                  while (runSizes[1].size() < 56) { // distribute runs
67                          for (int j = 0; j < runSizes[0].back(); ++j) {
68                                  makeMove(0, 1);
69                          }
70                          runSizes[1].push_back(runSizes[0].back());
71                          runSizes[0].pop_back();
72                  }
73                  while (runSizes[2].size() < 100) {
74                          for (int j = 0; j < runSizes[0].back(); ++j) {
75                                  makeMove(0, 2);
76                          }
77                          runSizes[2].push_back(runSizes[0].back());
78                          runSizes[0].pop_back();
79                  }
80                  while (runSizes[3].size() < 108) {
81                          for (int j = 0; j < runSizes[0].back(); ++j) {
82                                  makeMove(0, 3);
83                          }
84                          runSizes[3].push_back(runSizes[0].back());
85                          runSizes[0].pop_back();
86                  }
87                  int outputStack = 4;
88                  bool actDir = true;
89                  while (true) {
90                          runSizes[outputStack].push_back(0);
91                          int runSize = 0;
92                          for (int j = 0; j < 5; ++j) { // find size of the new run
93                                  if (j == outputStack) continue;
94                                  runSizes[outputStack].back() += runSizes[j].back();
95                                  runSize += runSizes[j].back();
96                          }
97                          // delete run sizes from the top of used stacks
98                          for (int j = 0; j < 5; ++j) {
99                                  if (j == outputStack) continue;
100                                 runSizes[j].pop_back();
101                         }
102                         while (runSize--) { // merge runs together
103                                 int minIn = -1;
104                                 for (int j = 0; j < 5; ++j) {
105                                         if (j == outputStack || runSizes[j].back() == 0) continue;
106                                         if (minIn == -1 ||
107                                                 (pyramids[j].back() < pyramids[minIn].back()) == actDir) {
108                                                 minIn = j;
109                                         }
110                                 }
111                                 makeMove(minIn, outputStack);
112                                 runSizes[minIn].back()--;
113                         }
114                         actDir = !actDir; // change sorting direction
115                         for (int j = 0; j < 5; ++j) {
116                                 if (j == outputStack) continue;
117                                 // the empty stack is the new output stack
118                                 if (runSizes[j].empty()) {
119                                         outputStack = j;
120                                         break;
```

```
121                                 }
122                         }
123                         int numEmpty = 0;
124                         for (int j = 0; j < 5; ++j) {
125                                 numEmpty += runSizes[j].empty();
126                         }
127                         if (numEmpty == 4) break; // only one run left, we are done
128                 }
129                 cout << "Case #" << i << ": " << moves.size() << "\n";
130                 cerr << moves.size() << " ";
131                 for (auto [s, e]: moves) {
132                         cout << s << " " << e << "\n";
133                 }
134         }
135 }
```

Labyrinth

Task Idea	Johannes Kapfhammer
Task Preparation	Bibin Muttappillil
Description English	Bibin Muttappillil
Description German	Charlotte Knierim
Description French	Théo von Düring
Solution	Bibin Muttappillil, Johannes Kapfhammer

Subtask 1: Mukhtar Tomb (12 points)

The graph is star shaped.

- if one is wrong, than a leaf must exist that can be fixed with one swap with center
- this move does not affect any other leaf
- thus, we can fix every leaf with one swap, and we can't fix more than one leaf with one swap
- solution: count all wrong leafs

Subtask 2: Dark Tomb (11 points)

There is only one torch.

- minimizing torch movement is minimizing the distance this torch moves
- solution: minimal distance from torch place to its target, using bfs or dfs

Subtask 3: Marathon Tomb (19 points)

The graph is a line.

- it does not make sense to cross torch paths
- solution: match left most torch with left most hole

Subtask 4: Tomb of Mouse Tutankhamun (26 points)

The graph is small.

- solve lines seperately and store deficit in the left path
- solve left path like a line

Subtask 5: Tomb of Great Pharaoh Stofl (32 points)

```
1 import sys
2 import resource
3
4 def solution():
5
6     # parse input
7     n = int(input())
8     torches = []
```

