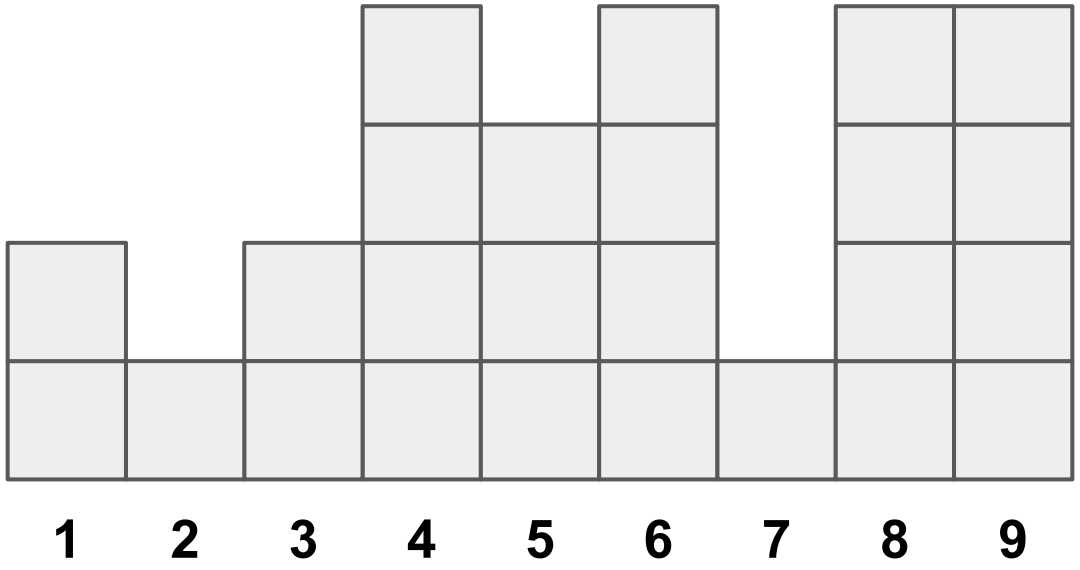
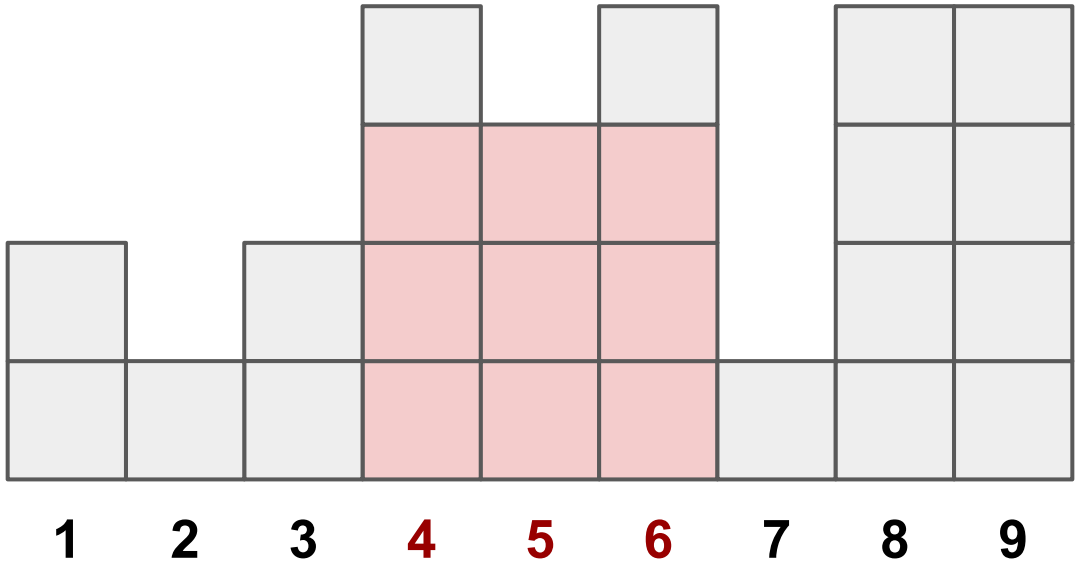
Aria maximă de sub histogramă

Alexandru Raul Todoran

Enunț: O histogramă este formată din n fâșii verticale de lățime 1 și înălțimi h_1, h_2, ..., h_n. Determinați dreptunghiul de arie maximă care poate fi încadrat sub histogramă.

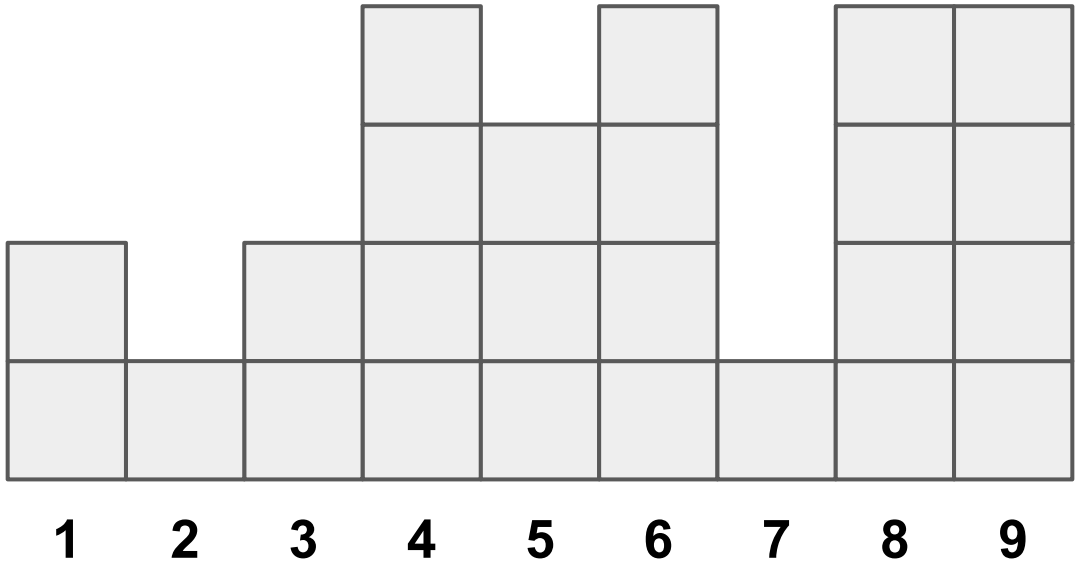


Enunț: O histogramă este formată din n fâșii verticale de lățime 1 și înălțimi h_1, h_2, ..., h_n. Determinați dreptunghiul de arie maximă care poate fi încadrat sub histogramă.

