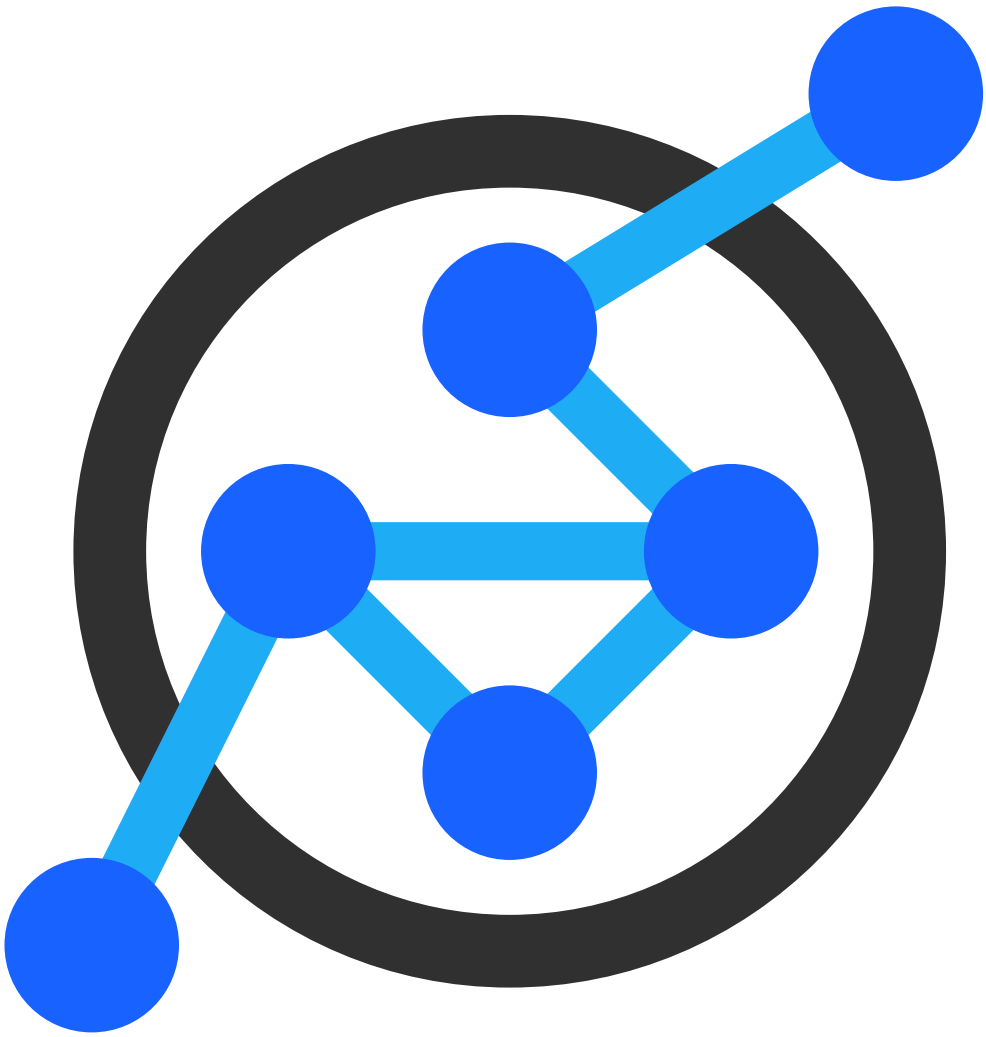
First Round SOI 2021

Solution Booklet



Swiss Olympiad in Informatics

15 September – 30 November 2020

Stairracing

Task Idea	Bibin Muttappillil
Task Preparation	Timon Gehr
Description English	Timon Gehr
Description German	Johannes Kapfhammer
Description French	Florian Gatignon
Solution	Timon Gehr
Correction	Jan Schär

We are given two arrays $a_0, \ldots, a_{N-1}$ and $b_0, \ldots, b_{N-1}$ describing the heights of skyscrapers on both sides of a road. Our goal is to find a longest possible racing track: we want to maximize the expression $a_i + \left|i - j\right| + b_j$ over indices i and j.

Subtask 1: $N = 1$

If $N = 1$, there is only one candidate for i and j, namely $i = 0$ and $j = 0$. If we evaluate our objective function for those values of i and j, we obtain $a_0 + b_0$. I.e., our task is to add two numbers:

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

int solve(int n, const vector<int> &a, const vector<int> &b){
    return a[0] + b[0];
}

int main(){ // (i/o, the same for all subtasks)
    int t;
    cin >> t;
    for(int i = 1; i <= t; i++){
        int n;
        cin >> n;
        vector<int> a(n), b(n);
        for(int i = 0; i < n; i++) {
            cin >> a[i];
        }
        for(int j = 0; j < n; j++) {
            cin >> a[k];
        }
        cout << "Case #" << i << ": " << solve(n, a, b) << '\n';
    }
}
```

Subtask 2: $N = 2$

If $N = 2$, there are four possible racing tracks and we compute the longest one:

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

int solve(int n, const vector<int> &a, const vector<int> &b){
    return max({a[0] + b[0], a[0] + 1 + b[1],
                a[1] + b[1], a[1] + 1 + b[0]});
}
```

Subtask 3: $N \leq 10^3$

If $N \leq 10^3$, there are at most 10^6 possible different racing tracks. Therefore, we can again just iterate through all of them and compute the longest one:

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

int solve(int n, const vector<int> &a, const vector<int> &b){
    int longest = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            longest = max(longest, a[i] + abs(i-j) + b[j]);
        }
    }
    return longest;
}
```

Subtask 4: $N \leq 10^5$

Now, there can be inputs for which there are up to 10^{10} possible different racing tracks. To solve such instances efficiently, we need to be more clever than before[1]: we no longer want to explicitly consider all possible racing tracks. To this end, we first rewrite our objective function so that it considers the cases $i \leq j$ and $j \leq i$ separately (this is a common trick we use to eliminate absolute values from an expression):

$$\max_{1 \leq i,j < N} \left(a_i + \left|i - j\right| + b_j\right) = \max\left(\max_{1 \leq i \leq j < N} \left(a_i - i + j + b_j\right), \max_{1 \leq j \leq i < N} \left(a_i + i - j + b_j\right)\right)$$

We will optimize our two partial objectives using dynamic programming. For instance, we have

$$\max_{1 \leq i \leq j < N} \left(a_i - i + j + b_j\right) = \max_{1 \leq j < N} \left(c_j + b_j\right), \text{ where } c_j := \max_{1 \leq i \leq j} \left(a_i - i + j\right).$$

We now derive a recurrence relation for the values c_j by considering the cases $i \leq j - 1$ and $i = j$ separately:

$$\begin{aligned} c_j = \max_{1 \leq i \leq j} \left(a_i - i + j\right) &= \max\left(\max_{1 \leq i \leq j-1} \left(a_i - i + j\right), a_j - j + j\right) \\ &= \max\left(\max_{1 \leq i \leq j-1} \left(a_i - i + (j-1)\right) + 1, a_j\right) = \max(c_{j-1} + 1, a_j). \end{aligned}$$

For the case $j \leq i$, we can compute values $d_i := \max_{1 \leq j \leq i} \left(i - j + b_j\right)$ in the same way. This leads to the following linear-time solution:

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

int solve(int n, const vector<int> &a, const vector<int> &b){
    int longest = 0;
    int c = -1, d = -1;
    for(int k = 0; k < n; k++) {
        c = max(c + 1, a[k]);
        d = max(d + 1, b[k]);
        longest = max({longest, c + b[k], a[k] + d});
    }
    return longest;
}
```

[1] It might however have been possible to solve this subtask inefficiently yet below the time limit by parallelizing our brute-force computation from before on a sufficiently powerful machine. This will no longer be possible in further rounds because the programs will be evaluated on an automated judging system instead of your own computer.

Secret Code

Task Idea	Johannes Kapfhammer
Task Preparation	Christopher Burckhardt
Description English	Christopher Burckhardt
Description German	Luc Haller
Description French	Florian Gatignon
Solution	Christopher Burckhardt
Correction	TODO

This task revolved around mouse Stofl inventing his own encryption protocol. However he needed help determining if his encrypted words could be decoded simply. A decoding was deemed to be simple if each letter from the encrypted word could be replaced with another letter to retrieve the decrypted cleartext.

Subtask 1: Two Letters (25 Points)

Subtask 2: Lots of Letters (25 Points)

In both of these subtasks you were given two words, the encrypted and decrypted word. You had to determine if it was possible to decode the encrypted word to the decrypted word.

This is not possible if and only if a letter from the first/encrypted word is at the same position as multiple different letters in the second/decrypted word. As this being the case implies a letter being replaced with multiple different letters, which is not allowed in a simple decoding.

Otherwise we may simply replace each letter in the first word, with the letter that appears at the same position in the second word. To store this translation table we use a map, a map allows us to store what letter each letter in the first word should be replaced with.
Since each letter must only be considered once the time complexity is $O(L)$ where L is the length of each word and the space complexity is also $O(L)$.

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

using namespace std;

void solve(){
	string enc, dec;
	cin >> enc >> dec;
	map<char, char> dic;
	for(int i = 'a'; i<= 'z'; i++){
		dic[i] = '#';
	}
	for(int i = 0; i< enc.size(); i++){
		if(dic[enc[i]] != '#' && dic[enc[i]] != dec[i]){
			cout << "No" << '\n';
			return;
		}
		dic[enc[i]] = dec[i];
	}
	cout << "Yes" << '\n';
	for(auto[key, val] : dic){
		if(val == '#'){
			cout << 'a';
		}else{
			cout << val;
		}
	}
	cout << '\n';
```

