Gummy Bears

Task Idea Christopher Burckhardt, Daniel Rutschmann
Task Preparation Johannes Kapfhammer
Description English Johannes Kapfhammer
Description German Tobias Feigenwinter
Description French Florian Gatignon
Solution Johannes Kapfhammer

In this task you were given a sequence $f_0, \ldots, f_{n-1}$ and you had to rearrange it into a $\frac{n}{k}$ groups such that each group consists of exactly $\frac{n}{k}$ elements and contains a majority element. In other words, inside each group, at least $\left\lfloor \frac{n}{k} \right\rfloor + 1$ elements had to be equal.

With the brackets $[x]$ we denote the floor function, which is operation of rounding down to the next smaller integer. Also see https://en.wikipedia.org/wiki/Floor_and_ceiling_functions. The ceil function, written as $\lceil x \rceil$ is rounding up to the next integer.

Let $R = k - (\left\lfloor \frac{n}{k} \right\rfloor + 1) = \left\lceil \frac{n}{k} \right\rceil - 1$. Then, the output will always have a form similar to this:

$$
\begin{array}{ccccccc}
  m_0 & m_0 & \cdots & m_0 & r_0 & r_1 & \cdots & r_{R-1} \\
  m_1 & m_1 & \cdots & m_1 & r_R & r_{R+1} & \cdots & r_{2R-1} \\
  \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \cdots & \vdots \\
  m_{n/k} & m_{n/k} & \cdots & m_{n/k} & r_{(n/k)R+1} & \cdots & r_{(n/k)R-1} \\
\end{array}
$$

The first $\left\lfloor \frac{n}{k} \right\rfloor + 1$ majority elements are identical and we call them “majority elements”. We don’t care about the value of other elements, so we call them the “filler elements”.

If we can somehow figure out the majority elements $m_0, \ldots, m_{n/k}$, we can just use the remaining elements as filler.

For that we can count the number of elements that are the same. Let’s say value $x$ occurs $c_x$ times.

We can use it for a majority element if $c_x \geq \left\lfloor \frac{n}{k} \right\rfloor + 1$. We can use it for two majority elements if $c_x \geq 2 \cdot (\left\lfloor \frac{n}{k} \right\rfloor + 1)$. So in total, we may use it for $\left\lfloor \frac{c_x}{\left\lfloor \frac{n}{k} \right\rfloor + 1} \right\rfloor$ majority elements.

50 Points

For 50 points, it was enough to check whether it’s possible or not. This can be done with the above observations: compute how many majorities (the elements $m_0, m_1, \ldots$) we can form, and check whether it will be at least $n/k$. 

```cpp
int n, k, s; cin >> n >> k >> s;
const int majority_quota = k/2 + 1;
unordered_map<int, int> flavour_count;
for (size_t i = 0; i < n; ++i) {
    int x; cin >> x;
    ++flavour_count[x];
}
int majorities = 0;
for (auto [f, cnt] : flavour_count) {
    majorities += cnt / majority_quota;
}
if (majorities >= n/k)
    cout << "Yummy!\n"
else
    cout << "Yuck!\n";
```
100 Points

For the reconstruction, you also need to keep track of the values of the majority elements and the values of the filler elements. The following solution builds the output table by first filling out the majority elements and then later pad it with the filler elements.

```cpp
int n, k, s; cin >> n >> k >> s;
const int majority_quota = k/2 + 1;

unordered_map<int, int> flavour_count;
for (int i = 0; i < n; ++i) {
    int x; cin >> x;
    ++flavour_count[x];
}

vector<vector<int>> blocks;
vector<int> filler;
for (auto [f, cnt] : flavour_count) {
    // repeatedly add blocks of size majority_quota
    while (((int)blocks.size()) < n/k &&
          cnt >= majority_quota) {
        cnt -= majority_quota;
        // add majority_quota times the value f
        blocks.emplace_back(majority_quota, f);
    }
    // all remaining elements will be used as filler
    for (int i=0; i<cnt; ++i)
        filler.push_back(f);
}
// not enough majority elements
if (((int)blocks.size()) < n/k) {
    cout << "Yuck!\n"
    return 0;
}
else {
    cout << "Yummy!\n"
    for (auto & block : blocks) {
        // fill it up to size k with filler elements
        while (((int)block.size()) < k) {
            block.push_back(filler.back());
            filler.pop_back();
        }
        copy(block.begin(), block.end(), ostream_iterator<int>(cout, " "));
        cout << '\n';
    }
}
```