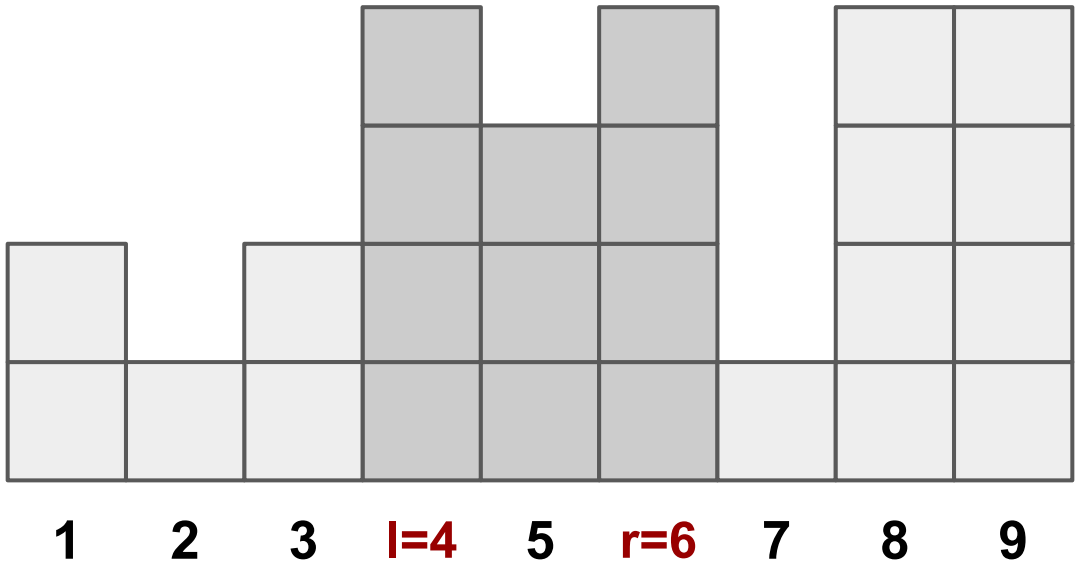


Arie maximă: 3x3=9

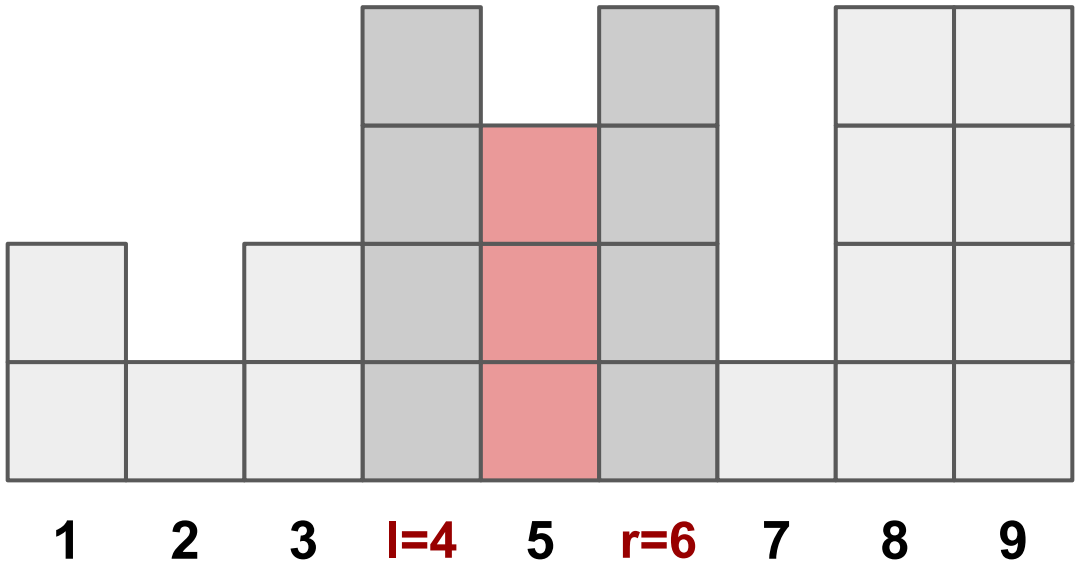
Soluții ineficiente



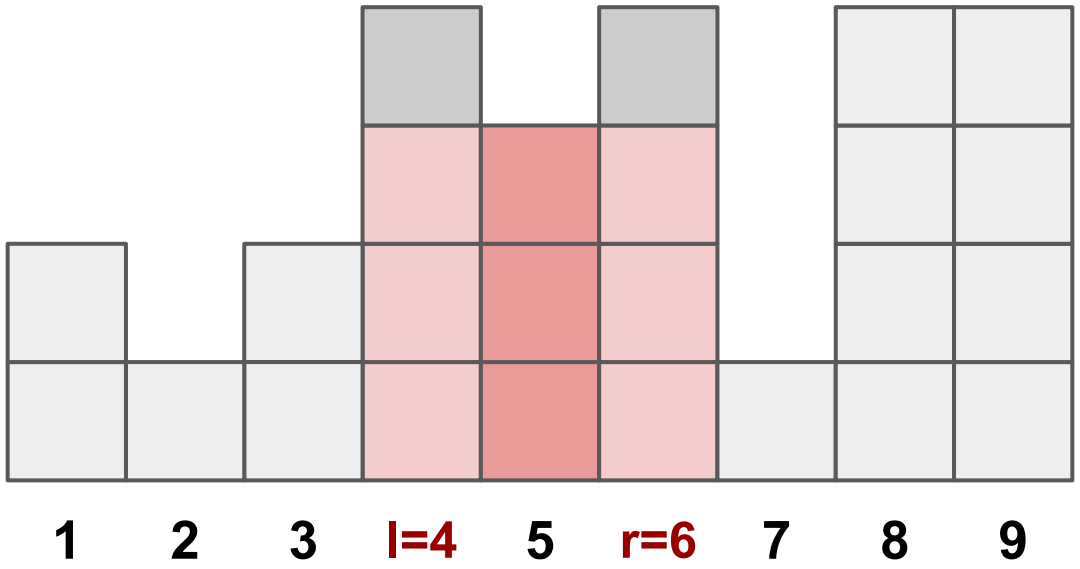
Soluția 1: Parcurgem folosind două for-uri toate perechile (l,r) între care s-ar putea afla dreptunghiul de arie maximă.