```
}

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

        int t;
        cin >> t;
        for(int i = 0; i< t; i++){
                cout << "Case #" << i<<": ";
                solve();
        }
}
```

Subtask 3: Two Mice (25 Points)

Subtask 4: Lots of Mice (25 Points)

In the last two subtasks you were tasked with finding a word that as many encryptions as possible could be decrypted to, additionally this word had to have as many unique letters as possible.

The second requirement was what made these subtasks difficult. Since every encryption could be decoded to a string of 'a', by simply replacing every letter with an 'a'. To satisfy both requirements, two positions in the decoded word have the same letter if and only if, the same letter is present at these positions in one of the encrypted words. Using this insight we can reformulate the problem as a graph, where each position is a node and two nodes are connected if the same letter appears at these two positions in one of the encrypted words. Now if two nodes are connected they must share the same letter. Because of this any node reachable from a given node must share the same letter. We now look at each position of the decrypted word, if a position has not been assigned a letter yet, we assign the next letter of the alphabet to this position and then assign this letter to all other reachable positions using DFS or BFS.
Once we have found a valid decoding using the maximum amount of unique letters, we may use the algorithm in subtask 2 to genarate a translation table for each encryption.
The final time complexity of this algorithm is $O(N \cdot L)$, where N is the number of words and L is the length of each word, since we only add two edges per letter of the input and DFS is linear in the number of edges. The same argument applies to space complexity which is also $O(N \cdot L)$.

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

using namespace std;

void dfs(int u, vector<char>& dec, vector<vector<int>>& g){
        for(auto v : g[u]){
                if(dec[v] != '#')continue;
                dec[v] = dec[u];
                dfs(v, dec, g);
        }
}

void solve(){
        int N, L;
        cin >> N >> L;
        vector<string> words;
        for(int i = 0; i< N; i++){
                string tmp; cin >> tmp;
                words.push_back(tmp);
        }
        vector<vector<int>> g(L, vector<int>());
        for(auto enc : words){
                map<char, int> last_pos;
                for(int i = 'a'; i<= 'z'; i++)last_pos[i] = -1;
                for(int i = 0; i< enc.size(); i++){
```

```
                        char c = enc[i];
                        if(last_pos[c] != -1){
                                g[last_pos[c]].push_back(i);
                                g[i].push_back(last_pos[c]);
                        }
                        last_pos[c] = i;
                }
        }
        vector<char> dec(L, '#');
        int counter = 0;
        for(int i = 0; i< L; i++){
                if(dec[i] == '#'){
                        dec[i] = 'a' + counter;
                        counter++;
                        dfs(i, dec, g);
                }
        }
        cout << counter << ' ';
        for(auto c : dec)cout << c;
        cout << '\n';
        for(auto enc : words){
                map<char, char> dic;
                for(int i = 'a'; i<= 'z'; i++)dic[i] = '#';
                for(int i = 0; i< enc.size(); i++)dic[enc[i]] = dec[i];
                for(auto[key, val] : dic){
                        if(val == '#'){
                                cout << 'a';
                        }else{
                                cout << val;
                        }
                }
                cout << '\n';
        }
}

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

        int t;
        cin >> t;
        for(int i = 0; i< t; i++){
                cout << "Case #" << i<<": ";
                solve();
        }
}
```

Bamboo

Task Idea	Johannes Kapfhammer
Task Preparation	Fabian Haller
Description English	Fabian Haller
Description German	Fabian Haller
Description French	Florian Gatignon
Solution	Michal Svagerka
Correction	TODO

Subtask 1: Three Bamboos (15 points)

Consider cutting bamboos $i, i+1, \ldots, j-1$ to height h. In an optimal solution, h equals to the desired height of one of the bamboos, that is, there exists $i \leq k < j$ such that $h_k = h$.

We can prove this by contradiction. Assume we cut bamboos $i, i+1, \ldots, j-1$ to height h and for all $i \leq k < j$ holds $h_k < h$. For each k there are two cases:

- Height of k-th bamboo was smaller than h before this cut and this bamboo is unaffected.
- Height of k-th bamboo was at least h, but the desired height $h_k < h$. This means that in some point in the future we will cut the k-th bamboo again.

In both cases removing the cut does not affect the end result and leads to a solution using one fewer cut.

As a result, for each cut we pick one of the contiguous subsets of bamboos (there are 6 of them) and one of their heights (there are at most 3 of them). Also, note that it is always possible to solve the problem using three cuts: $(0, 1, h_0), (1, 2, h_1), (2, 3, h_2)$. Is is thus sufficient to try all possibilities for two cuts – there are at most $(3 \cdot 6)^2 = 324$ of them.

```cpp
#include <iostream>
#include <vector>

using namespace std;
constexpr int INF = 1e9;

int min_cuts(const vector<int> &wanted, const vector<int> &current, int cuts) {
    int N = wanted.size();
    if (wanted == current) return cuts;
    if (cuts == N-1) return N;

    int ans = N;
    for (int i = 0; i < N; ++i) {
        for (int j = i+1; j <= N; ++j) {
            for (int k = i; k < j; ++k) {
                vector<int> changed = current;
                for (int l = i; l < j; ++l) changed[l] = min(changed[l], wanted[k]);
                ans = min(ans, min_cuts(wanted, changed, cuts+1));
            }
        }
    }
    return ans;
}

int main() {
    int T; cin >> T;
    for (int t = 0; t < T; ++t) {
        int N; cin >> N;
        vector<int> H(N);
        for (int &h: H) cin >> h;
```

```
31          cout << "Case #" << t << ": "min_cuts(H, vector<int>(N, INF), 0) << '\n';
32      }
33  }
```

Subtask 2: Two Heights (25 points)

First consider the case where $h_i = 1$ for all $0 \leq i < N$. This can be simply solved using the cut $(0, N, 1)$, that is, cutting everything to height 1.

From now on, we assume that there exists i such that $h_i = 2$. It is evident that we can start with the cut $(0, N, 2)$, that is, cutting everything to height 2. This is because we need at least a single cut to height 2 and all such cutsve all cuts can be moved to happen earlier than all cuts to height 1. Then we merge all 2 cuts to a single one that spans the whole range. These changes cannot make the solution worse.

Now, we need to perform some cuts to height 1. It is clear that it is optimal to use cuts that are as wide as possible, that is, the whole range between two 2s, or either ends of the array. We can simply count the number of ranges of consecutive ones by counting all ones that have either 2 or the end of the array to the left of them.

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main() {
7      int T; cin >> T;
8      for (int t = 0; t < T; ++t) {
9          int N; cin >> N;
10         vector<int> H(N);
11         for (int &h: H) cin >> h;
12
13         if (count(H.begin(),H.end(),2) == 0) {
14             cout << "Case #" << t << ": 1\n";
15         } else {
16             int answer = 1;
17             for (int i = 0; i < N; ++i) {
18                 if (H[i] == 1 && (i == 0 || H[i-1] == 2)) {
19                     ++answer;
20                 }
21             }
22             cout << "Case #" << t << ": " << answer << '\n';
23         }
24     }
25 }
```

Subtask 3: Small Forest (25 points)

Let's generalize the idea from the previous subtask. What we did in each step was:

- Select (one of) the maximal ranges in which no bamboo is at correct height yet.
- Cut all bamboos to the maximum of the desired heights among those in the range.

Why does this work? First, observe that we can always rearrange each solution such that heights of all cuts are non-increasing. Secondly, we cannot lose anything by cutting the largest available range.

This immediately yields a solution for this subtask: find all maximums in the array and cut the whole array to this maximum height. This splits the array into some amount of smaller subarrays (possibly empty), and we solve the same problem recursively on these subarrays.

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
```

```cpp

int min_cuts(const vector<int>&H, int from, int to) {
    // empty array
    if (from >= to) return 0;

    int maximum = *max_element(H.begin() + from, H.begin() + to);
    int left = from;
    int answer = 1;
    for (int i = from; i < to; ++i) {
        if (H[i] == maximum) {
            answer += min_cuts(H, left, i);
            left = i+1;
        }
    }
    answer += min_cuts(H, left, to);
    return answer;
}

int main() {
    int T; cin >> T;
    for (int t = 0; t < T; ++t) {
        int N; cin >> N;
        vector<int> H(N);
        for (int &h: H) cin >> h;

        cout << "Case #" << t << ": " << min_cuts(H, 0, N) << '\n';
    }
}
```

To analyze the complexity of the above solution, note that when `min_cuts` is called on a subarray of length k, the total length of all subarrays on which `min_cuts` is invoked recursively is at most $k-1$, as at least one element, namely the maximum, isn't used in any of the recursive calls. To eliminate this maximum element, we do $O(k)$ work. This implies that the time complexity of the whole solution is $O(N^2)$.

Subtask 4: Large Forest (35 points)

The expensive part of the solution from the previous subtask is the linear processing of the subarray to find the value of maximum and all its occurrences. There are at least two ways of speeding this up.

The first solution doesn't need any additional observation. Recall that we can perform all cutting operations in a non-increasing order of height. Hence, we can use a single sort to find the order in which the cuts will be performed.

We will maintain a set of intervals of the original array that still need cutting. Initially, this set will contain just the interval $[0, N)$. When processing the cuts of height h, find the intervals to which all the bamboos with height h belong. The number of such distinct intervals equals the number of cuts of height h to be performed. Following this we simply split the intervals into smaller ones.

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <set>

using namespace std;

int main() {
    int T; cin >> T;
    for (int t = 0; t < T; ++t) {
        int N; cin >> N;
        vector<int> H(N);
        for (int &h: H) cin >> h;

        map<int, vector<int>> byHeight;
```

```
16          for (int i = 0; i < N; ++i) byHeight[H[i]].push_back(i);
17
18          // We store the right point of each interval to make updating easy.
19          set<int> intervals{N};
20          int answer = 0;
21          for (auto it = byHeight.rbegin(); it != byHeight.rend(); ++it) {
22              int current_interval = -1;
23              for (int pos: it->second) {
24                  // Find the interval to which index 'pos' belongs.
25                  // The right point of this interval is going to be the
26                  // smallest right point larger than 'pos'.
27                  auto interval = intervals.upper_bound(pos);
28
29                  // The bamboos of given height are processed in increasing
30                  // order of their indices. As a result, we can compute their
31                  // count by simply calculating the number of bamboos that
32                  // belong to a different interval than the previous bamboo.
33                  if (current_interval != *interval) {
34                      current_interval = *interval;
35                      answer++;
36                  }
37
38                  // Cutting a bamboo at 'pos' creates a new interval with
39                  // right-most point at 'pos'. Again, since bamboos of same
40                  // height are processed from left to right, we can add this
41                  // right away, and it won't negatively affect future queries.
42                  intervals.insert(pos);
43              }
44          }
45
46          cout << "Case #" << t << ": " << answer << '\n';
47      }
48  }
```

The processing of each bamboo costs $O(\log N)$ (map insert, set query and set insert). The total complexity is $O(N \log N)$.

There is another solution. Let's say that each cut has cost 1. When we perform cut on interval $[a, b)$ to height h, this cost is charged to one of the bamboos on the interval $[a, b)$ that have desired height h. Recall that none of the bamboos in this interval have larger desired height, in other words, h is maximum on interval $[a, b)$. Without loss of generality, we will charge the cost of the operation to the left-most maximum on an interval.

In order to determine which elements are the left-most maxima, we look at maximums of certain subarrays. Consider element at position k and compute the suffix maximums $s_i = max_{j=i}^{k-1} h_j$, that is, maximums of all subarrays that end just to the right of k. Note that these maximums are non-increasing when i is increasing.

The following is true: k the left-most maximum, if and only if the set $\{s_i\}_{i=1}^{k-1}$ does not contain h_k as a value. Let's prove it.

Assume that $h_k \in \{s_i\}_{i=1}^{k-1}$. Let j be the largest index for which this is true. Then we clearly have $h_j = h_k$ and $h_l < h_j$ for all $j < l < k$ (otherwise $s_j \neq h_j$ or j is not maximal). This means that a cut containing both bamboos j and k to height h_k can be performed, as it doesn't cut any bamboo in-between to height smaller than it should.

For the other direction, assume that h_k is the left-most maximum. There are two cases. Either there is no $j < k$ for which $h_j \geq h_k$ and we are done. Otherwise, let j be the largest of them. We know that the suffix maxima s_i for $i > j$ are smaller, because all elements in (j, k) are smaller. Since k is the left-most maximum, we know that $h_j > h_k$. As a result $s_i > h_k$ for $i \leq j$ and we are done.

To implement this solution, we process the array from left to right, and keep the set of suffix maximums. Note that we maintain just the set, without possible duplicates. How does this set change when we add new element h_k? Clearly, all elements smaller than h_k will cease to be suffix maximums, and h_k will become a new suffix maximum (if it wasn't already). This can be implemented using a set in a straightforward fashion and leads to a $O(N \log N)$ solution.

We can, however, do even better. Recall that suffix maxima occur in non-increasing order, and we only ever remove some of the smallest ones, and then add a new one that is the smallet of the remaining. Hence, we can store all of the maxima in a stack in a sorted order.

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

int main() {
    int T; cin >> T;
    for (int t = 0; t < T; ++t) {
        int N; cin >> N;
        vector<int> H(N);
        for (int &h: H) cin >> h;

        stack<int> maxima;
        int answer = 0;
        for (int i = 0; i < N; i++) {
            while (!maxima.empty() && maxima.top() < H[i]) maxima.pop();
            if (maxima.empty() || maxima.top() > H[i]) {
                answer++;
                maxima.push(H[i]);
            }
        }
        cout << "Case #" << t << ": "<< answer << endl;
    }
}
```

This solution has time complexity $O(N)$. This is because we only add $O(N)$ elements to stack, hence only $O(N)$ elements can be also removed from it.