Below is also a Python solution:

```python
from collections import Counter

n, k, s = map(int, input().split())
q = k // 2 + 1

count = Counter(map(int, input().split()))
majority = list(Counter({b:(c - c % q) for b, c in count.items()}).elements())
filler = list(Counter({b:(c % q) for b, c in count.items()}).elements())

if len(majority) // q < n // k:
    print("Tuck!\n")
else:
    print("Yummy!")
    for mouthful in zip(*[iter(majority[:q*n//k])]*q, *[iter(filler-majority[q*n//k:])]*(k-q)):
        print("mouthful")
```
**Water View**

Task Idea: Daniel Rutschmann  
Task Preparation: Luc Haller  
Description English: Luc Haller  
Description German: Tobias Feigenwinter  
Description French: Elias Boschung  
Solution: Luc Haller

**Subtask 1: 30 Points \((n \leq 500)\)**

To solve the first test group, we can iterate over all possible locations \(0 \leq i \leq n - 1\), and in an inner loop counting the number of beautiful segments for this choice of \(i\) by checking for each segment \(0 \leq j \leq n - 1\) if it’s beautiful by doing a third nested loop and checking if there’s no higher segment between \(j\) and the beginning, or the end, or \(i\).

The running time is \(O(n^3)\).

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

int main() {  
    int n; cin >> n;  
    vi hs(n);  
    for (int & hi : hs) cin >> hi;  
    int maxbeauty = 0;  
    for (int i=0; i<n; ++i) {  
        int beauty = 0;  
        for (int j=0; j<n; ++j) {  
            bool beautiful = true;  
            for (int k=j; k<n; ++k) {  
                if (k == i) {  
                    beautiful = true;  
                } else if (hs[k] > hs[j]) {  
                    beautiful = false;  
                }  
            }  
            if (beautiful) {  
                ++beauty;  
                continue;  
            }  
            beautiful = true;  
            for (int k=n-1; k>=j; --k) {  
                if (k == i) {  
                    beautiful = true;  
                } else if (hs[k] > hs[j]) {  
                    beautiful = false;  
                }  
            }  
            beauty += beautiful;  
        }  
        maxbeauty = max(maxbeauty, beauty);  
    }  
    cout << maxbeauty << endl;  
}  
```

**Subtask 2: 60 Points \((n \leq 10'000)\)**

To improve our solution from above and bring it down to a running time of \(O(n^2)\), we observe that it’s not really necessary to check each \(j\) with a separate loop over the segments: Starting from a lake in one direction, the segments which can see it are exactly those which are at least as
Round 2P, 2021

Task waterview

high as everything between themselves and the lake, so we can easily mark all of them in one scan from the lake. So to compute the beautiful segments if we place the third lake at a given \( i \), we’ll just do one scan from 0 up to mark all segments which can see the lake on the left, one scan from \( n-1 \) down for the lake on the right, and one scan left from \( i-1 \) and one right from \( i+1 \) for the lake at \( i \). After that, we count the number of marked segments.

This results in the desired running time of \( O(n^2) \).

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

using vi = vector<int>;

void markgood(const vi& hs, int from, bool reverse, vi& good) {
    int maxyet = 0;
    for (int i=from; reverse ? i>=0 : i<good.size(); i += reverse ? -1 : 1) {
        if (hs[i] >= maxyet) {
            good[i] = 1;
            maxyet = hs[i];
        }
    }
}

int main() {
    int n; cin >> n;
    vi hs(n);
    for (int& hi : hs) cin >> hi;
    int maxgood = -1;
    for (int i=0; i<n; ++i) {
        vi good(n);
        markgood(hs, 0, false, good);
        markgood(hs, n-1, true, good);
        markgood(hs, i-1, false, good);
        markgood(hs, i+1, false, good);
        good[i] = 1;
        int goodcount = accumulate(good.begin(), good.end(), 0);
        maxgood = max(maxgood, goodcount);
    }
    cout << maxgood << "\n";
}
```

Subtask 3: 100 Points \( (n \leq 500'000) \)

For the full score, we need to consider all possible locations for the third lake in a constant number of scans over the list of heights. First, we do a scan from 0 up and one from \( n-1 \) down to mark all segments which can see at least one of the two fixed lakes at the ends, since for these the placement of the third lake won’t matter.