Soluția 1: Parcurgem folosind două for-uri toate perechile (l,r) între care s-ar putea afla dreptunghiul de arie maximă. Pentru fiecare dintre aceste perechi, parcurgem cu un alt for fâșiile din intervalul [l,r] și determinăm fâșia k de înălțime minimă, care determină totodată înălțimea dreptunghiului.



Soluția 1: Parcurgem folosind două for-uri toate perechile (l,r) între care s-ar putea afla dreptunghiul de arie maximă. Pentru fiecare dintre aceste perechi, parcurgem cu un alt for fâșiile din intervalul [l,r] și determinăm fâșia k de înălțime minimă, care determină totodată înălțimea dreptunghiului. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.

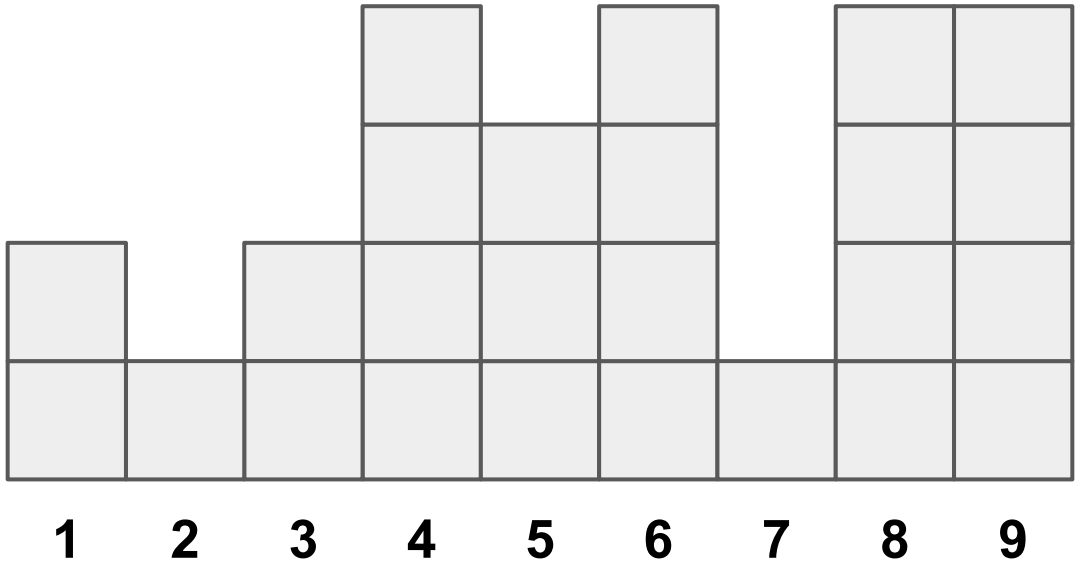


```cpp
#include <bits/stdc++.h>
using namespace std;
const int N_MAX = 400;
int n;
int h[N_MAX + 2];
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> h[i];
    }
    int mx = 0;
    for (int l = 1; l <= n; l++) {
        for (int r = l; r <= n; r++) {
            int k = l;
            for (int i = l + 1; i <= r; i++) {
                if (h[i] < h[k]) {
                    k = i;
                }
            }
            mx = max(mx, h[k] * (r - l + 1));
        }
    }
    cout << mx << "\n";
    return 0;
}
```

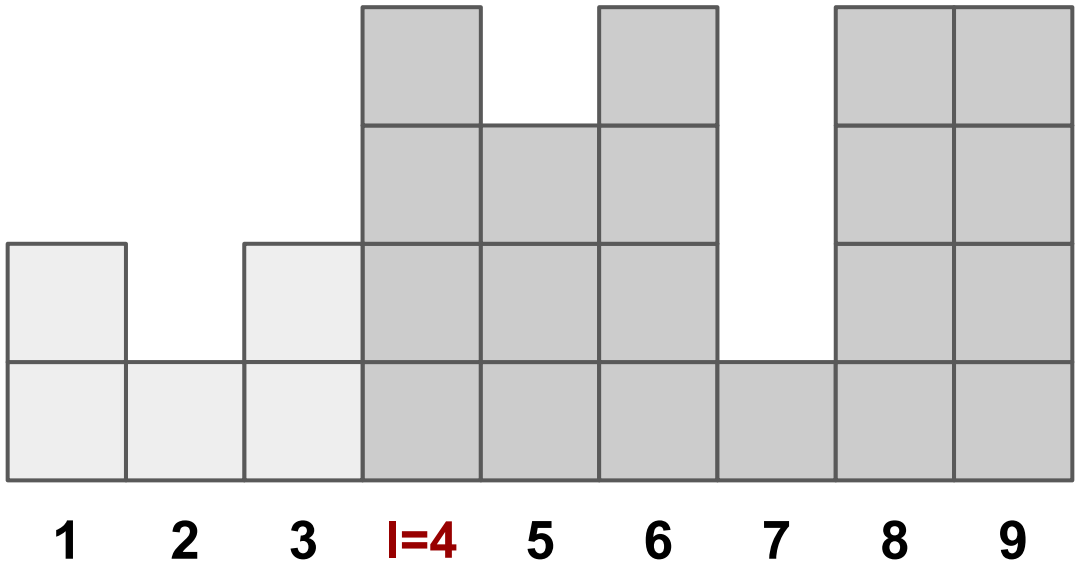
```cpp
#include <bits/stdc++.h>
using namespace std;
const int N_MAX = 400;
int n;
int h[N_MAX + 2];
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> h[i];
    }
    int mx = 0;
    for (int l = 1; l <= n; l++) {
        for (int r = l; r <= n; r++) {
            int k = l;
            for (int i = l + 1; i <= r; i++) {
                if (h[i] < h[k]) {
                    k = i;
                }
            }
            mx = max(mx, h[k] * (r - l + 1));
        }
    }
    cout << mx << "\n";
    return 0;
}
```

Soluția 1 face $O(n^3)$ operații pentru a ajunge la rezultat. Așadar, este eficientă pentru $n \leq 400$.