Battery Fix

Task Idea	Bibin Muttappillil
Task Preparation	Bibin Muttappillil
Description English	Bibin Muttappillil
Description German	Jan Schär
Description French	Elias Boschung
Solution	Daniel Rutschmann, Johannes Kapfhammer, Bibin Muttappillil
Correction	Bibin Muttappillil, Johannes Kapfhammer, Tobias Feigenwinter

We are given a permutation p of $\{0, 1, ..., 2 \cdot N - 1\}$ lying on a circle. Those are our pole levels p_i. A wire connecting the pole levels p_i and p_j has the power $|p_i - p_j|$. Our goal is to find a perfect matching [1] of maximum power [2], where the wires don't overlap.

Observation: Overlapping

Let's denote that a wire (i, j) connects the poles at **positions** i and j where w.l.o.g [3] $i < j$. Two wires (i, j) and (i', j') (w.l.o.g. $i < i'$) are overlapping if and only if

$$i' < j < j'$$

Intuitively this means that two wires overlap if and only if one of them starts in the range of the other and ends up outside. So $i < j < i' < j'$ and $i < i' < j' < j$ wouldn't be overlapping.

Subtask 2: Small Battery (10 points)

In this subtask N is small enough to test every matching and then take the maximum of the ones that don't overlap.

To see that we can calculate the number of matchings: For the first pole we have $2N - 1$ possible end poles. If we remove this wire we are left with a graph of $2N - 2$ poles ans we can apply the same principle again to find the number of matching in the smaller graph. Continuing this inductive argument we can see that we need to test at most $(2N - 1) \cdot (2N - 3) \cdot (2N - 5)...4 \cdot 2$ matchings.

The correctness follows from the fact that we will test literally every possible matching and therefor we will eventually encounter one of the optimal ones.

This approach can be implemented in $O(complicated)$ runtime and $O(N)$ memory (we only need to keep track of the matching of the poles).

Solution code will be added soon

Observation: Maximum Power

Let's call the poles with levels grater or equal to N 'greater' poles and the one with levels smaller than N 'smaller' poles. One crucial observation is that the optimal can only be achieved by summing up all the levels of the 'greater' poles and subtracting the ones from 'smaller' poles. This means that we want to always connect a 'greater' pole with a 'smaller' pole.

To show this we will do a proof of contradiction: Assume that we have an optimal solution where not all wires connect a 'greater' and a 'smaller' pole. Since the number of 'greater' poles and 'smaller' poles are equal it means that if a wire connects two 'greater' poles, let's call them g and h, then another wire has to connect two 'smaller' poles, let's call them s and t. The same

[1] perfect matching means that every pole is connected to exactly one other pole
[2] with maximum power we mean the sum of the powers should be as large as possible
[3] read the example on wikipedia.org if you don't know w.l.o.g. `https://en.wikipedia.org/wiki/Without_loss_of_generality`

argument works the other ways around. Assume w.l.o.g. $l_g > l_h > l_s > l_t$. If we now would reconnect the wires to $g - -s$ and $h - -t$ the power change would be

$$-(|l_g - l_h| + |l_s - l_t|) + (|l_g - l_s| + |l_h - l_t|) = -l_g + l_h - l_s + l_t + l_g - l_s + l_h - l_t = 2l_h - 2l_s > 0$$

(subtracting the powers of the old wire and adding the powers of the new ones). By reconnecting the wires we achieved a higher power which contradicts the assumption that the solution was optimal. Therefore every wire has to connect a 'greater' and a 'smaller' pole.

Observation: Reduction to a Binary Problem

With the observation above we can reduce our original problem to a simpler one: Convert all the 'greater' poles to 1 and all the 'smaller' poles to 0. Now the question is just if it is possible to connect every 1 with a 0 without overlapping and how to find such a matching. As it turns out it is actually always possible which means we don't need to look for a solution which doesn't achieve the maximum possible power.

Subtask 1: Ordered (10 points)

In this subtask the permutation is ordered meaning that $p_i = i$ for every i. With the above observation we can reduce this problem to one where $p_i = 0$ for $i < N$ and $p_i = 1$ for $i \geq N$ with the goal to find a non-overlapping matching. A solution is to always connect the pole i with the pole $(2N - 1) - i$. [4]

Correctness. This is a perfect matching as every pole is connected to exactly one other. Every pair of wires $(i, 2N - 1 - i)$ and $(j, 2N - 1 - j)$ (w.l.o.g. $i < j$) aren't overlapping as they fulfill the inequality $i < j < 2N - 1 - j < 2N - 1 - i$ (see Observation: Overlapping). And from the observation of maximum power we also know that no other matching has a higher power. Therefore we have found a correct solution.

This approach can be implemented in $O(N)$ runtime, as we only need one loop from 0 to $N - 1$, and $O(1)$ memory, as we only have constant size variables.

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

using namespace std;

#define int long long

int N;
vector<int> potentials;

void solve(){
        for(int i = 0; i < N; i++){
                cout << i << " " << 2*N - 1 - i << "\n";
        }
}

signed main() {

    ios_base::sync_with_stdio(false);
    cin.tie(0);

    cin >> N;
    potentials = vector<int>(2*N);
    for(int i = 0; i < 2*N; i++){
            cin >> potentials[i];
    }

    solve();
}
```

[4] As it turns out this is the only solution. The proof for this fact is left as an exercise for the reader ;)

Observation: Good Wires

Notice that if we connect two poles i and j with a wire we create two sections: the poles starting from $i+1\%(2N-1)$ walking clockwise until $j-1\%(2N-1)$ and $j+1\%(2N-1)$ to $i-1\%(2N-1)$ (the $\%(2N-1)$ is here in case we walk over $2N-1$ to 0). We can look at each section again as a smaller 'circle' as the actual form of the figure doesn't matter, as long as they have the same property for overlapping wires. So connecting a wire splits the binary problem into two smaller problems, which we can try to solve recursively.

It could be that if we connect a specific wire that we can't find a solution to the binary problem without reconnecting or overlapping said wire. Let's call those bad wires. If it is still possible to find a solution we will call it a good wire.

The cool thing is that good wires fulfill a specific property: A wire is good if and only if each of the resulting sections has equal numbers of 1 and 0.

If we don't have an equal number of 1 and 0 then it is not possible to solve it, as every wire connects a 1 with a 0 and thus the need to be available in equal amount.

With induction we can show that it is always possible to solve the binary problem if they have the same numbers of 1 and 0.

Base Case: For an empty circle it is already solved thus fulfilling our induction hypothesis.

Step: Let's start from a arbitrary pole x and walk around the circle, keeping track of the difference between 1 and 0 we've seen so far. After the first step the difference will be -1 (assuming w.l.o.g. that the x is a 0). If the last element is a 1 we can connect x with the last element. Otherwise the difference will be 1 right before the end (as in the end the difference has to be 0 in the end). And after every element the difference can only change by $+1$ or -1. With that we can conclude that there exists a pole where the difference will be 0. If we look at the first occurrence we see that this pole has to have been a 1 (to increase the difference from something negative to 0). Thus this pole fulfills the good wire condition and can be connected to x. The two resulting sub problems are now strictly smaller and they can be solved according to our induction hypothesis.

One useful result from that is that a wire connecting two neighboring poles (where one is 1 and the other is 0) is always good, because one section is empty and the other contains all the remaining poles. So each section has still an equal amount of both.

Subtask 3: Medium Battery (10 points)

We can use the result from above to solve the more general problem. We just need to find a 1 next to a 0, connect them with a wire, remove them from the list and do the same thing with the smaller list until the list is empty.

The correctness follow from our result above as we only connect good wires.

This approach can be implemented in $O(N^2)$ runtime, as finding a pair as well as removing them from the list takes $O(N)$ time, and we need to do it for N wires. We need $O(N)$ space for the list of not connected poles.

Solution code will be added soon

Subtask 4 (Theoretical): Future Developments

Note, this is a theoretical task. In those a correct solution isn't enough to get all the points. A sizable part of the points is also rewarded for explaining why the solution is correct. In this case things like the 'Observations' and their respective explanation are what we would expect. You also need to argue about your runtime and memory usage. For simple algorithms like above a sentence is usually enough. If it gets more complicated though you have to write up a detailed analysis of them.

$O(N)/O(N)$ (40 points)

We'll use the same strategy as in Subtask 3, but we want to find the pairs in a faster fashion. Notice that it is useless work to start again from the beginning, as deleting a neighboring pair

won't introduce a new pair before. Also at the point of deletion we didn't even look at the rest of the poles yet, so it is not necessary to already have them in the list. These two facts allow us to change our datastructure from a list to a stack. We'll compare the next element with our current top of the stack: if we can connect them with a good wire we will do so on remove the top from the step and look a the element after (which is like deleting in our list before). If we can't connect them then we push the current element to the stack (which is like iterating in our list before).

This change will improve the slow step of the previous algorithm of finding and deleting the pair from $O(N)$ to $O(1)$ and thus leading us to a $O(N)$ runtime solution. The space is still $O(N)$ as the stack could have N elements in the worst case.

The correctness follows from the correctness of Subtask 3, we just changed how we find and delete the poles.

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

using namespace std;

#define int long long

int N;
vector<int> potentials;


bool is_great(int i){
        return potentials[i] >= N;
}

void solve(){
        vector<int> stack;
        for(int i = 0; i < 2*N; i++){
                if(stack.size() == 0 || is_great(stack.back()) == is_great(i)){
                        stack.push_back(i);
                }else{
                        cout << stack.back() << " " << i << "\n";
                        stack.pop_back();
                }
        }
}

signed main() {

    ios_base::sync_with_stdio(false);
    cin.tie(0);

    cin >> N;
    potentials = vector<int>(2*N);
    for(int i = 0; i < 2*N; i++){
            cin >> potentials[i];
    }

    solve();
}
```

Alternative Proof for $O(N)/O(N)$

I want to show an alternative proof for the stack based solution, as this was by far the most popular solution. We still require the observations of 'Maximum Power', 'Reduction to Binary Problem' and especially 'Overlapping' but we don't need the 'Good Wires'.