```
9       for i in range(n):
10          m, t = input().split()
11          m, t = int(m), int(t)
12          torches.append(m - t)
13
14      adj = [[] for _ in range(n)]
15      for _ in range(n-1):
16          v, u = [int(word) for word in input().split()]
17          adj[v].append(u)
18          adj[u].append(v)
19
20      # algorithm
21      def dfs(v, p):
22          result = 0  # sum of abs(diff) of all subtrees
23          diff = torches[v]  # number of torches missing (positive) or spare (negative) in this subtree
24          for u in adj[v]:
25              if u == p: continue
26
27              r, d = dfs(u, v)
28              result += r
29              diff += d
30          result += abs(diff)
31          return result, diff
32
33      return dfs(0, -1)[0]
34
35  if __name__ == "__main__":
36      sys.setrecursionlimit(1_000_000)
37      resource.setrlimit(resource.RLIMIT_STACK, [0x10000000, resource.RLIM_INFINITY])
38
39      T = int(input())
40      for test_case_number in range(T):
41          print(f'Case #{test_case_number}: {solution()}')
```

Running time and space usage are $O(N)$.

Irrigation

Task Idea	Karolina Alexiou
Task Preparation	Petr Mitrichev
Description English	Petr Mitrichev
Description German	Charlotte Knierim
Description French	Théo von Düring
Solution	Petr Mitrichev

In this problem, the set of irrigation canals is represented as an undirected forest, with nodes corresponding to the junctions and edges corresponding to the actual canals. Some junctions are water sources. In one improvement, you can choose one junction as the *pivot junction*, destroy one of the canals connected to the pivot junction, and build a new canal connecting the pivot junction to any other junction. Your goal is to find such a sequence of improvements that touches each junction at most once, and after all improvements are made the number of junctions that are connected to at least one water source is as high as possible.

Subtask 1: Wadi Feiran (4 Points)

In this subtask there are at most three junctions and exactly one water source.

There are only three possible forests on three junctions: either there are no canals at all, one canal, or two canals. With zero canals, we cannot do anything. With two canals, everything is already connected to water so we do not need to do anything. With one canal, if one of its endpoints is a water source we cannot improve, and if not, we can replace this canal with any of the other two possible canals and get two junctions connected to water.

Subtask 2: Wadi Abbad (13 Points)

In this subtask there are at most seven junctions and exactly one water source.

With only seven junctions, we can do a brute force search of all possible sequences of actions to find one that maximizes the number of junctions connected to water, but of course it is logical to try solving the next subtask instead, and then use the same solution for this subtask, since the brute force will not generalize to other subtasks.

Subtask 3: Wadi El-Kharit (21 Points)

In this subtask there are at most 100 junctions and exactly one water source.

Let us examine a bit closer: what happens when we do an improvement? We split one tree into two parts, and then connect one of those parts to another tree. The other part remains disconnected. Since our goal is to connect everything to the tree that contains the water source, it is logical to try the following repeatedly: take any tree that is not connected to water and is not a single isolated junction, find any leaf in this tree, disconnect this leaf from the rest of the tree, and connect the rest of the tree to the tree containing the water source.

This process can end for two possible reasons. First, it might happen that at some point all trees that do not contain the water source are isolated junctions. In this case, we have clearly reached the maximum possible answer, since the number of connected components does not change, and therefore having all connected components without water have only one junction is the best we can do. We will call this situation a *good ending*. Second, it might happen that all junctions in the

tree containing the water source have already been touched, and therefore we cannot connect any new tree to it. We will call this situation a *bad ending*.

How do we prevent the bad ending from occurring? Define the *potential* as number of junctions in the tree containing the water source that still have not been touched. Initially, the potential is at least 1. If we take a tree with k junctions, and connect this tree except one leaf to the water tree as described above, then the potential increases by $k - 3$ (k new junctions are brought into consideration, but 3 additional junctions are touched). Therefore as long as we have a remaining waterless tree with at least 3 junctions, we can connect any such tree and the potential will not decrease, therefore the potential will stay positive and the bad ending will not occur. Therefore we will always reach a situation where all waterless trees have 1 or 2 junctions. Then we can keep connecting trees with 2 junctions, decreasing the potential by 1 every time. Either we run out of such trees, meaning that we have achieved a good ending, or the potential reaches 0, meaning that we have achieved the bad ending.