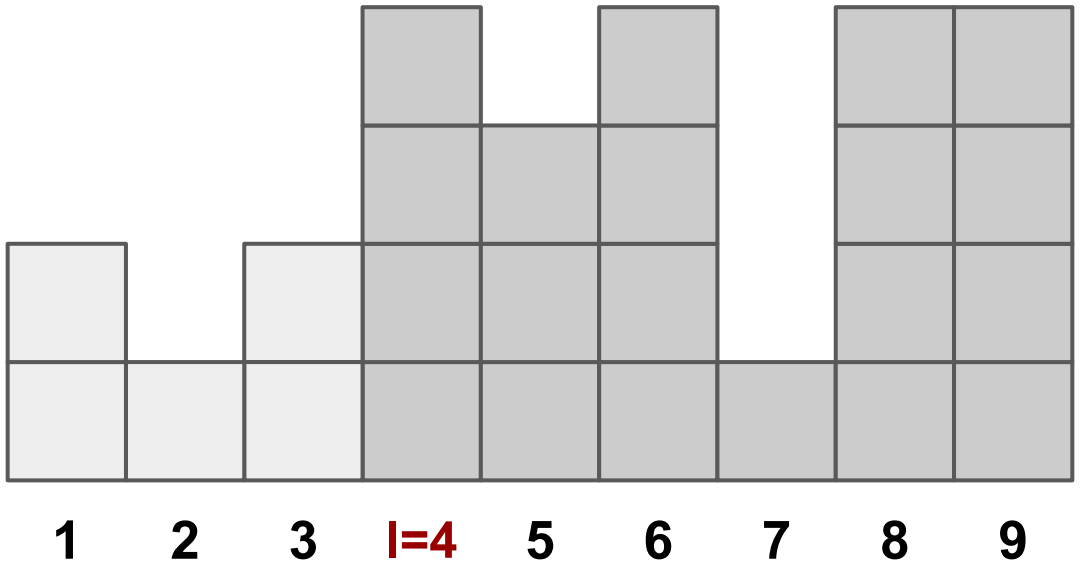
Soluții ineficiente



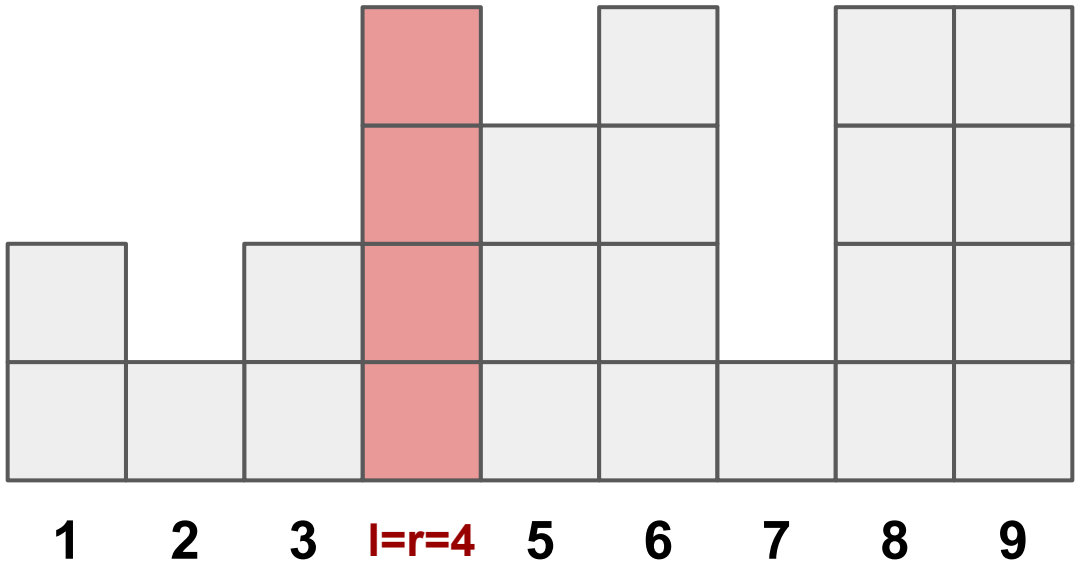
Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă.



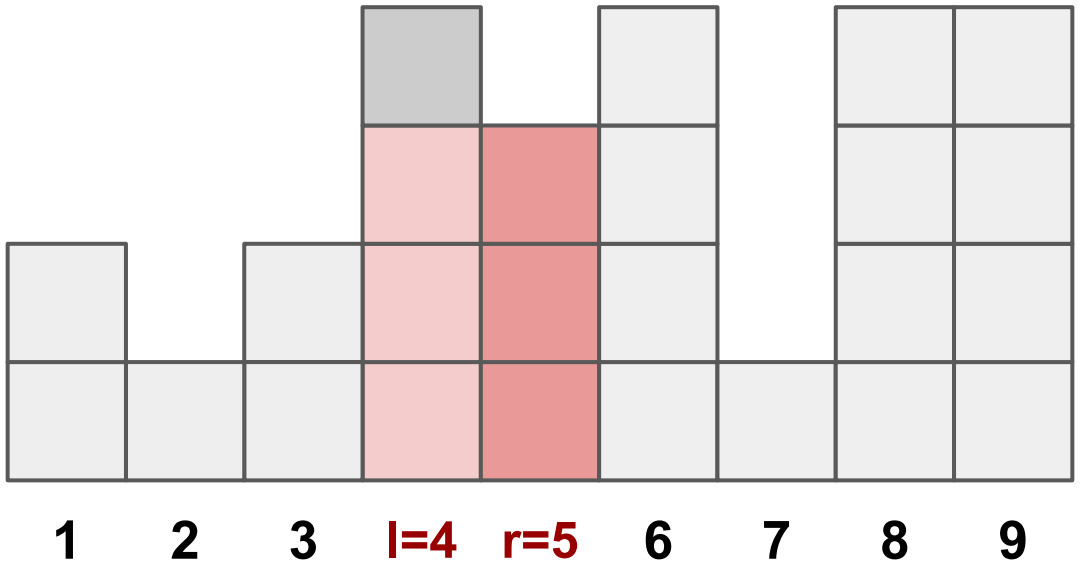
Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă. Parcurgem un for toate pozițiile r la care s-ar putea termina dreptunghiul și menținem fâșia k de înălțime minimă. Aria va fi $h_k(r-l+1)$.