In general optimization problems we need to proof that our algorithm terminates, outputs a valid solution and show that no better solution exists.

The last part is already covered with the 'Maximum Power' & 'Reduction to a Binary Problem' part.

For the rest we need some helper observations. First, the stack at each times either contains only 1 or only 0. That is because we only push to the stack if it's the same as the top element. Second, the stack will be empty at the end. Since every element not in a stack is connected by

a wire, which is a pair of 0 and 1, and since they were in equal amount in the beginning the remaining elements in the stack have to have the same amount of 1 and 0. With the first fact this can only be the case if the stack is empty and thus every pole is connected to exactly one other pole.

For the overlapping issue we will do a proof of contradiction: Assume we have two wires (i, j) and (i', j') (w.l.o.g. $i < j, i' < j', i < i'$) that overlap. Therefor (with the overlapping observation) they have to fulfill the inequalities: $i < i' < j < j'$, which is therefore also the order we iterate them on the circle. As the stack only contains elements which are not matched yet the elements i and i' have to be in the stack if we look at j. Now because a stack is order LIFO (last in, first out) the element i' has to be matched before the element i which implies a contradiction with our assumption that i' is connected to j' which comes after j. Therefor two wires can't overlap in the output of our solution.

The runtime is exactly $O(N)$ as we have a loop from 0 to $2N - 1$ with constant operations in its body. Therefore we have also shown that it terminates. The stack certainly won't grow past $2N$ elements as every element is in the stack at most once. With subtask 1 as an example it is also clear that the stack could grow to N elements. Thus the memory needed is also $O(N)$.

$O(N \cdot \log(N))/O(\log(N))$ (60 points)

The idea here is to find a good wire for a specific pole, connect them, solve the smaller sub section recursively and update our state to solve the remaining bigger sub problem. A crucial trick here is to not explicitly solve the big sub problem recursively. In this ways we can bound the depth of our recursion. (One can also try to program the function *tail-recursively* and let the compiler optimize the call away).

Another way to think about it is that we want to split our big subproblem into several smaller sub problem such that each of them is at most half the size.

To do that efficiently we will iterate from a given start point simultaneously in both directions (clockwise & counter-clockwise) and for each side keep track of the difference between the number of *greater* poles and the number of *smaller* poles. We will iterate the two pointer until one shows to a pole which can be connected with a *good* wire to our start, then we will connect it with the start and recursively solve the sub problem under the wire we just iterated through. Note that we only need to pass the start and the end as none of the wires in between will have been connected. After the recursive call return we will update our current begin and end to solve the rest of the problem (by finding the next small sub problem).

```python
def solve(N, potentials):

        def is_great(i):
                return potentials[i] >= N

        def solve_segment(begin, end):  # end is exclusive
                if(end <= begin): return

                # setup
                diff_left, diff_right = 0, 0
                left, right = begin + 1, end - 1

                while left <= right:
                        if diff_left == 0 and is_great(begin) != is_great(left):
                                print(begin, left)
                                solve_segment(begin + 1, left)
                                # like solve_segment(left + 1, end)
                                # instead of tail-recursion
                                begin = left + 1
                                # re-setup
                                diff_left, diff_right = 0, 0
                                left, right = begin + 1, end - 1
                        elif diff_right == 0 and is_great(begin) != is_great(right):
                                print(begin, right)
                                solve_segment(right + 1, end)
```

```
                                # like solve_segment(begin, right)
                                begin = begin + 1
                                end = right
                                # re-setup
                                diff_left, diff_right = 0, 0
                                left, right = begin + 1, end - 1
                            else:
                                # iteration
                                diff_left += 1 if is_great(left) else -1
                                left += 1
                                diff_right += 1 if is_great(right) else -1
                                right += -1

        solve_segment(0, 2*N)


n = int(input())
p = list(map(int, input().split()))

solve(n, p)
```

Correctness. Is our output valid: is every pole connected & are there no overlaps?. Assume we have an unconnected pole and we will look at the first, in order our algorithm, pole which isn't connected. We will look at the time one of the pointers first reaches set pole. This can only happen at the beginning or right after we have completed a sub call. Therefore our remaining problem has to have the same number of *greater* and *smaller* poles left (as we assume that the sub calls correctly solve the problem). We know from the observations that a good wire has to exist and that we will find it, as we check every pole for its *good* wire condition. This though contradicts the existence of the unconnected pole. The no overlap part follows from the fact that the sub problem is solved recursively and thus independently from the other poles. Therefore there are no poles left in that segment to cross the wire over the sub problem.

Is our solution optimal? This follows directly from the observation of maximum power.

Note that $n = 2N$ for the following sections.

Memory We don't change the input so it doesn't count to our memory. The only thing we have is out recursive function. The memory usage of these is the product of one function call and the maximum depth of the recursion. The memory usage of one function call is $O(1)$ as we only have constant variables. The depth is bounded by $O(\log n)$ as we guarantee in every call that the size of the range is at most half. And for such a thing it is well know to reach the base case in at most $\log_2 n$ steps. So the total memory is $O(\log n)$.

Runtime We only have our recursive function to analyze. Let's first define k_i as the size of the sub section we recurse on from the top level (which is the distance of the wires - 1). These have two useful properties: $\sum_i k_i + 2 = n$, the sum of the sub sizes is at the whole size, and $k_i \leq \frac{n-2}{2}$, as we recurse only on small segments (the ±2 comes from the fact that we have already connected two poles).

For recursive functions we usually define $T(n)$ as the total runtime of the recursion of size n (which in our case is the size of the segment). We then have to find a relationship to solve for $T(n)$. In this case it is $T(n) = C \cdot n + \sum_i T(k_i)$. The second part sums up all the work for the recursive calls. The first part is the linear work the function does without the sub calls. The reason it that for every time we find a sub problem of size k_i we have spent $k_i + 2$ iterations on both sides to find it and thus the whole part of finding the sub problems is in $O(\sum_i k_i + 2) = O(n)$.

Note that we can actually choose C to be any constant, as long as it is large enough to account for the linear work. Also with log we mean $\log_2$.

We now prove by induction that $T(n) \leq C \cdot (n+1) \cdot \log(n+2)$.

Base case: $T(0) \leq C = C(0+1)\log(0+2)$ This is true if we choose a large enough C as the base case only returns.

Step:

$$T(n) = C \cdot n + \sum_i T(k_i) \text{ (by our analysis)} \tag{.1}$$

$$= Cn + \sum_i C(k_i + 1)(\log(k_i + 2)) \text{ (by induction hypothesis)} \tag{.2}$$

$$\leq Cn + \sum_i C(k_i + 1)(\log(\frac{n+2}{2})) \text{ (using } k_i \leq \frac{n-2}{2}) \tag{.3}$$

$$= Cn + C(\log(\frac{n+2}{2}) \sum_i (k_i + 1) \text{ (factoring out)} \tag{.4}$$

$$\leq Cn + C(\log(n+2) - \log(2)) \cdot n \text{ (using log and } \sum_i k_i + 2 = n) \tag{.5}$$

$$= Cn \log(n+2) \leq C(n+1)\log(n+2) \tag{.6}$$

Instead of induction one could also use the *Master Theorem*.

$O(N \cdot \log(N))/O(1)$ **(70 points)**

There are various divide-and-conquer solution that run in $O(N \log N)$ time with $O(\log N)$ space. We will now describe such a solution that only uses $O(1)$ space. The key idea for this solution is to turn the input into a balanced bracket sequence, figure out which pairs of brackets belong to each other and then connect the corresponding poles.

First, we'll replace each pole with potential $0, \ldots, N-1$ by an opening bracket and each pole with potential $N, \ldots, 2N-1$ by a closing backet. Next, we'll cut open the circle. The cutting point is chosen in such a way that we get a balanced bracket sequence. This can be done in $O(N)$ time and $O(1)$ space by a simple sweep around the circle. Note that we don't need to explicitly form this bracket sequence. For our algorithm, we only need to answer questions of the form "What is the i-th bracket from the left and which pole does it correspond to?", which can be done in $O(1)$ time and space if we know the cutting point.

A balanced bracket sequence can be *uniquely* decomposed into pairs of one opening and one closing bracket such that the opening bracket is to the left of the closing one and such that two such pairs are either disjoint or one contains the other. For example, such a decomposition might look like this: (((())())(())). If we then connect the poles in each such pair, then we'll achieve maximum power, as we always match an opening brachet with a closing one, and we won't have any wire crossings, as there can't be two bracket pairs forming an intersection like this: (()). Hence, all we have to do is figure out which opening bracket is paired with which closing one.

Finding bracket pairs

In general, a (possibly unbalanced) bracket sequence looks like this $\ldots)\ldots)\ldots)\ldots(\ldots(\ldots$ where each $\ldots$ is a (possibly empty) *balanced* bracket sequence. Given such a bracket sequence of length k, it is possible to iterate over all red brackets from left to right in $O(k)$ time with $O(1)$ space by sweeping from left to right and keeping track of how many unmatched opening brackets we have found so for. If we encounter a closing bracket, then this bracket is red if and only if we have zero unmatched opening brackets. Similarly, we can iterate over the blue brackets from right to left.

This allows for the following divide-and-conquer solution: Consider a recursive function that takes a (possibly unbalanced) bracket sequence and matches all brackets in this sequence that aren't red or blue. This function first splits the bracket sequence in half and recursively solves each part. Next, it simultaneously iterates over the blue brackets in the left half and over the red brackets in the right half and matches red brackets with blue ones. For example, in a situation like this $)\ldots(\ldots(\ldots(\,\big|\,)\ldots)\ldots(\ldots(\,$, the brackets would be matched as follows $)\ldots(\ldots(\ldots(\,\big|\,)\ldots)\ldots(\ldots(\,$, after which the bracket sequence looks like this: $)\ldots(\ldots(\ldots(\,$.