It turns out that even if we have achieved a bad ending in this way (first connect all trees with at least 3 junctions in any order, then connect the trees with 2 junctions until all junctions in the water tree have been touched), we have still maximized the number of junctions connected to water. This is intuitively clear, but here is a formal proof of this fact.

With every improvement, we connect a new tree that is originally without a water source to water. If we connect x such trees, we need to make at least x improvements, which requires to touch at least $3x$ junctions. If we reach a bad ending, it means that all junctions connected to water has been touched, so $3x$ is equal the size of the tree that was originally connected to water plus the total size of the x trees that were originally not connected to water. But since the remaining trees have size at most 2, and therefore we have connected the x biggest trees, for any sequences of $x + 1$ improvements we would touch $3(x + 1)$ junctions, so 3 more, but the total size of the connected trees would increase by at most 2, which means that we would have touched more junctions than there are available, which is a contradiction. This means that we will never be able to do $x + 1$ or more improvements that affect the tree containing the water source, and since we connect the x biggest trees, our answer is optimal.

Subtask 4: Mendesian (24 Points)

In this subtask there are at most 100 junctions and the water sources are not connected to each other, directly or indirectly.

It turns out that both the solution and the proof from the previous subtask can be applied here with a small modification: instead of connecting a waterless tree to *the* tree with the water source, we will connect a waterless tree to *any* tree with a water source in each improvement, and we stop when all such trees have all their junctions touched.

Subtask 5: Nile (38 Points)

In this subtask there are at most 100 junctions and no further restrictions. In particular, two water sources can belong to one tree.

First of all, do you see where does the above proof stop working in this case? It is still true that we will never be able to do $x + 1$ or more improvements that affect the trees containing the water sources, and it is still true that we connect the x biggest trees. The only issue is that when connecting a tree, we leave one of its leaves as a waterless isolated junction. In the previous subtasks, this was unavoidable; however in this subtask we can do better. If we split a tree containing at least two water sources in such a way that both parts have a water source, we can then connect one of those parts to a waterless tree, and after such an improvement no waterless isolated junction remains, so the answer becomes one better than with the previous approach. Let us call an improvement that starts by splitting a tree with at least two water sources into two

parts with water sources a *water split*.

Since each water split increases the answer by 1, we simply need to maximimize the number of water splits while still connecting x biggest trees. The maximum number of water splits can be found using a greedy approach inside a depth-first search. Our depth-first search function will return two boolean values for a subtree: whether the part of the subtree remaining after the water splits we have already done has at least one water source; and whether we have already done a water split using one of the canals adjacent to the root of the subtree.

As long as there is at least one water source outside a subtree, whenever the first of those booleans is true (we have a water source still pending in the subtree) and the second is false (we have not touched the root of the subtree), it is optimal to do a water split using the canal leading to this subtree (for a formal proof we can consider the situation where we do not use this canal, and obtain a solution with the same or better score by replacing another canal adjacent to one of the ends of this canal with this canal). After we do this for one of the children of a junction we touch the junction, and therefore cannot do the same for other children, so we need to merely propagate the first boolean upwards for them.

Here is the code for this depth-first search for more clarity:

```
def dfs(v, skip=-1):
  pending_this = is_source[v]
  consumed_this = False

  for u in adj[v]:
    if u == skip:
      continue
    pending, consumed = dfs(u, skip=v)
    if pending:
      if consumed or consumed_this:
        pending_this = True
      else:
        res.append((u, v))
        consumed_this = True

  return pending_this, consumed_this

res = []
pending, consumed = dfs(v)
if not pending:
  # The last water split left no water source reachable from
  # the root of the tree, so we need to undo it.
  res.pop()
```

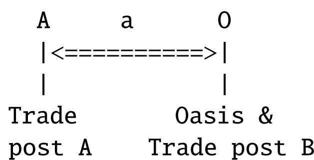
The variable `res` will contain the maximum possible set of water splits in the given tree.

Now we just need to do those splits first, connecting any waterless tree of size at least 2 on each split if it exists, otherwise connecting any isolated waterless junction, until either everything is connected to water, in which case we are done, or we run out of water splits, after which we can continue with the solution from the previous subtask. Connecting all waterless trees of size at least 2 in the beginning guarantees that the potential will be at least 1 when we are done with the water splits, and therefore the solution from the previous subtask will be able to proceed.

Donkey Caravan

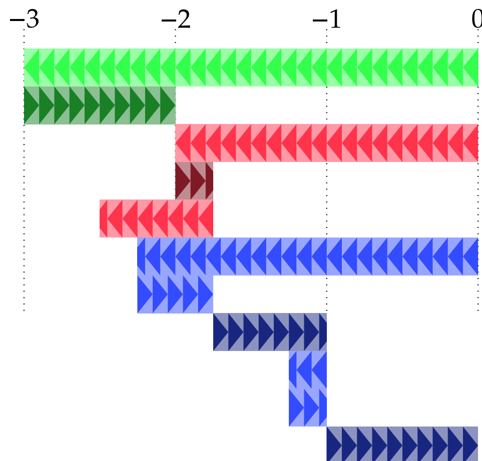
Task Idea	Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Johannes Kapfhammer
Description French	Théo von Düring
Solution	Johannes Kapfhammer

Subtask 1: Bahariya Oasis (10 points)



In this subtask all you had to do was to check whether it's possible to carry the chest from a tradepost on the left the oasis at 0. This section will analyze a slightly more general setting where we try to find some optimal assignment, as this will provide insights for the later subtasks.

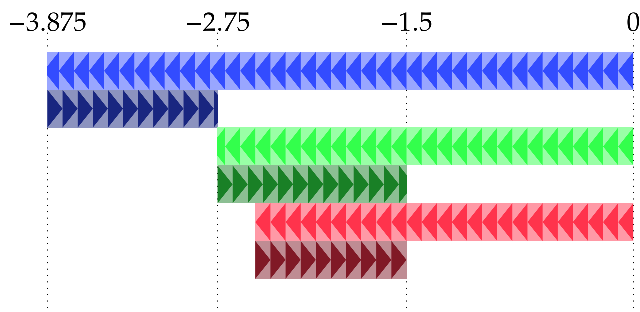
For example, if you have donkeys A, B, C with distances 3, 4 and 5, respectively, you can bring back something from distance 3.



donkey B moves left by 3.0.
donkey B moves right by 1.0.
donkey A moves left by 2.0.
donkey A moves right by 0.25.
donkey A moves left by 0.75.
donkey C moves left by 2.25.
donkey C moves right by 0.5.
donkey C moves right by 0.75.
donkey C moves left by 0.25.
donkey C moves right by 0.25.
donkey C moves right by 1.0.

However, doing it like this is not optimal.

The furthest one can carry with those three donkeys is distance 3.875 like this:



donkey C moves left by 3.875.
donkey C moves right by 1.125.
donkey B moves left by 2.75.
donkey B moves right by 1.25.
donkey A moves left by 2.5.
donkey A moves right by 1.0.

By looking at some examples one can come up with the following claim:

Claim 1 *There is always an optimal solution where every donkey will first travel left without the chest, then right with the chest until either it has exhausted its distance or has reached the point 0.*

Or visually, the path of a single donkey will always look like this:

donkey C moves left by 3.0.
donkey C moves right by 1.0.

We rule out any other shapes by a proof of contradiction. The proof is built by combining the statements below:

There always exists an optimal solution in which we:

- *(i):* Never carry the chest left.
- *(ii):* Never move after dropping off the chest.