We can count the number of beautiful segments to the left of each \( i \) and to the right of each \( i \) in two separate scans. So now we’ve essentially reduced the problem to computing a list \( L_i \), where \( L_i \) is the number of beautiful segments to the left of \( i \) if the third lake is placed at \( i \).

To compute the \( L_i \), consider the lists \( L_i \) of beautiful segments left of each \( i \) if the third lake is placed at \( i \), or more specifically, consider how \( L_{i+1} \) differs from \( L_i \): The only change in the relevant list of heights is that \( h_i \) is added at the end. No new segment before \( i \) can become visible because of this, but some segments may become hidden. By definition of visibility, everything at least as high as \( h_i \) stays visible, and everything lower is blocked from view. Also, \( i \) becomes visible.

This tells us how to compute \( L_{i+1} \) from \( L_i \), but if we implemented this naively by looking at all elements of \( L_i \), the worst case running time of the solution would still be quadratic. But we can combine the idea with the observation we made for the second test set: The heights of the elements of \( L_i \) are non-increasing. So to transform \( L_i \) into \( L_{i+1} \), we can just keep removing the last element until its height is at least \( h_i \) or the list is empty, then add \( i \). Note that we compare at most one element more than we remove, and each element is removed at most once (since it’s added at most once), so all of these transformations from \( L_i \) to \( L_{i+1} \) in total run in \( O(n) \).

So far I ignored the issue of how to take into account the two original lakes when computing the
\( L_i \) and \( l_i \). We can’t just deal with them fully separately, since that might lead to double-counting some beautiful segments. Consider an additional list \( m_i \) where we pre-marked the segments which can see an endpoint with 1 and the others with 0. So far \( l_i \) would just be \( |L_i| \), and \( L_i \) may contain some segments \( j \) which can also see one of the lakes at the ends. We can easily solve that by just not adding \( i \) to \( L_{i+1} \) if \( m_i = 1 \). (Note that we still have to remove the segments which become hidden, though.) Now \( l_i \) is just \( |L_i| + \sum_{j=0}^{i-1} m_j \).

But what if placing a lake at \( i \) and thereby removing \( h_i \) means additional \( j < i \) can now see the lake at the right end? This would mean that \( m_i \) doesn’t correctly represent visibility of the endpoints anymore. Yes, but that’s not a problem: Any such \( j \) can also see the lake at \( i \), since \( j < i < n \), so it must be in \( L_i \) and we still count it exactly once for \( l_i \). In fact, we don’t need to take visibility of the end into account for \( m_i \) at all by this argument.

We analogously compute \( r_i \) on the reversed height list, and the total beauty if the third lake is placed at \( i \) is \( l_i + r_i + 1 \).

As we showed above, computing all \( l_i \) and \( r_i \) runs in \( \Theta(n) \), so this solution has linear running time.

```
#include <bits/stdc++.h>
using namespace std;
using vi=vector<int>;

vi solveleft(const vi& hs) {
    const int n = hs.size();
    vi beginvis(n); // m_i in the solution text
    int maxh = 0;
    for (int i=0; i<n; ++i) {
        if (hs[i] >= maxh) {
            beginvis[i] = 1;
            maxh = hs[i];
        }
    }
    int leftbeginvis = 0; // running sum over the current prefix of m_i
    vi leftvis(n); // l_i
    vi visstack; // L_i
    for (int i=0; i<n; ++i) {
        leftvis[i] = leftbeginvis + visstack.size();
        if (beginvis[i]) ++leftbeginvis;
        while (!visstack.empty() && visstack.back() < hs[i])
            visstack.pop_back();
        if (!beginvis[i])
            visstack.push_back(hs[i]);
    }
    return leftvis;
}

int main() {
    int n; cin >> n;
    vi hs(n);
    for (int& hi : hs) cin >> hi;
    vi leftvis = solveleft(hs);
    reverse(hs.begin(), hs.end());
    vi rightvis = solveleft(hs);
    reverse(rightvis.begin(), rightvis.end());
    int maxvis = -1;
    for (int i=0; i<n; ++i)
        maxvis = max(maxvis, leftvis[i] + rightvis[i] + 1);
    cout << maxvis << endl;
    return 0;
}
```
Sliding mouse

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

Subtask 1: 25 Points \((n, m \leq 100 \text{ and there are no ice tiles})\)

In the first test group, we just need to find the length of the shortest path in a graph where all edges have length 1. This can be done with a simple BFS.