A recursive implementation would use $\Theta(\log N)$ stack space. To avoid this, we use an iterative bottom-up implementation: Initially, every bracket forms a block of size one. In the first iteration, we'll merge the first block with the second one, the third block with the fourth one, and so on. This leaves us with blocks of size two (the last block may be smaller if N is not a power of two). Next, we'll merge blocks of size two to get blocks of size four. This is repeated until we're left with a single block of size $2N$. More precisely, there are $\log_2(2N)$ iterations. In the i-th iteration, we're given a bunch of blocks of size $s = 2^i$. We merge $[0, s-1]$ with $[s, 2s-1]$, merge $[2s, 3s-1]$ with $[3s, 4s-1]$ and so on. Thus, we can use simple for loops to figure out which parts have to be merged, so we only need $O(1)$ space. The total running time is $O(N \log N)$, as there are $O(\log N)$ iterations that each take $O(N)$ time.

```python
# O(n log n) time, O(1) space
def solve(n, p):
    # find cutting point
    offset = 0
    bal = 0
    for i,e in enumerate(p):
        if e < n:
            bal += 1
        else:
            bal -= 1
            if bal < 0:
                offset = i+1
                bal = 0
    # helper functions to find i-th bracket
    def index(i):
        return (offset+i)%(2*n)
    def get(i):
        return '(' if p[index(i)] < n else ')'

    s = 1
    while s <= 2*n:
        for i in range(0, 2*n-s, 2*s):
            # merge [i, i+s-1] with [i+s, i+2s-1]
            l = i+s-1
            r = i+s
            while True:
                # find next unmatched ( in left part
                bal = 0
                while l >= i:
                    if get(l) == '(':
                        if bal == 0:
                            break
                        bal -= 1
                    else:
                        bal += 1
                    l -= 1
                # find next unmatched ) in right part
                bal = 0
                while r < min(i+2*s, 2*n):
                    if get(r) == ')':
                        if bal == 0:
                            break
                        bal -= 1
                    else:
                        bal += 1
                    r += 1
                if l >= i and r < min(i+2*s, 2*n):
                    yield (index(l), index(r))
                    l -= 1
                    r += 1
                else:
                    break
        s *= 2

```

```
T = int(input())
for t in range(T):
        n = int(input())
        p = list(map(int, input().split()))

        print(f"Case #{t}:")
        for (a, b) in solve(n, p):
                print(a, b)
```

Lily Pads

Task Idea	Daniel Rutschmann
Task Preparation	Daniel Rutschmann, Johannes Kapfhammer, Martin Chikov
Description English	Johannes Kapfhammer, Martin Chikov
Description German	Joël Mathys
Description French	Florian Gatignon
Solution	Daniel Rutschmann
Correction	TODO

Changi Falls

Task Idea	Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer, Jan Schär
Description English	Johannes Kapfhammer
Description German	Jan Schär
Description French	Elias Boschung
Solution	Johannes Kapfhammer
Correction	Johannes Kapfhammer

In this task you were given a set of non-intersecting circles, where each circle has a value h_i. The goal was to find a sequence of at most L circles where the circles are adjacent and the values of the circles are increasing, such that the difference between largest and smallest value is maximized.

The first observation to make is that this is actually a graph task: If we put circles are vertices and the "adjacency" property as edges, we end up with a graph.

And in fact, the graph is a tree. Let's say a circle is a parent of another circle if it is adjacent and its radius is larger. Then the edges represent the adjacency property. Also all edges point to smaller circles, so there can never be cycles.

So this task actually reduces to two subtasks:

- compute the tree represented by the circles
- find the path of length at most L of increasing values that maximizes the value difference in that tree.

Subtask 1: Concentric Circles (15 points)

In the first subtask all circles were concentric, which means that the resulting tree is actually a path. To compute the order, of the path, one could just sort the circles by radius.

To find the solution, there were many possible approaches. One could first find the best increasing path and then do the same again for decreasing (or multiply the values by -1). The code below does this in one go by keeping track of two things:

- the current direction (whether the sequences is increasing or decreasing)
- the earliest possible start of the sequence given the direction.

For a new value, if it keeps the direction, just make the sequence longer. If it changes the direction, start a new sequence from the previous value.

```
#include <bits/stdc++.h>
using namespace std;
#define int int64_t

struct terrace { int x, y, r, h; };

int solve_sub1() {
  int N, L; cin >> N >> L;
  vector<terrace> ts;
  for (int i=0; i < N; ++i) {
    terrace t;
    cin >> t.x >> t.y >> t.r >> t.h;
    ts.push_back(t);
  }
  sort(ts.begin(), ts.end(),
       [&](terrace const& lhs, terrace const& rhs) {
         return lhs.r < rhs.r;
       });

```

```
  bool last_cmp = false;
  int l = 0;
  int best = 0;
  for (int r=1; r < N; ++r) {
    bool cmp = ts[r-1].h < ts[r].h;
    if (last_cmp != cmp) {
      last_cmp = cmp;
      l = r - 1;
    }
    l = max(l, r - L + 1);
    best = max(best, abs(ts[r].h - ts[l].h));
  }
  return best;
}

signed main() {
  int t; cin >> t;
  for (int i=0; i<t; ++i)
    cout << "Case #" << i << ": " << solve_sub1() << '\n' << flush;
}
```

The running time is $O(n \log n)$, dominated by the sorting.

Subtask 2: Flowing Inwards (15 points)

The second subtask was already quite tricky: the only restriction was that the heights and the radii are the same.

This means we need to come up with a good idea on how to build the tree from the input data.

There are two key ideas: first, we can write a function that checks whether some terrace a is inside another terrace b by checking that a has smaller radius and the distance between the center of a and the center of b is smaller than the radius of b. If we square the equations, this can be done without any use of doubles:

```
int sq(int x) { return x*x; }
// is terrace a inside terrace b?
bool inside(terrace const& a, terrace const& b) {
  return a.r < b.r && // a has smaller radius and ...
    sq(a.x - b.x) + sq(a.y - b.y) < sq(b.r); // a's center is contained inside b
}
```

So why can't we just check all pairs of terraces and add edges between the pairs that are inside? Well, because if three terraces a, b and c are nested, then b is inside a, c is inside b but also c is inside a. And we don't want an edge beween c and a.

One way of resolving this is by noting that each terrace has exactly one outer terrace. And the outer terrace is the one with the smallest radius. So we could compute all outer terraces and take the smallest, but there's an easier way of doing it: we sort the terraces by radius (which is doing some kind of topological sort). Then we start with the smallest terrace and move on to larger ones. We again check all smaller terraces, but we only add those that are inside and *have not yet been added*. If a, b and c are nested, we first add the pair (b, c) and when we check the pair (a, c) we note that c has already been added to b and don't add it to a as well.

In code form:

```
struct terrace { int x, y, r, h; };
vector<vertex> build_tree(vector<terrace> ts) {
  // sort in decreasing order of radius
  sort(ts.begin(), ts.end(), [](terrace const& lhs, terrace const& rhs) {
    return lhs.r > rhs.r;
  });
  const int N = ts.size();
  vector<vertex> tree(N);

  for (int i=0; i < N; ++i)
    tree[i].value = ts[i].h;
```

```
12
13   for (int i=N-1; i >= 0; --i)
14     for (int j=i+1; j < N; ++j)
15       if (tree[j].is_root && inside(ts[j], ts[i])) {
16         tree[i].children.push_back(j);
17         tree[j].is_root = false;
18       }
19   return tree;
20 }
```

The last challenge remaining is to compute the path of length at most L with largest height difference.

For that we can do a DFS (depth-first search). We pass along the path to each vertex. Then we can simply compute the height difference of each node by looking at the vertex in the path L vertices before.

```
1 int dfs(int L, vector<vertex> const& tree, int v, vector<int>& path) {
2   path.push_back(tree[v].value);
3   int best = path[max((int)0, (int)path.size()-L)] - path.back();
4   for (auto w : tree[v].children)
5     best = max(best, dfs(L, tree, w, path));
6   path.pop_back();
7   return best;
8 }
```

This is fast because we pass a copy to the path: we only need to add the current value once and pop it once. After DFS returns, the path has not changed, that's why this is correct.

The full code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  #define int int64_t
4
5  struct terrace { int x, y, r, h; };
6
7  int sq(int x) { return x*x; }
8
9  // is terrace a inside terrace b?
10 bool inside(terrace const& a, terrace const& b) {
11   return a.r < b.r && // a has smaller radius and ...
12     sq(a.x - b.x) + sq(a.y - b.y) < sq(b.r); // a's center is contained inside b
13 }
14
15 pair<int, vector<terrace>> read_input() {
16   int N, L; cin >> N >> L;
17   vector<terrace> ts;
18   for (int i=0; i < N; ++i) {
19     terrace t;
20     cin >> t.x >> t.y >> t.r >> t.h;
21     ts.push_back(t);
22   }
23   return {L, ts};
24 }
25
26 struct vertex {
27   vector<int> children;
28   int value = -1;
29   bool is_root = true;
30 };
31
32 vector<vertex> build_tree(vector<terrace> ts) {
33   // sort in decreasing order of radius
34   sort(ts.begin(), ts.end(), [](terrace const& lhs, terrace const& rhs) {
35     return lhs.r > rhs.r;
36   });
37   const int N = ts.size();
38   vector<vertex> tree(N);
```

```cpp

  for (int i=0; i < N; ++i)
    tree[i].value = ts[i].h;

  for (int i=N-1; i >= 0; --i)
    for (int j=i+1; j < N; ++j)
      if (tree[j].is_root && inside(ts[j], ts[i])) {
        tree[i].children.push_back(j);
        tree[j].is_root = false;
      }
  return tree;
}

int dfs(int L, vector<vertex> const& tree, int v, vector<int>& path) {
  path.push_back(tree[v].value);
  int best = path[max((int)0, (int)path.size()-L)] - path.back();
  for (auto w : tree[v].children)
    best = max(best, dfs(L, tree, w, path));
  path.pop_back();
  return best;
}

int solve_sub2() {
  auto [L, ts] = read_input();
  auto tree = build_tree(move(ts));
  int best = 0;
  vector<int> path;
  for (size_t i = 0; i < tree.size(); ++i)
    if (tree[i].is_root)
      best = max(best, dfs(L, tree, i, path));
  return best;
}

signed main() {
  int t; cin >> t;
  for (int i=0; i<t; ++i)
    cout << "Case #" << i << ": " << solve_sub2() << '\n' << flush;
}
```

The running time is $O(n^2)$ for computing the tree and $O(n)$ for the DFS; so $O(n^2)$ in total.

Subtask 3: Jewel Changi (20 points)

This is the full task but with small limits. In particular, the parsing from before still is fast enough, but we need some new idea how to deal with water flowing outwards as well as inwards.

One idea is to *direct* the edges according to value: if a has a bigger value then b, add the edge $a \to b$, otherwise add $b \to a$.

The resulting graph is something that is called a "polytree". It's not exactly a tree, as there can be multiple in-edges and multiple out-edges per vertex. However, the underlying shape is a tree.

This means starting a DFS from each node not only works (as it computes all paths so it is correct by construction), but is also fast (since there are only very few paths in a tree).

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int int64_t

struct terrace { int x, y, r, h; };

int sq(int x) { return x*x; }

// is terrace a inside terrace b?
bool inside(terrace const& a, terrace const& b) {
  return a.r < b.r && // a has smaller radius and ...
    sq(a.x - b.x) + sq(a.y - b.y) < sq(b.r); // a's center is contained inside b
}
```

```cpp

pair<int, vector<terrace>> read_input() {
  int N, L; cin >> N >> L;
  vector<terrace> ts;
  for (int i=0; i < N; ++i) {
    terrace t;
    cin >> t.x >> t.y >> t.r >> t.h;
    ts.push_back(t);
  }
  return {L, ts};
}

struct vertex {
  vector<int> adj;
  int value = -1;
  bool is_source = true;
};

vector<vertex> build_polytree(vector<terrace> ts) {
  // sort in decreasing order of radius
  sort(ts.begin(), ts.end(), [](terrace const& lhs, terrace const& rhs) {
    return lhs.r > rhs.r;
  });
  const int N = ts.size();
  vector<vertex> tree(N);

  for (int i=0; i < N; ++i)
    tree[i].value = ts[i].h;

  for (int i=N-1; i >= 0; --i)
    for (int j=i+1; j < N; ++j)
      if (tree[j].is_source && inside(ts[j], ts[i])) {
        if (ts[i].h > ts[j].h)
          tree[i].adj.push_back(j);
        else
          tree[j].adj.push_back(i);
        // use simlar to "is_root" before in order to add only direct edges
        tree[j].is_source = false;
      }

  // reset is_source and only mark those that have no incoming edges
  for (int i=0; i < N; ++i)
    tree[i].is_source = true;
  for (int i=0; i < N; ++i)
    for (auto w : tree[i].adj)
      tree[w].is_source = false;

  return tree;
}

int dfs(int L, vector<vertex> const& tree, int v, vector<int>& path) {
  path.push_back(tree[v].value);
  int best = path[max((int)0, (int)path.size()-L)] - path.back();
  for (auto w : tree[v].adj)
    best = max(best, dfs(L, tree, w, path));
  path.pop_back();
  return best;
}

int solve_sub3() {
  auto [L, ts] = read_input();
  auto tree = build_polytree(move(ts));
  int best = 0;
  vector<int> path;
  for (size_t i = 0; i < tree.size(); ++i)
    if (tree[i].is_source)
      best = max(best, dfs(L, tree, i, path));
  return best;
```

```
}

signed main() {
  int t; cin >> t;
  for (int i=0; i<t; ++i)
    cout << "Case #" << i << ": " << solve_sub3() << '\n' << flush;
}
```

This code has $O(n^2)$ running time, as a single DFS run takes $O(n)$ time since the underlying graph is a tree, and we start at most $O(n)$ DFS runs.