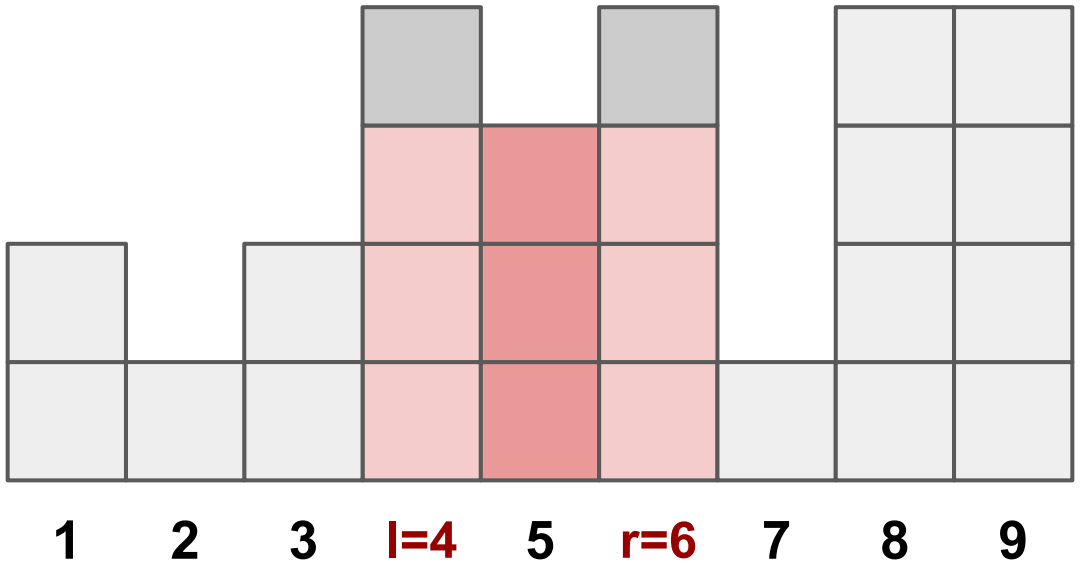
Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă. Parcurgem un for toate pozițiile r la care s-ar putea termina dreptunghiul și menținem fâșia k de înălțime minimă. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.



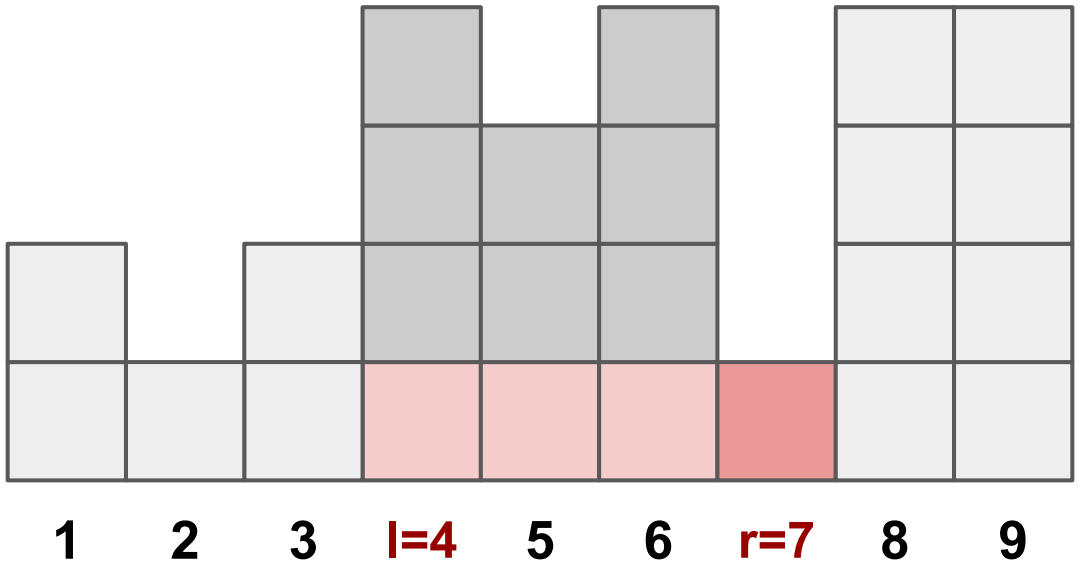
Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă. Parcurgem un for toate pozițiile r la care s-ar putea termina dreptunghiul și menținem fâșia k de înălțime minimă. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.



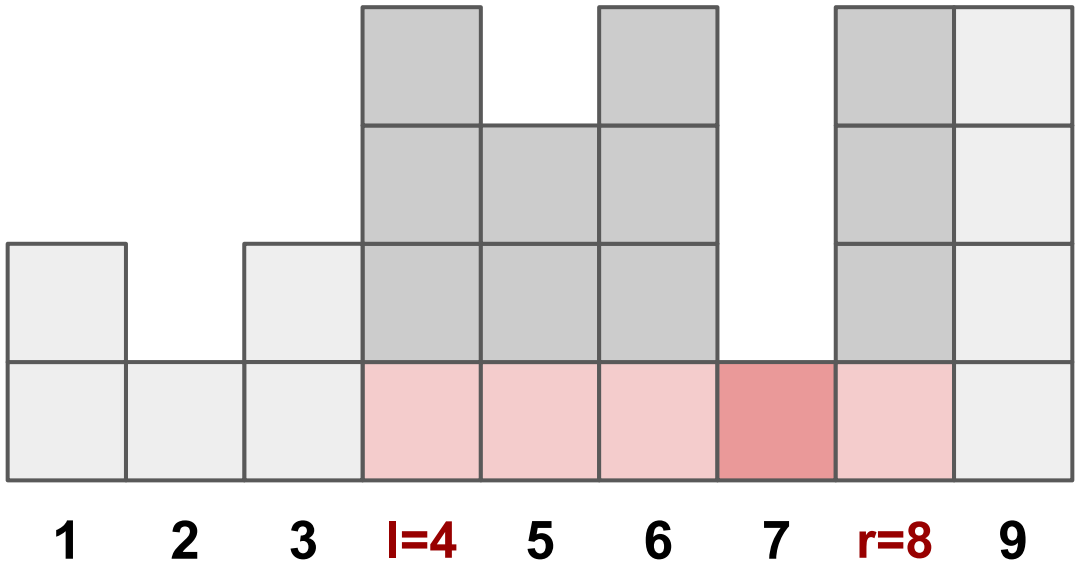
Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă. Parcurgem un for toate pozițiile r la care s-ar putea termina dreptunghiul și menținem fâșia k de înălțime minimă. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.