This solution runs in \(O(nm)\).

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

struct state {
    int x, y, distance;
};

struct point {
    int x, y;
};

vector<point> offset = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

int main () {
    int n, m;
    cin >> n >> m;
    vector<string> field(m);
    for(string &s : field) cin >> s;

    vector<vector<bool>> visited(m, vector<bool>(n, false));
    queue<state> q;
    for(int y = 0; y < m; ++y) {
        for(int x = 0; x < n; ++x) {
            if(field[y][x] == 'd') q.push({x, y, 0});
        }
    }

    while(!q.empty()) {
        state now = q.front();
        q.pop();
        if(visited[now.y][now.x]) continue;
        visited[now.y][now.x] = true;
        char type = field[now.y][now.x];
        if(type == '#') continue;
        if(type == 'g') {
            cout << now.distance << "\n";
            return 0;
        }
        for(point off : offset) {
            q.push({now.x + off.x, now.y + off.y, now.distance + 1});
        }
    }
    cout << "IMPOSSIBLE\n";
}
```

Subtask 2: 75 Points \((n, m \leq 1000)\)

We can compute for each node and direction, where mouse Daniel stops when going from that node in that direction. In this interpretation, edges can be longer than one (when Daniel slides over the ice), which means that we need to use Dijkstra’s algorithm.
Computing where mouse Daniel stops can be done by a linear scan, which takes \(O(n + m)\) and leads to an overall running time of \(O(nm(n + m))\). This solution achieves 75 points (or if the inner loop is very efficient, 100 points).

The solution can be improved by precomputing the ice slide stops. This can be done by a simple linear scan in all four directions, which takes \(O(nm)\). The overall solution then has running time \(O(nm \log(nm))\), which achieves 100 points.

Shown here is the solution without precomputation:

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

struct state {
    int x, y, distance;
};

bool operator <(const state &a, const state &b) {
    return a.distance > b.distance;
}

struct point {
    int x, y;
};

vector<point> offset = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

int main () {
    ios_base::sync_with_stdio(false);
    int n, m;
    cin >> n >> m;
    vector<string> field(m);
    for(string &s : field) cin >> s;
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    priority_queue<state> pq;
    for (int y = 0; y < m; ++y) {
        for (int x = 0; x < n; ++x) {
            if (field[y][x] == 'd') pq.push({x, y, 0});
        }
    }
    while (!pq.empty()) {
        auto [x, y, distance] = pq.top();
        pq.pop();
        if (visited[y][x]) continue;
        visited[y][x] = true;
        if (field[y][x] == 'g') {
            cout << distance << "\n";
            return 0;
        }
        for (point off : offset) {
            if (field[y + off.y][x + off.x] != '#') {
                int d = 1;
                while (field[y + d * off.y][x + d * off.x] == 'd' &&
                       field[y + (d + 1) * off.y][x + (d + 1) * off.x] != '#') {
                    d++;
                }
                pq.push({
                    x + d * off.x, y + d * off.y,
                    distance + d});
            }
        }
        cout << "IMPOSSIBLE\n";
    }
}
```

**Subtask 3: 100 Points \((n, m \leq 2000)\)**

The intended solution for 100 points is to use BFS on a state graph. This means that each tile corresponds to 4 nodes – one for each direction. Both the BFS queue and the visited vector need to be extended with this extra dimension. For a dirt (.) tile, all 4 nodes have a directed edge to each non-wall neighbor. But for an ice (+) tile, each of the 4 nodes only has a single outgoing edge.
corresponding to its direction (unless there is a wall in that direction, in that case it connects to all non-wall neighbors). It is not necessary to explicitly construct the state graph, instead the edges can be easily computed from the input.

The running time is \( O(nm) \).

A common mistake was to either not add the direction dimension to the visited vector, or to forget to check if a node was already visited.