Subtask 4: Prepping for a Giant Jewel Changi (25 points)

This subtask was quite difficult, possibly the most difficult subtask of the first round this year.

The idea is to do a scanline approach over one coordinate and make a data structure over the other coordinate.

We start a scanline at $y = -\infty$ and move it upwards to $y = +\infty$. In the scanline we keep track of all intersection points of the scanline with the circles, and for each point we remember which to circle belongs (we explain later how exactly).

When we hit a new circle, we can figure out the direct outer parent of that circle by a case distinction with the two neighboring points. We then add that circle to the scanline data structure.

When a circle closes, we remove that circle from the data structure.

The data structure is a `set<halfcircle, scanline_cmp>`. A `halfcircle` stores a half circle in the shape of "(" or ")": center, radius, direction (whether it goes "(" or ")") and ID of the circle. `scanline_cmp` compares the halfcircles by their x coordinate, given y. We will change the y as the scanline progresses. As the circles don't intersect, we have the guarantee that the order of the half-circles does not change, so we are allowed to change y a little bit.

To show this in code, those are the definitions of `halfcircle` and `scanline_cmp`:

```
// shape of '(' or ')'
struct halfcircle {
  int x0, y0, r, id;
  int dir; // +1 for upper half, -1 for lower half

  // get the x coordinate of the halfcircle at position y
  double x_at_y(double y) const {
    assert(y0 - r <= y && y <= y0 + r);
    int cy = y - y0;
    return x0 + dir*sqrt(r*r - cy*cy);
  }
};

// comparison operator used in the std::set of this scanline
// the y member can be updated dynamically
struct scanline_cmp {
  scanline_cmp(int *y) : y(y) {}
  int *y;

  bool operator()(halfcircle const& a, halfcircle const& b) const {
    // sentinel: with id==-1, dir=-1 always smaller, dir=+1 always larger
    if (a.id == -1 && b.id == -1) return a.dir < b.dir;
    if (a.id == -1) return a.dir == -1;
    if (b.id == -1) return b.dir != -1;

    // tie breaker for left-most and right-most points of the same circle
    if (tie(a.x0, a.y0, a.r) == tie(b.x0, b.y0, b.r))
      return a.dir < b.dir;

    // otherwise the task guarantees no intersections
    assert(a.x_at_y(*y) != b.x_at_y(*y));
    return a.x_at_y(*y) < b.x_at_y(*y);
  }
};
```

So what about the scanline? For each circle we define two events:

- at $y - r$ we "open" the circle by adding the halfcircles "(" and ")".
- at $y + r$ we "close" the circle by removing the halfcircles again.

We order the events such that for a fixed y we first handle all closing events and then all opening events.

```
struct terrace { int x, y, r, h; };

int solve_sub4() {
  int n; cin >> n;
  vector<terrace> ts;
  vector<pair<int, int>> events;
  for (int i=0; i<n; ++i) {
    terrace t;
    cin >> t.x >> t.y >> t.r >> t.h;
    ts.push_back(t);
    assert(t.r > 0);
    events.emplace_back(t.y - t.r, i);   // for opening we store the id
    events.emplace_back(t.y + t.r, ~i);  // for closing we store its two's complement (-id-1)
  }
  sort(events.begin(), events.end());
```

Next up we prepare the scanline by setting up all required variables: `scanline_y`, the current y position of the scanline, `s`, the data structure containing all half circles and `par`, where we store the parent of a terrace.

```
int scanline_y = -3e9;
set<halfcircle, scanline_cmp> s(scanline_cmp{&scanline_y});
vector<int> par(n, -2); // -2 is the first invalid value (as -1 denotes no parent)

// compute the parent based on the iterator and its neighbors
auto get_par = [&](set<halfcircle, scanline_cmp>::iterator it) { /* shown later */ };

// add sentinels so prev(it) and next(it) are always valid
s.insert(halfcircle{0, 0, -1, -1, -1});
s.insert(halfcircle{0, 0, -1, -1, 1});
```

We also defined `get_par`, a function that computes the parent based on an iterator into the set. We show this function last.

Next to the scanline: we loop over all events, and either insert or remove the two half circles. If we insert them, we also set the parent using the `get_par` function.

```
for (auto [ev_y, id] : events) {
  scanline_y = ev_y;

  if (id >= 0) { // insert two new halfcircle
    auto& t = ts[id];
    auto [it, _] = s.insert(halfcircle{t.x, t.y, t.r, id, -1});
    par[id] = get_par(it);
    assert(par[id] != -2);
    s.insert(halfcircle{t.x, t.y, t.r, id, 1});
  } else { // remove two halfcircles
    id = ~id;
    auto& t = ts[id];
    auto it = s.find(halfcircle{t.x, t.y, t.r, id, -1});
    assert(it != s.end());
    it = s.erase(it); // erase returns an iterator to the next element
    assert(it->id == id); // it now points to the corresponding halfcircle, so it has the same id
    s.erase(it);
  }
}
```

The only thing that is missing now is how to compute the parent. For that we exploit the property of an `std::set` which ensures that its elements are sorted in increasing order of the

given comparator. Thus if we look at `prev(it)` and `next(it)` we can see the half circles to the left and to the right of the freshly inserted halfcircle. This allows us to do the following case distinction to determine the parent:

```
// compute the parent based on the iterator and its neighbors
auto get_par = [&](set<halfcircle, scanline_cmp>::iterator it) {
  auto& l = *prev(it); // the halfcircle to the left
  auto& r = *next(it); // the halfcircle to the right
  // we know both iterators are valid since we have added the two sentinels

  if (l.dir == -1 && r.dir == 1) { // case ( x ) -> found the parent
    assert(l.id == r.id);
    return l.id;
  }
  if (l.dir == -1) { // case ( x ( -> left one is parent
    assert(par[r.id] == l.id);
    return l.id;
  }
  if (r.dir == 1) { // case ) x ) -> right one is parent
    assert(par[l.id] == r.id);
    return r.id;
  }
  // case ) x ( -> parent of either one is parent
  assert(par[l.id] == par[r.id]);
  return par[l.id];
};
```

And that's it! The only stuff remaining is computing the hash and writing the output.

The running time is dominated by the `std::set`: we insert at most $O(n)$ values and call `find` at most $O(n)$ times. Thus our total running time is $O(n \log n)$. For reference, the full code for subtask 4:

```
#include <bits/stdc++.h>
using namespace std;
#define int int64_t

// shape of '(' or ')'
struct halfcircle {
  int x0, y0, r, id;
  int dir; // +1 for upper half, -1 for lower half

  // get the x coordinate of the halfcircle at position y
  double x_at_y(double y) const {
    assert(y0 - r <= y && y <= y0 + r);
    int cy = y - y0;
    return x0 + dir*sqrt(r*r - cy*cy);
  }
};

// comparison operator used in the std::set of this scanline
// the y member can be updated dynamically
struct scanline_cmp {
  scanline_cmp(int *y) : y(y) {}
  int *y;

  bool operator()(halfcircle const& a, halfcircle const& b) const {
    // sentinel: with id==-1, dir=-1 always smaller, dir=+1 always larger
    if (a.id == -1 && b.id == -1) return a.dir < b.dir;
    if (a.id == -1) return a.dir == -1;
    if (b.id == -1) return b.dir != -1;

    // tie breaker for left-most and right-most points of the same circle
    if (tie(a.x0, a.y0, a.r) == tie(b.x0, b.y0, b.r))
      return a.dir < b.dir;

    // otherwise the task guarantees no intersections
    assert(a.x_at_y(*y) != b.x_at_y(*y));
```

```
    return a.x_at_y(*y) < b.x_at_y(*y);
  }
};

struct terrace { int x, y, r, h; };

int solve_sub4() {
  int n; cin >> n;
  vector<terrace> ts;
  vector<pair<int, int>> events;
  for (int i=0; i<n; ++i) {
    terrace t;
    cin >> t.x >> t.y >> t.r >> t.h;
    ts.push_back(t);
    assert(t.r > 0);
    events.emplace_back(t.y - t.r, i);   // for opening we store the id
    events.emplace_back(t.y + t.r, ~i);  // for closing we store its two's complement (-id-1)
  }
  sort(events.begin(), events.end());

  int scanline_y = -3e9;
  set<halfcircle, scanline_cmp> s(scanline_cmp{&scanline_y});
  vector<int> par(n, -2); // -2 is the first invalid value (as -1 denotes no parent)

  // compute the parent based on the iterator and its neighbors
  auto get_par = [&](set<halfcircle, scanline_cmp>::iterator it) {
    auto& l = *prev(it); // the halfcircle to the left
    auto& r = *next(it); // the halfcircle to the right
    // we know both iterators are valid since we have added the two sentinels

    if (l.dir == -1 && r.dir == 1) { // case ( x ) -> found the parent
      assert(l.id == r.id);
      return l.id;
    }
    if (l.dir == -1) { // case ( x ( -> left one is parent
      assert(par[r.id] == l.id);
      return l.id;
    }
    if (r.dir == 1) { // case ) x ) -> right one is parent
      assert(par[l.id] == r.id);
      return r.id;
    }
    // case ) x ( -> parent of either one is parent
    assert(par[l.id] == par[r.id]);
    return par[l.id];
  };

  // add sentinels so prev(it) and next(it) are always valid
  s.insert(halfcircle{0, 0, -1, -1, -1});
  s.insert(halfcircle{0, 0, -1, -1, 1});

  for (auto [ev_y, id] : events) {
    scanline_y = ev_y;

    if (id >= 0) { // insert two new halfcircle
      auto& t = ts[id];
      auto [it, _] = s.insert(halfcircle{t.x, t.y, t.r, id, -1});
      par[id] = get_par(it);
      assert(par[id] != -2);
      s.insert(halfcircle{t.x, t.y, t.r, id, 1});
    } else { // remove two halfcircles
      id = ~id;
      auto& t = ts[id];
      auto it = s.find(halfcircle{t.x, t.y, t.r, id, -1});
      assert(it != s.end());
      it = s.erase(it);
      assert(it->id == id); // both halfcircles must be adjacent now
      s.erase(it);
```

```
    }
  }

  const uint64_t a = 1000003;
  const uint64_t mod = 1000000007;
  uint64_t ans = 0;
  uint64_t ai = 1;
  for (auto p : par) {
    ans = (ans + ai*(p + 2))%mod;
    ai = ai*a % mod;
  }
  return ans;
}

signed main() {
  int t; cin >> t;
  for (int i=0; i<t; ++i)
    cout << "Case #" << i << ": " << solve_sub4() << '\n' << flush;
}
```

Subtask 5: Giant Jewel Changi (25 points)

The last subtask could be solved with a lot of different techniques: With centroid decomposition in $O(n \log n)$ (which both solution we received make use of) or using smaller to larger with either `std::set`, fenwick trees or persistent queues.