- *(iii):* Never drop off a chest until the donkey is out of moves.
- *(iv):* Any donkey picks up the chest at most once.
- *(v):* Never move right without a chest.

An easy way to show those statements is to use proof by contradiction. See also `https://soi.ch/wiki/theoretical-greedy/` for an introduction on this proving scheme.

Claim *(i)*: Assume there is only an optimal solution if we violate (i). Let's take this solution and modify it slightly: instead of moving left with the chest, we drop it off, and then picking it up the last time we pass by again (this always happens because we need to carry it to 0 eventually). Note that since the previous solution was valid, this one is too, and it adheres property (i). So the claim was wrong that there is no optimal solution violating (i), which leads to a contradiction. Therefore (i) must be true.

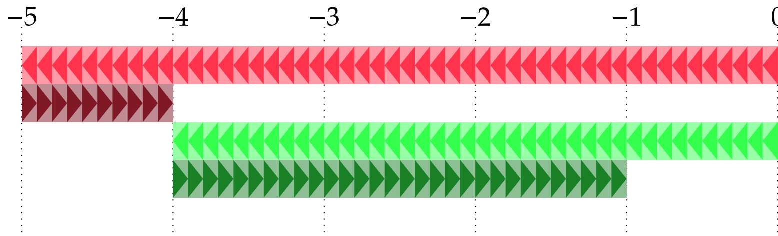
The same proof sketch works for the other claims. We just state the main arguments here:

Claim *(ii)* If a donkey moves at the end without chest, it could just stop earlier. Claim *(iii)* If a donkey stops before its full distance, it could also move the chest further left. Claim *(iv)* If a donkey picks up the chest twice, it could also carry it all the time in between the first and second time. Claim *(v)* Once we have the chest we never drop it off until the end, so if we move right we would have done it before picking up the chest. In that case there is a sequence "left,right" without chest in the path of the donkey. We could just remove this stretch from the donkey's path.

Combing this: We can only move left without chest, and move right with chest. And once we have the chest we carry it until we're done. Therefore the sequence can only be moving left moves without chest followed by moving right with chest. And once we have the chest we move all the way. Note that this is exactly the claim.

Claim 2 *We can always use the donkeys ordered decreasingly in distance (largest distance first).*

Again we do a proof by contradiction. From the set of all possible solutions, select one where the donkeys are ordered as long as possible. And assume they are not fully ordered, so at some point we use two donkeys A and B after each other and $d_A < d_B$, e.g.:



donkey A moves left by 5.0.
donkey A moves right by 1.0.
donkey B moves left by 4.0.
donkey B moves right by 3.0.

If we start at distance $-x$ then donkey A carries the chest from $-x$ to $-y = -x + (d_A - x)$ and donkey B carries it from $-y$ to $-y + (d_B - y)$, or, in total, we carry it to

$$-x + (d_A - x) + (d_B + (-x) + (d_A - x)) = -x + (d_A - x) + (d_B + d_A) = 2 \cdot d_A + d_B - 2 \cdot x$$

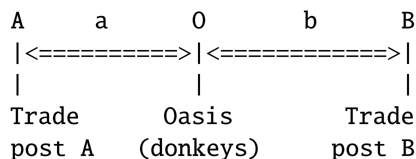
If we would do it the other way around, we would carry it to $2 \cdot d_B + d_A - 2 \cdot x$. Since $d_A < d_B$ the second number is larger, meaning the chest is further right (closer to 0). Thus it would be benefitial to swap, which contradicts the original assumption. Therefore we have proven Claim 2.

Thus we can code a simple greedy that sorts the distances and uses the donkeys one by one:

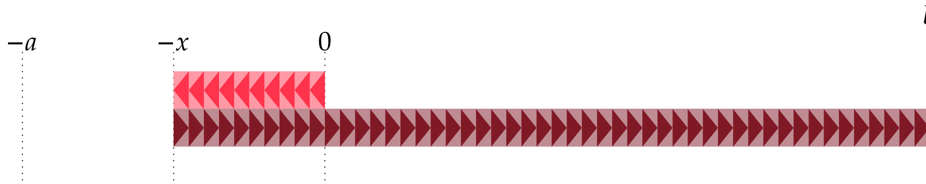
```
#include <bits/stdc++.h>
using namespace std;