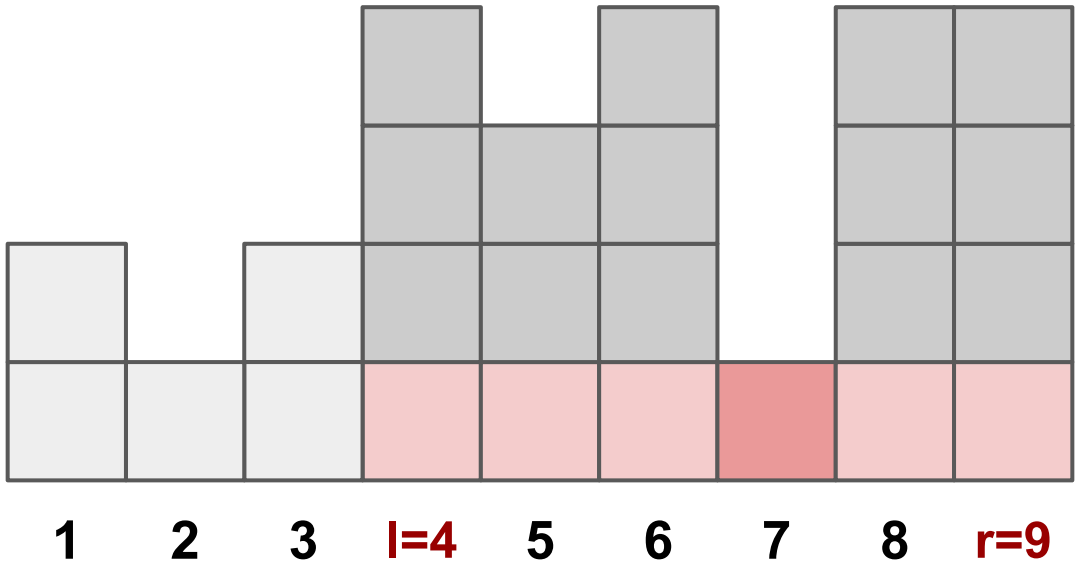
Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă. Parcurgem un for toate pozițiile r la care s-ar putea termina dreptunghiul și menținem fâșia k de înălțime minimă. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.



Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă. Parcurgem un for toate pozițiile r la care s-ar putea termina dreptunghiul și menținem fâșia k de înălțime minimă. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.



Soluția 2: Parcurgem folosind un for toate pozițiile l la care ar putea începe dreptunghiul de arie maximă. Parcurgem un for toate pozițiile r la care s-ar putea termina dreptunghiul și menținem fâșia k de înălțime minimă. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.

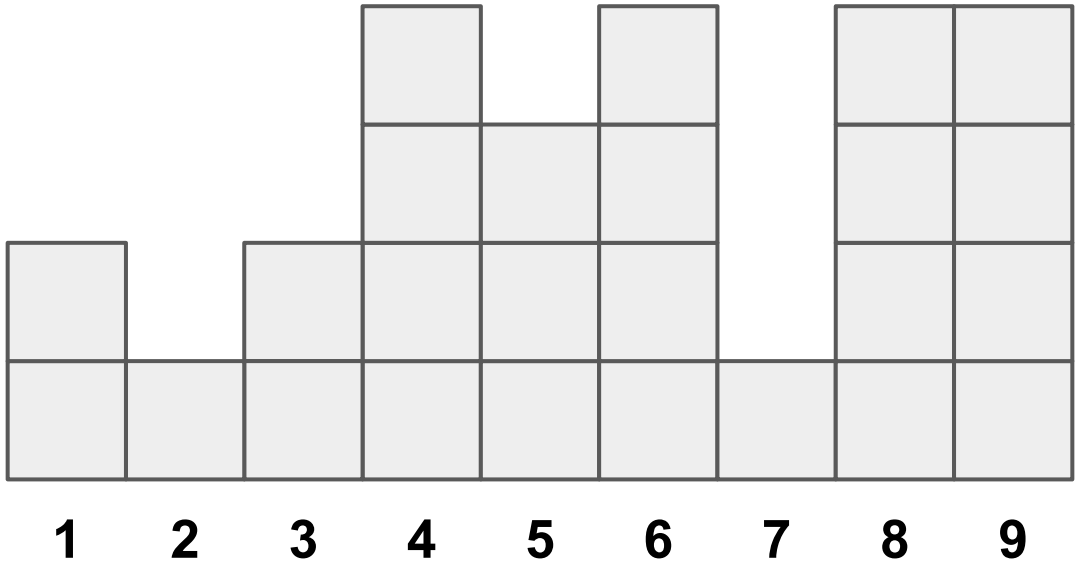


```cpp
#include <bits/stdc++.h>
using namespace std;
const int N_MAX = 5000;
int n;
int h[N_MAX + 2];
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> h[i];
    }
    int mx = 0;
    for (int l = 1; l <= n; l++) {
        int k = l;
        for (int r = l; r <= n; r++) {
            if (h[r] < h[k]) {
                k = r;
            }
            mx = max(mx, h[k] * (r - l + 1));
        }
    }
    cout << mx << "\n";
    return 0;
}
```