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

struct state {
    int x, y, dir, distance;
};

struct point {
    int x, y;
};

vector<point> offset = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

int main () {
    ios_base::sync_with_stdio(false);
    int n, m;
    cin >> n >> m;
    vector<string> field(m);
    for (string &s : field) cin >> s;

    vector<vector<vector<bool>>> visited(m, vector<vector<bool>>(n, vector<bool>(4, false)));
    queue<state> q;
    for (int y = 0; y < m; ++y) {
        for (int x = 0; x < n; ++x) {
            if (field[y][x] == 'd') q.push({x, y, 0, 0});
        }
    }

    while (!q.empty()) {
        auto [x, y, dir, distance] = q.front();
        q.pop();
        if (visited[y][x][dir]) continue;
        char type = field[y][x];
        if (type == 'g') {
            cout << distance << "\n"
            return 0;
        }
        for (int ndir = 0; ndir < 4; ++ndir) {
            if (field[y + offset[ndir].y][x + offset[ndir].x] != '#') {
                q.push({
                    x + offset[ndir].x, y + offset[ndir].y, 
                    dir, distance + 1});
            } else {
                for (int ndir = 0; ndir < 4; ++ndir) {
                    if (field[y + offset[ndir].y][x + offset[ndir].x] != '#') {
                        q.push({
                            x + offset[ndir].x, y + offset[ndir].y, 
                            ndir, distance + 1});
                    }
                }
            }
        }
        cout << "IMPOSSIBLE\n";
    }
}
```

9/13
**Round 2P, 2021**

**Task ladderbalcony**

---

**Ladder Balcony**

Task Idea  
Bibin Muttappillil

Task Preparation  
Daniel Rutschmann

Description English  
Daniel Rutschmann

Description German  
Tobias Feigenwinter

Description French  
Florian Gatignon

Solution  
Daniel Rutschmann

---

40 Points \(O(NW^4)\)

This task can be solved with dynamic programming: Let \(DP[x][n][y]\) be the maximum amount of sunlight of any ladder balcony that is contained in the first \(x\) columns, uses exactly \(n\) ladders and has a ladder in the \(y\)-th position of the \(x\)-th column.

To compute \(DP[x + 1][n][y]\) from \(DP[x][n][y]\), we iterate over all ranges \([l, r]\) with \(0 \leq l \leq y \leq r < n\) and consider placing ladders at positions \(l, l+1, \ldots, r\) in the \((x+1)\)-st column. When placing ladders this way, we should pick the best option for the first \(x\) columns, which is \(\max_{l \leq y' \leq r} DP[x][n][y']\), add the sunlight for the \((x+1)\)-st column, i.e. \(a_{l,x} + \ldots + a_{r,x}\) and consider this as an option for \(DP[x + 1][n + (r - l + 1)][y]\).

Doing this naively takes \(O(NW^4)\) time, as we loop over \(x, n, y\), consider \(\Omega(\Delta^2)\) ranges and process each such range in \(O(H)\) time.

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

template<typename T>
void xmax(T&a, T const&b){
a = max(a, b);
}
constexpr int inf = 1e9;

signed main(){
    int H, W, n;
    cin >> H >> W >> n;

    vector<vector<int>> v(H, vector<int>(W));
    for(auto&e:v) {
        for(auto&f:e) {
            cin >> f;
        }
    }

    // Only store last 2 rows of table, re-use memory
    vector<vector<int>> dp2 = dp;
    vector<int> &v = dp;
    for(auto &v) {
        for(auto &v) {
            cin >> f;
        }
    }

    // iterate over vertical ranges [l, r]
    ```
for(int l=0; l<=y; ++l){
    for(int r=y; r<=H; ++r){
        const int d = r-l+1;
        if(i+d > n) continue;
        // find best option and sum
        int best = -inf;
        int sum = 0;
        for(int y2=l; y2<=r; ++y2){
            sum += v[y2][x];
        }
        best += sum;
        xmax(dp2[i+d][y], best);
    }
}