bool solve(vector<int> v, int64_t a) {
  sort(v.rbegin(), v.rend());
  int n = v.size();
  for (int j = 0; j < n; ++j) {
    a -= max<int64_t>(0, v[j] - a);
    if (a <= 0)
      return true;
  }
  return false;
}

int main() {
  cin.exceptions(ios::badbit | ios::eofbit | ios::failbit);
  int t; cin >> t;
  for (int i=0; i<t; ++i) {
    int n, a, b; cin >> n >> a >> b;
    vector<int> v; copy_n(istream_iterator<int>(cin), n, back_inserter(v));
    cout << "Case #" << i << ": " << (solve(v, a) ? "YES" : "NO") << '\n';
  }
}
```

Subtask 2: Dakhla Oasis (10 points)

In this subtask we have to bring the chest from A to B.


```
A      a       0        b       B
|<==========>|<============>|
|             |              |
Trade       Oasis          Trade
post A    (donkeys)       post B
```

There is a quadratic solution with the following insight: Exactly one donkey i with distance d_i moves $x = \frac{b-d_i}{2}$ steps to the left and then right all the way to b (for a total distance of $2 \cdot \frac{b-d_i}{2} + b = d_i$).



Claim 3 *If there is a solution, there is always one where the distance from 0 to b is traveled by exactly one donkey.*

Obviously it is traveled by at least one donkey, otherwise the chest can't get to B. So consider the donkey that drops off the chest at B. We could just let this one pick the chest up at 0 and carry it B alone, and leave all other donkeys left of 0.

So which donkey i do we want to use? It can be proven that this is the donkey with the largest i (with the smallest distance) having $d_i \geq b$ (that can reach B), which we could compute via binary search.

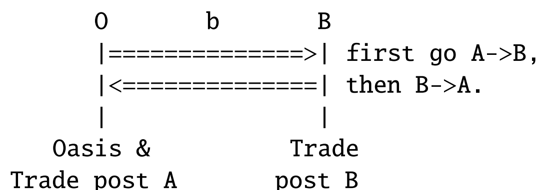
However since $n \leq 100$ in this subtask, one could also try out all the i's.

Once we assigned donkey i to go all the way to the right, the remaining problem is very similar to the first subtask, and we can prove that again it's optimal to go through all the donkeys in decreasing size.

```
1  bool solve(vector<int> v, int a, int b) {
2    sort(v.rbegin(), v.rend());
3    int n = v.size();
4    if (v[0] >= 2*a + b)
5      return true;
6    for (int i = 0; i < n; ++i) {
7      if (v[i] < b) break;
8      int carry = max(0, v[i] - b)/2;
9      int rest = a;
10     for (int j = 0; j < n; ++j) {
11       if (i == j) continue;
12       rest -= max(0, v[j] - rest);
13       if (rest <= carry)
14         return true;
15     }
16   }
17   return false;
18 }
```

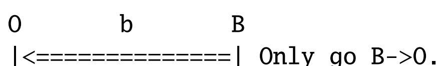
Subtask 3: Karga Oasis (20 points)

Starting from this subtask we have to bring the chest from A to B and back. Additionally, we don't just have to output YES or NO, but instead compute the largest possible b.



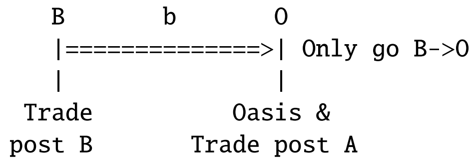
There are two key insights.

First, if we manage to bring the chest from B to A, then some donkey must have went to B beforehand, anyways, and thus the task is equivalent to solving it without bringing the chest from A to B:



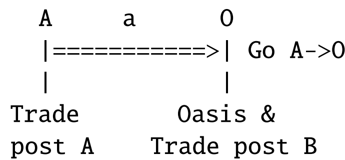
```
     |               |
   Oasis &         Trade
Trade post A      post B
```

If we mirror the image it will look like this:



```
      B       b       O
      |==============>| Only go B->O
      |               |
    Trade          Oasis &
   post B        Trade post A
```

And this very much ressembles subtask 1 which was the same up to swapping the name of A and B.