We now explain the solution using centroid decomposition as it is the shortest solution.

Best solution containing vertex v

let's say we know that we are just interested in the best solution containing vertex v (i.e. some path with maximal difference, where the path contains v). Looking at the graph rooted in vertex v, we notice that some edges can be labelled as increasing and some as decreasing.

Focusing on just the increasing edges, we now want to compute the largest vertex at distance i for each i from 0 to L. We can do this by running a DFS from v and keeping track of the current distance:

```
void compute_increasing_paths(int v, int d) {
  if (d == L) return; // stop at distance L
  largest_at_dist[d] = max(largest_at_dist[d], height[v]);
  for (auto w : increasing_children[v])
    if (height[v] < height[w]) // if this is an increasing edge
      compute_increasing_paths(w, d+1);
}
```

The result of this DFS is the vector `largest_at_dist`.

We can do the same for the decreasing edges.

Now if we want the best solution containing vertex v we can just compute the maximum of `largest_at_dist[i] + largest_at_dist[L-i]` over all i.

Thus we can compute the best solution containing vertex v in $O(n)$ runing time.

Iteratively picking Centroids

Why is that useful? We can pick some vertex v and compute the best solution containing v using the algorithm above. Then we can remove v from the tree and end up with a smaller tree. We can then proceed doing the same on the smaller trees: pick some vertex v', compute the best solution containing v' and then remove v'. And so on until no more vertices remain.

This may remind you of the idea of divide and conquer: given an array, we compute something and then split the array into two halves. Then we solve each half recursively. One example of that would be quicksort: we pick some value v, put all smaller values on the left and all larger values to the right, and then sort each side recursively. It turns out quicksort would be $O(n \log n)$ assuming we could always find some value that would split the array exactly into two halves.

We now do a similar thing on the tree on vertices: we try find a central vertex such that the resulting tree is split into a new trees such that no tree has more than half of the vertices. Such a vertex is called a *centroid* and there is a theorem that each tree has at least one centroid.

Now if we iteratively find a centroid and remove it, and each iteration takes $O(\text{<tree size>})$ time, our resulting algorithm runs in $O(n \log n)$.

You can read more about centroid decomposition here: `https://tanujkhattar.wordpress.com/2016/01/10/centroid-decomposition-of-a-tree/`

An good implementation of centroid decomposition can be looked at here: `https://codeforces.com/blog/entry/58025`

Implementation

The code below is exactly doing that:

- we mark removed vertices by setting a flag in "`vector<bool> dead`".
- `compute_size` uses DFS to compute subtree sizes is a helper function for finding centroids.
- `find_centroid` finds a centroid of some component ignoring all dead vertices.
- `compute_paths` is conceptually the same function as `compute_increasing_paths`, however it takes a comparison function as argument so it works for both increasing and decreasing paths. It takes index I (either 0 for increasing or 1 for decreasing) and stores the result in the vector `q[I]`.
- `centroid_decomposition` is the main function for solving the task. It takes a vertex v, finds a centroid in the component of v, computes the paths and combines them.

  It is important that combining them runs in $O(\text{<tree size>})$ and not $O(L)$; for that the `compute_increasing_paths` returns the maximal depth found and we take care that the rest of the code only runs in $O(d0 + d1)$.
- `solve_sub5` is the driver function that reads the input, resets the global variables and calls `centroid_decomposition` in each component.

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int int64_t

int N, L;
vector<vector<int>> g;
vector<int> h;

vector<bool> dead;
vector<int> subtree_size;
const int INF = 2e18;
array<vector<int>, 2> q;

int compute_size(int v, int p) {
  subtree_size[v] = 1;
  for (auto w : g[v])
    if (w != p && !dead[w])
      subtree_size[v] += compute_size(w, v);
  return subtree_size[v];
}

// returns (heaviest child, vertex); assumes subtree_size is set correctly
pair<int, int> find_centroid(int v, int p, int total_size) {
  int heaviest_child = total_size - subtree_size[v];
  pair<int, int> best{total_size, v};
  for (auto w : g[v]) {
    if (w != p && !dead[w]) {
      best = min(best, find_centroid(w, v, total_size));
      heaviest_child = std::max(heaviest_child, subtree_size[w]);
    }
  }
```

```cpp
  return min(best, {heaviest_child, v});
}

template <size_t I, typename Comparator>
int compute_paths(int v, int p, int d, Comparator comp={}) {
  if (d == L) return -1;
  q[I][d] = max(q[I][d], h[v], comp);
  int maxd = d;
  for (auto w : g[v])
    if (w != p && !dead[w] && comp(h[v], h[w]))
      maxd = max(maxd, compute_paths<I>(w, v, d+1, comp));
  return maxd;
}

int centroid_decomposition(int v) {
  int c = find_centroid(v, -1, compute_size(v, -1)).second;

  int d0 = compute_paths<0>(c, -1, 0, less<int>{});
  int d1 = compute_paths<1>(c, -1, 0, greater<int>{});
  // compute prefix minimum/maximum
  for (int i = 1; i <= d0; ++i) q[0][i] = max(q[0][i], q[0][i-1]); // not necessary
  for (int i = 1; i <= d1; ++i) q[1][i] = min(q[1][i], q[1][i-1]);

  // combine the paths
  int ans = 0;
  for (int i = 0; i <= d0; ++i)
    ans = max(ans, q[0][i] - q[1][min(d1, L - i - 1)]);

  // reset q[0] and q[1]
  fill(q[0].begin(), q[0].begin()+d0+1, 0);   // reset q0
  fill(q[1].begin(), q[1].begin()+d1+1, INF); // reset q1

  // mark c as dead and recurse
  dead[c] = true;
  for (auto w : g[c]) // solve each subtree
    if (!dead[w])
      ans = max(ans, centroid_decomposition(w));

  return ans;
}

int solve_sub5() {
  // reset global variables
  cin >> N >> L;
  g.assign(N, {});
  h.clear();
  h.reserve(N);

  // read input
  vector<int> roots;
  for (int i=0; i < N; ++i) {
    int _, h_, p;
    cin >> _ >> _ >> _ >> h_ >> p;
    h.push_back(h_);
    if (p != -1) {
      g[i].push_back(p);
      g[p].push_back(i);
    }
  }

  int n = g.size();
  subtree_size.assign(n, 0);
  dead.assign(n, false);
  q[0].assign(n, 0);
  q[1].assign(n, INF);

  int ans = 0;
  for (int i=0; i<N; ++i)
```

```
100     if (!dead[i])
101       ans = max(ans, centroid_decomposition(i));
102   return ans;
103 }
104
105 signed main() {
106   int t; cin >> t;
107   for (int i=0; i<t; ++i)
108     cout << "Case #" << i << ": " << solve_sub5() << '\n' << flush;
109 }
```

The running time of this code is $O(n \log n)$.

Satay

Task Idea	Bibin Muttappillil
Task Preparation	Daniel Rutschmann
Description English	Daniel Rutschmann
Description German	Jan Schär
Description French	Florian Gatignon
Solution	Daniel Rutschmann
Correction	Daniel Rutschmann

In this task N mice are standing next to each other, forming a tree. To goal is to get all skewers and plates to a single mouse. We may only move skewers or plates along edges of the tree and we may not move plates to a mouse that is holding skewers or move plates without the skewers on top of them.

Subtask 1: Just Skewers (10 points)

In this subtask, we're only dealing with skewers. Note that the risk of moving two skewers at once is the same as moving them separately. Hence, we may assume that only one skewer is moved at a time. Then our goal is to minimize the number of moves each skewer does.

If we fix the mouse that will end up with all the skewers, then it is clearly optimal to move every skewer along the unique shortest path to this mouse. The length of all these paths can be computed by a single BFS starting from the fixed mouse. This way, we can compute the minimal total risk needed to pass all skewers to this mouse in $O(N)$ time. We do this for every mouse and take the least risky one. The total running time is $O(N^2)$.

Subtask 2: Everything to Stofl (20 points)

Now we're also dealing with plates, but we know which mouse they should end up at. By playing around with some examples, you might figure out that we can always move the plates along the unique shortest path to Stofl and that all skewers except one need to take a small detour, so you might conjecture that the minimal total risk is $D \cdot (S + P) + 2(n - 1) \cdot S$ where D is the sum of distances from all mice to Stofl. This is indeed the case and we'll prove this is Subtask 4.

As in Subtask 1, we can compute D in $O(N)$ time with a single BFS starting from Stofl, so the total running time is $O(N)$.

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = int64_t;

ll get_total_dist(const int n, vector<vector<int> > const& g, int root){
    vector<int> dist(n, -1);
    queue<int> q;
    auto cand = [&](int i, int d){
        if(dist[i] == -1){
            dist[i] = d;
            q.push(i);
        }
    };
    cand(root, 0);
    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(auto &e:g[u]){
            cand(e, dist[u]+1);
        }
    }
    return accumulate(dist.begin(), dist.end(), 0ll);
```

```
}
void solve(){
    int n, S, P;
    cin >> n >> S >> P;
    vector<int> a(n-1), b(n-1);
    for(auto &e:a) cin >> e;
    for(auto &e:b) cin >> e;
    vector<vector<int> > g(n);
    for(int i=0; i<n-1; ++i){
        g[a[i]].push_back(b[i]);
        g[b[i]].push_back(a[i]);
    }
    ll d = get_total_dist(n, g, 0);
    ll ret = d*(S+P) + 2*S*(ll)(n-1);
    cout << ret << "\n";
}
int main()
{
    int TTT; cin >> TTT;
    for(int cas = 0;cas<TTT;++cas){
        cout << "Case #" << cas << ": ";
        solve();
    }
    return 0;
}
```

Subtask 3: Skewers and plates without Stofl (10 points)

In this subtask, we no longer know which mouse the skewers and plates have to end up at. As in Subtask 2, the minimal total risk for getting everything to mouse u is $D[u] \cdot (S+P) + 2(n-1) \cdot S$ where $D[u]$ is the sum of distances from all mice to mouse u. We would like to compute this for every mouse and then pick the least risky one. Unfortunately, running a BFS starting from every mouse runs in $\Theta(N^2)$ time in total, which is too slow. We hence need a faster way of computing $D[u]$ for every mouse u.