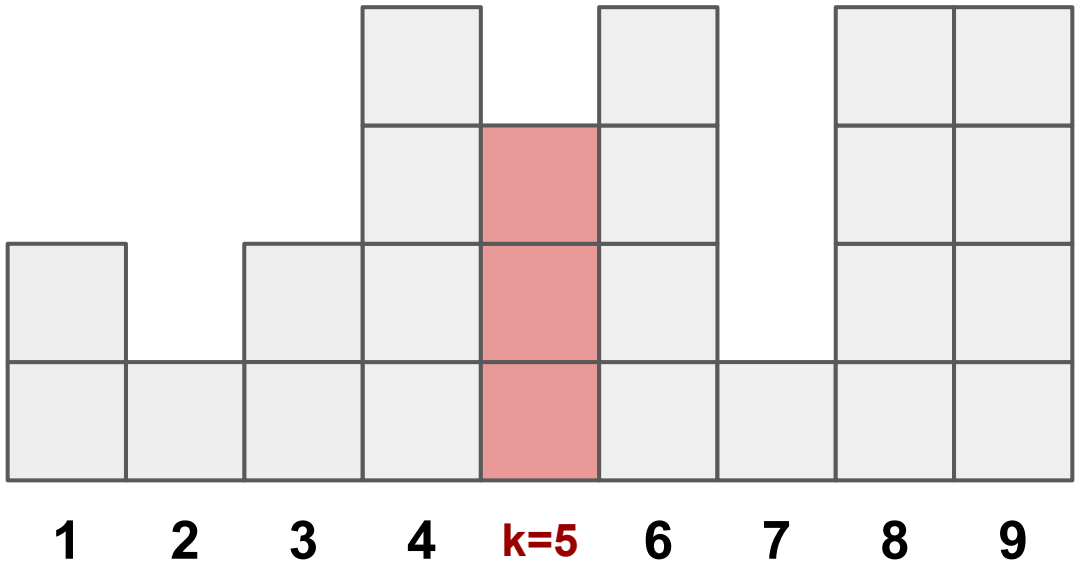
```cpp
#include <bits/stdc++.h>
using namespace std;
const int N_MAX = 5000;
int n;
int h[N_MAX + 2];
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> h[i];
    }
    int mx = 0;
    for (int l = 1; l <= n; l++) {
        int k = l;
        for (int r = l; r <= n; r++) {
            if (h[r] < h[k]) {
                k = r;
            }
            mx = max(mx, h[k] * (r - l + 1));
        }
    }
    cout << mx << "\n";
    return 0;
}
```

Soluția 2 face $O(n^2)$ operații pentru a ajunge la rezultat. Așadar, este eficientă pentru $n \leq 5000$.

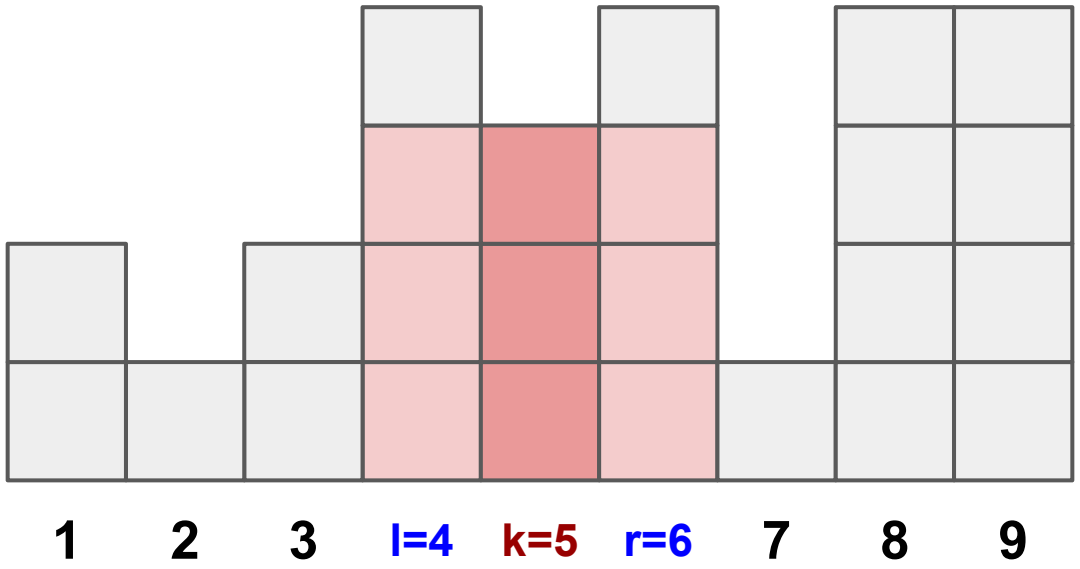
Soluța eficientă



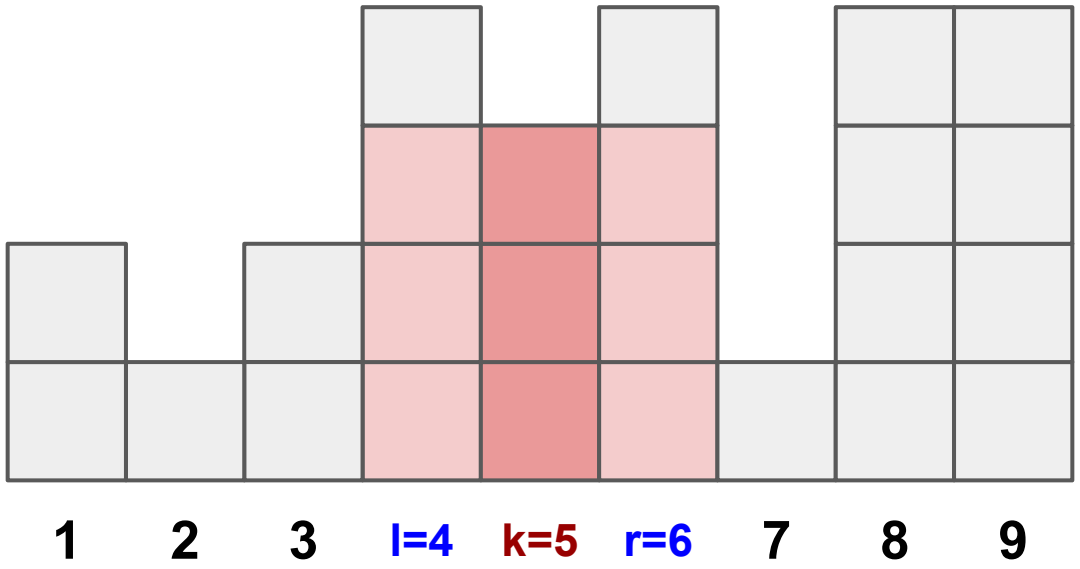
Soluția 3: Fiecare dreptunghi candidat conține o fâșie de înălțime minimă.
Parcurgem cu un for toate pozițiile k unde s-ar putea afla acea fâșie minimă.