// update answer
swap_dp();
for(int y=0; y<=H; ++y){
    xmax(ans, dp[n][y]);
}
cout << ans << "\n";

80 Points $O(NWH^3)$

We can speed the previous solution up by a factor of $H$ if we loop over ranges $[l, r]$ and then consider all $y$ for which this range is valid. This way, we can separate the loops for $l$ and $y$, so we don’t recompute the sum and maximum over $[l, r]$ for every $y$. 

...  
int ans = -inf;
for(int y=0; y<=H; ++y){
    dp[0][y] = 0;
} 

for(int x=0; x<=W; ++x){
    for(int i=0; i<n; ++i){
        // iterate over vertical ranges $[l, r]$ 
        for(int l=0; l<=H; ++l){
            for(int r=l; r<=H; ++r){
                const int d = r-l+1;
                if(i+d > n) continue;
                // find best option and sum
                int best = -inf;
                int sum = 0;
                for(int y=l; y<=r; ++y){
                    sum += v[y][x];
                }
                best += sum;
                // consider the range $[l, r]$ for all valid $y$
                for(int y=l; y<=r; ++y){
                    xmax(dp2[i+d][y], best);
                }
            }
        }
    }
} 

swap_dp();
for(int y=0; y<=H; ++y){
    xmax(ans, dp[n][y]);
}
...
100 Points $O(NWH^2)$

To get full points, we need shave off another factor of $H$. When going from DP[$x$] to DP[$x+1$], let’s try to simultaneously consider all ranges $[l, r]$: Fix $x$ and let

$$Best[n][l][r] = \sum_{y=l}^{r} a_{y,x} + \max_{l \leq y \leq r} dp[x][n - (r - l + 1)][y]$$

We can compute $Best$ with $dp$ as follows:

$$Best[n][y][y] = dp[x][n - 1][y] + a_{y,x}$$
$$Best[n][l][r] = \max(Best[n - 1][l + 1][r] + a_{y,l}, Best[n - 1][l][r + 1] + a_{r,r})$$

This avoids the first loop over $y$ in the previous solution. We can similarly avoid the second loop: Fix $n$ and $x$ and let

$$Up[l][r] = \max_{r \leq l' \leq r} Best[n][l'][r']$$

This will then allows us to compute $DP[x+1]$ via

$$DP[x+1][n][y] = Up[y][y]$$

We can compute $Up$ (with some care to avoid out of bounds errors) via

$$Up[l][r] = \max(Up[l-1][r], Up[r][r+1], Best[n][l][r])$$

The total running time is $O(NWH^2)$.

**Note:** When implementing this, you should avoid excessive memory allocations and re-use the same vectors for every $x$. This makes the solution up to $5x$ faster. (But the time limit was lenient and the model solution only uses 1/3 of the time limit.)

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

template<typename T>
void xmax(T&a, T const&b){
    a = max(a, b);
}
constexpr int inf = 1e9;

signed main(){
    int H, W, n;
    cin >> H >> W >> n;
    vector<vector<int>> v(H, vector<int>(W));
    for(auto& e:v){
        for(auto& f:e){
            cin >> f;
        }
    }
    // re-use memory
    vector<vector<int>> dp(n+1, vector<int>(H, -inf));
    vector<vector<int>> best(n+1, vector<int>(H, vector<int>(H, -inf)));
    vector<vector<int>> up(H, vector<int>(H, -inf));
    int ans = -inf;
    for(int y=0; y<H; ++y){
        dp[0][y] = 0;
    }
    for(int x=0; x<W; ++x){
        // consider all ranges [1, r] in parallel
        for(int i=1; i<=n; ++i){
            for(int y=0; y<H; ++y){
                best[i][y][y] = dp[i-1][y] + v[y][x];
            }
        }
        for(int l=0; l+1; ++l){
```
for(int r=l+1; r<H; ++r){
    best[i][l][r] = max(best[i-1][l][r-1]+v[r][x], best[i-1][l+1][r]+v[l][x]);
}
}

for(int i=0; i<=n; ++i){
    for(int l=0; l<H; ++l){
        for(int r=H-1; r>=l; --r){
            up[l][r] = best[i][l][r];
            if(l>0) xmax(up[l][r], up[l-1][r]);
            if(r+1<H) xmax(up[l][r], up[l][r+1]);
        }
    }
}

for(int y=0; y<H; ++y){
    dp[i][y] = up[y][y];
}

for(int y=0; y<H; ++y){
    xmax(ans, dp[n][y]);
}
}

cout << ans << "\n";