```
  A     a      O
  |===========>| Go A->O
  |            |
Trade        Oasis &
post A     Trade post B
```

Thus we just have to solve subtask A with the additional constraint of finding the largest a.

Since we have a predicate that tells us whether any givern a works, we can apply binary search:

```cpp
bool check(vector<int> const& v, double a) {
  int n = v.size();
  for (int j = 0; j < n; ++j) {
    a -= max(0.0, v[j] - a);
  }
  return a <= 0;
}

double solve(vector<int> v) {
  sort(v.rbegin(), v.rend());
  double l=0, r=accumulate(v.begin(), v.end(), 0) / 2.0;
  while (r - l > 1e-9) {
    double m = (l + r)/2;
    if (check(v, m))
      l = m;
    else
      r = m;
  }
  return l;
}
```

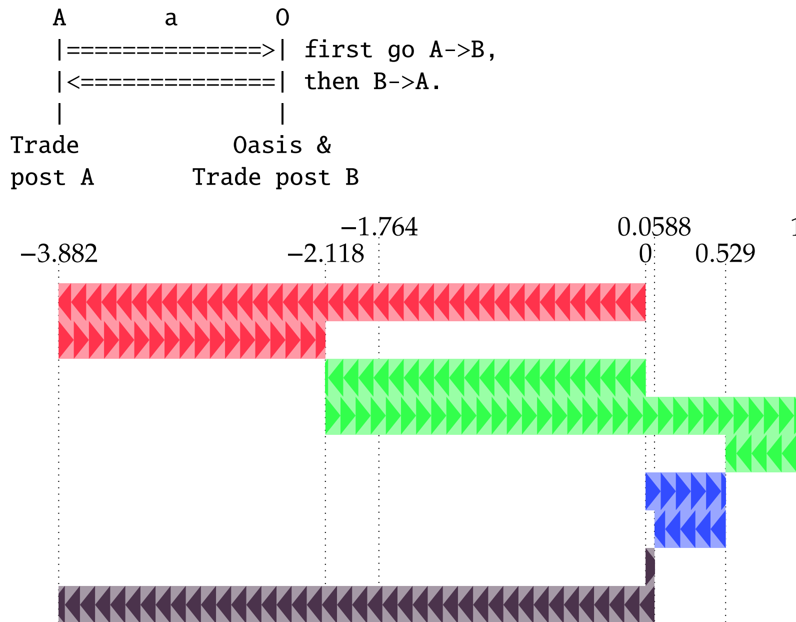
However, one further insight one could have is that we don't even need the binary search! We can just work backwards:

1. The donkey with the smallest distance d_{n-1} can bring the chest from $-d_{n-1}/2$ to 0.
2. The donkey with the second smallest distance d_{n-2} can bring the chest from $-d_0/2 - \frac{d_{n-1}/2 - d_{n-2}}{2}$ to $d_0/2$.
3. etc.

Which leads to very short solution:

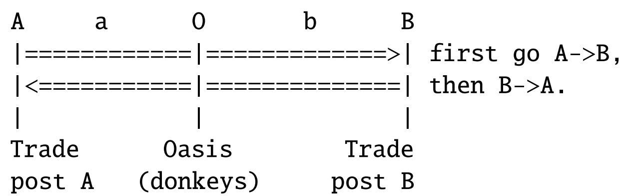
```python
def solve_sub3(xs):
    a = 0
    for x in sorted(xs):
        a += max(0, (x - a)/2)
    return a
```

Subtask 4: Karga Oasis (35 points)



```
   A       a       O
   |==============>| first go A->B,
   |<==============| then B->A.
   |               |
Trade           Oasis &
post A       Trade post B
```


Subtask 5: El Fashir Oasis (25 points)



```
A     a      O       b      B
|============|=============>| first go A->B,
|<===========|==============| then B->A.
|            |              |
Trade      Oasis          Trade
post A   (donkeys)       post B
```