This can be done by doing dynamic programming over all (directed) edges with prefix sums at every mouse, but there is an easier solution: Suppose the mice u and v are connected by an edge. Let A be the set of mice on the u-side of this edge and let B be the set of mice on the v-side of this edge. Every mouse in A needs to take one more step to get to v instead of u and every mouse in B needs to take one less step to get to v instead of u. Therefore, we have

$$D[v] = D[u] + |A| - |B| = D[u] + n - 2 \cdot |B|$$

This leads to a faster solution: Root the tree arbitrarily and compute $D[root]$ in $O(N)$ time. Run a DFS to compute subtree sizes. This gives us $|B|$ for the edge from every mouse v to its parent. Finally, run a second DFS to compute $D[v]$ for all mice v. The total running time is $O(N)$, hence the memory usage is also $O(N)$.

```
#include <bits/stdc++.h>
using namespace std;
using ll = int64_t;

int n;
vector<vector<int> > g;
vector<int> subsize;
vector<ll> dist;

void dfs(int u, int p){
    subsize[u] = 1;
    for(auto const&e:g[u]) if(e != p){
        dfs(e, u);
        subsize[u] += subsize[e];
    }
}
void dfs_2(int u, int p){
```

```
    for(auto const&e:g[u]) if (e != p){
        dist[e] = dist[u] + n - 2*subsize[e];
        dfs_2(e, u);
    }
}
void solve(){
    int S, P;
    cin >> n >> S >> P;
    vector<int> a(n-1), b(n-1);
    for(auto &e:a) cin >> e;
    for(auto &e:b) cin >> e;
    g.assign(n, vector<int>());
    subsize.assign(n, 0);
    dist.assign(n, 0);

    for(int i=0; i<n-1; ++i){
        g[a[i]].push_back(b[i]);
        g[b[i]].push_back(a[i]);
    }
    dfs(0, -1);
    dist[0] = accumulate(subsize.begin(), subsize.end(), 0ll) - n;
    dfs_2(0, -1);
    ll d = *min_element(dist.begin(), dist.end());
    ll ret = d*(S+P) + 2*S*(ll)(n-1);
    cout << ret << "\n";
}
int main()
{
    int TTT; cin >> TTT;
    for(int cas = 0;cas<TTT;++cas){
        cout << "Case #" << cas << ": ";
        solve();
    }
    return 0;
}
```

Subtask 4: Subtask 4: More Skewers and Plates (50 points)

We'll now show that the minimal risk to get everything to mouse u is given by $D[u] \cdot (S+P) + 2(n-1) \cdot S$ where $D[u]$ is the sum of distances from all mice to mouse u. From this it easily follows that our solution for Subtasks 2 and 3 is correct.

First, note that the risk $S \cdot x + P \cdot y$ is linear. Therefore, we can compute the total risk by summing up the contribution of every single skewer and plate. If a skewer is moved z times, then it contributes $S \cdot z$ units of risk.

Feasibility

The following strategy achieves a total risk of $D[u] \cdot (S+P) + 2(n-1) \cdot S$.

1. Pick an arbitrary skewer at a leaf different from u and color the skewer and plate at this leaf green. Such a leaf always exists as any tree with $N \geq 2$ vertices has at least two leaves.
2. Move all other skewers on direct paths of u.
3. Move all non-green plates on direct paths to neighbors of u.
4. Move the green plate and skewer on a direct path to the closest neighbor of u.
5. Move all skewers onto the green plate.
6. Move all non-green plates to u.
7. Move all skewers with the green plate to u.

It is easy to see that this is a valid strategy: We can always move just skewers, we never move plates to mouse that is holding skewers and we always move plates together with all skewer currently on top of them.

In this strategy, every plate moves along the direct path to u, so the total risk contribution from plates of $D[u] \cdot P$. The skewers also move along direct paths to u, but every non green skewer takes two extra steps to move onto the green plate and back to u. Hence the total risk contribution from skewers is $D[u] \cdot S + 2(n-1) \cdot S$.

Optimality

We'll now show that any strategy to get everything to mouse u has a total risk of at least $D[u] \cdot (S+P) + 2(n-1) \cdot S$. Clearly, a single plate can't contribute less risk than P times the distance to u, so the plates contribute at least $D[u] \cdot P$ risk, Similarly, the skewers contribute at least $D[u] \cdot S$ risk. Moreover, every skewer that takes a detour, i.e. that doesn't just move along the shortest path to u, contributes an additional $2P$ risk: As trees are bipartite, any detour involves at least two extra steps. Thus, it suffices to show that there is at most one skewer that doesn't take a detour.

Let's color all skewers that don't take a detour green. Consider an arbitrary green skewer. If this skewer is ever moved away from its plate, then it will always be closer to u than the plate. As we can't move the plate to the skewer, this makes it impossible to get the skewer and plate to the same mouse. This contradicts the fact that all plates and skewers end up at u. Therefore, a green skewer may never move away from its plate.

Now suppose there are two (or more) green skewers. As green skewers may only be moved with their plates, it is impossible to move one green skewer to the mouse that is holding the other one. Thus, every mouse can hold at most one green skewer at a time, which again contradicts the fact that all plates and skewers end up at u. This shows that there is at most 1 green skewer, so at least $S-1$ skewers take a detour.

Muffins

Task Idea	Fabian Lyck
Task Preparation	Fabian Lyck
Description English	Fabian Lyck
Description German	Fabian Lyck
Description French	Florian Gatignon
Solution	Fabian Lyck
Correction	Fabian Lyck

Subtask 1: Greedy (25 points)

This subtask greatly simplifies the problem by posing some important constraints:

1. We have to bake all muffins. Thus there's no need to select a subset of orders to fulfill.
2. There will only be one order. This means, we can fetch all of the necessary dough, lemons and chocolate at the beginning in one go. This will be optimal, since returning to any of these stations multiple times incurs an additional switching time cost.
3. The number of cups is an integer multiple of the bowl capacity. It is optimal to repeat the cycle of bowl filling, mixing and cup filling until all cups are filled. Since it is an integer multiple, we can always fill the bowl fully. This maximizes the efficiency of mixing.

Taking these steps, we find the greedy solution consisting of gathering all ingredients and then repeating a baking cycle until all orders are fulfilled. During each baking cycle, we make sure to fill all of the cups to maximize baking efficiency. Only in the last cycle do we fill the cups partially. Since we always have to fill the bowls with an even amount of ingredients before mixing, we have to round up the ordered muffins to the next even number. When filling cups, it suffices to only fill the ordered number of cups and leave some dough in the bowl. Lastly, we should always wait until the oven has finished baking before trying to bake more muffins.

```
uint32_t oven_done = 0;
void mouse_baking_muffins(const Mouse& mouse, const Storages& old_storages,
        uint32_t finish_tick) {
    oven_done = finish_tick + contexts.baking.baking_time;
}

void make_muffins() {
    uint32_t muffin_type = orders[0].muffin_type;
    uint32_t amt_muffins = orders[0].amount;
    uint32_t half_muffins = (amt_muffins + 1) / 2;
    go_to(contexts.dough); // go to dough storage
    for (uint32_t i = 0; i < (half_muffins + contexts.dough.max_amount - 1) /
            contexts.dough.max_amount; i++) {
        get_dough();
    }
    if (muffin_type == 1) {
        go_to(contexts.chocolate); // go to chocolate storage
        for (uint32_t i = 0; i < (half_muffins + contexts.chocolate.max_amount - 1) /
                contexts.chocolate.max_amount; i++) {
            get_chocolate();
        }
    } else {
        go_to(contexts.lemons); // go to lemon storage
        for (uint32_t i = 0; i < (half_muffins + contexts.lemons.max_amount - 1) /
                contexts.lemons.max_amount; i++) {
            get_lemons();
        }
    }
    uint32_t dough_made = 0;
    uint32_t cups_filled = 0;
```

```
31      while (dough_made < amt_muffins) {
32          for (uint32_t i = 0; i < storages.max_cups /
33                  (storages.bowls[0].capacity * storages.bowls.size()); i++) {
34              go_to(contexts.bowl_filling); // go to bowl filling station
35              for (const Bowl& bowl : storages.bowls) {
36                  while (dough_made < amt_muffins and bowl.amount < bowl.capacity) {
37                      fill_bowl(bowl, muffin_type);
38                      dough_made += 2;
39                  }
40              }
41              go_to(contexts.mixing); // go to bowl mixing station
42              for (const Bowl& bowl : storages.bowls) {
43                  if (bowl.amount > 0) {
44                      mix_bowl(bowl);
45                  }
46              }
47              go_to(contexts.cup_filling); // go to cup filling station
48              for (const Bowl& bowl : storages.bowls) {
49                  while (bowl.amount > 0 and storages.cups[muffin_type] < storages.max_cups and
50                          cups_filled < amt_muffins) {
51                      fill_cups(bowl);
52                      cups_filled ++;
53                  }
54              }
55              if (dough_made >= amt_muffins) break;
56          }
57          go_to(contexts.baking); // go to oven
58          idle(oven_done);
59          bake_muffins();
60      }
61      idle();
62  }
```

Subtask 2: Optimization (25 points)

This subtask poses some tricky optimization problems. The only constraint was that there would be only one mouse and that the test data would always be the same. This allows manually analyzing and writing a solution tailored to the test data. There are many ways to approach this task. One idea would be to simulate the game and implement some different greedy strategies, then one could evaluate the score of each strategy in dynamic programming fashion to combine them into a solution.

Each of the seven test cases evaluates a different aspect of the bot. The first two cases each offer twice as many orders as Stofl can fulfill. By choosing the larger orders, more points can be scored. Since the mixing time is very long and the bowls are large, it is also necessary to mix all ingredients at the beginning. The third case requires Stofl to process two orders at the same time. There are two bowls and both muffin types have to be prepared and filled into cups in one go. The fourth case is the same, except that the number of bowls is reduced to one. So the dough has to be prepared one type after another, but both types of muffins still have to be baked at the same time. The fifth case simply consists of many small two-muffin orders that all have to be prepared from start to finish as they come in. The sixth case tests the ability to fill and mix all of the necessary dough in advance, to then fulfill orders of both muffin types as they come in. Two large bowls are provided, so all the necessary dough can be stored there. The seventh case is the same, except that there are more, smaller bowls. Thus it is necessary to correctly split up the dough among different bowls, so that still all of the dough can be prepared in advance.

Subtask 3: Creativity Tourney (50 points)

The idea of this subtask would've been to have a competition between different submissions. Where each submission would've been paired up with a mouse controlled by a different submission. The goal would be to bake as many muffins as possible, cooperating as well as possible with the other mouse. The strategies from subtask 2 should still be applicable, but

many more checks are necessary to ensure that the two mice don't interfere with each other. Unfortunately there was only one submission, so the creativity tourney did not take place.