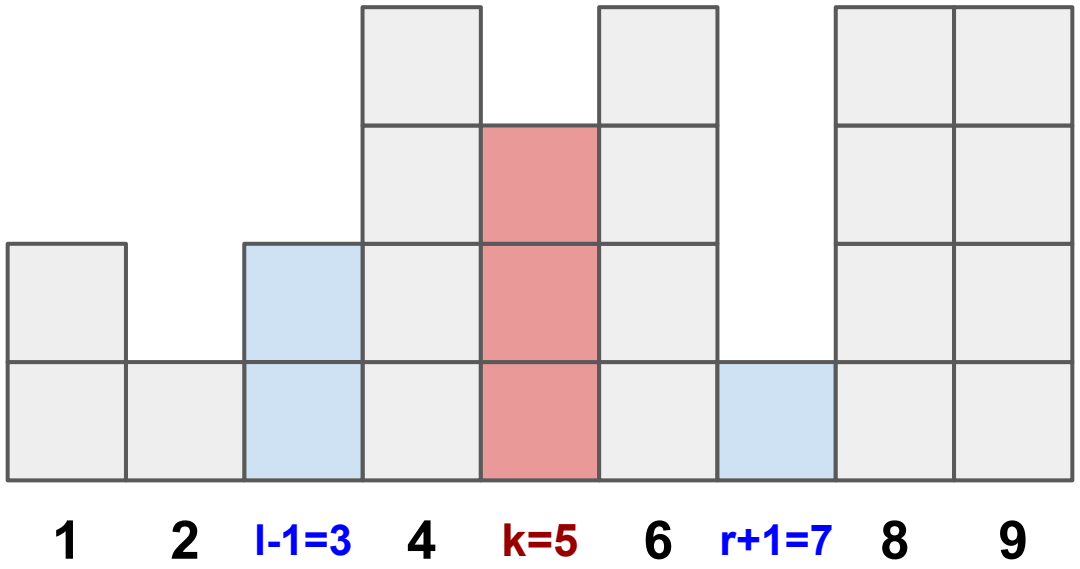
Soluția 3: Fiecare dreptunghi candidat conține o fâșie de înălțime minimă. Parcurgem cu un for toate pozițiile k unde s-ar putea afla acea fâșie minimă. Pentru fiecare, determinăm intervalul maximal [l,r] pe care această fâșie este minimă. Aria va fi $h_k(r-l+1)$. Dintre toate aceste arii, o alegem pe cea maximă.



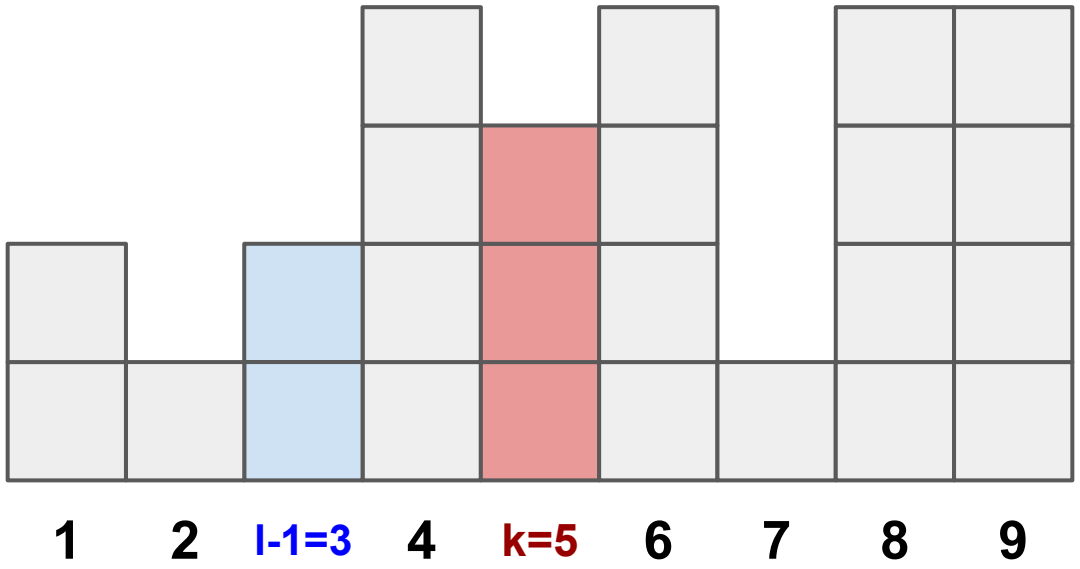
Cum determinăm eficient intervalul [l,r] pe care fâșia k este minimă?



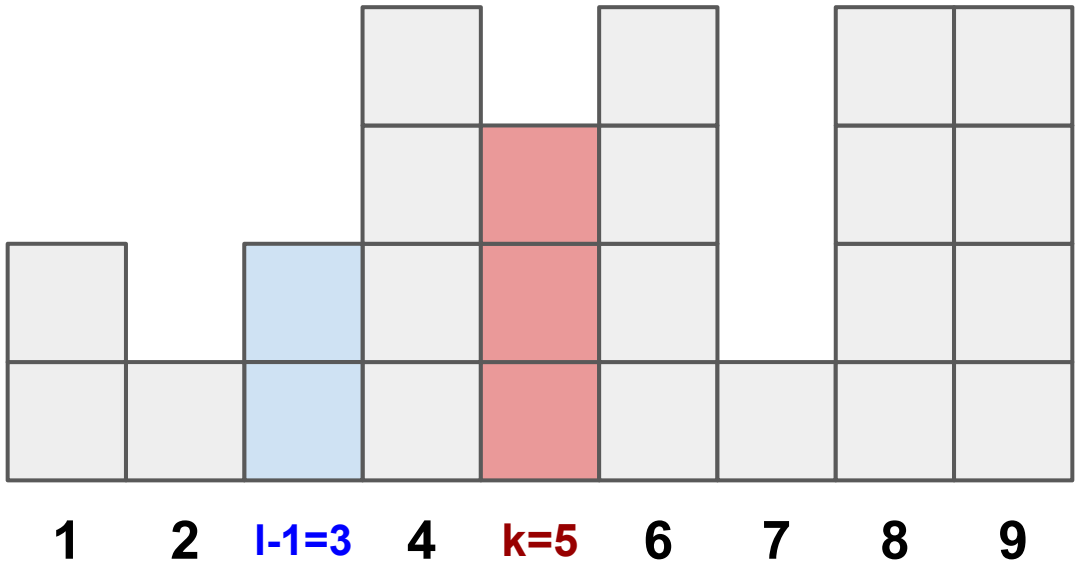
Observăm că fâșiile l-1 și r+1 sunt cele mai apropiate fâșii strict mai mici decât fâșia k, la stânga și la dreapta.



Observăm că fâșiile l-1 și r+1 sunt cele mai apropiate fâșii strict mai mici decât fâșia k, la stânga și la dreapta. Determinarea celor două fâșii se face similar, așa că ne concentrăm doar pe fâșia din stânga.

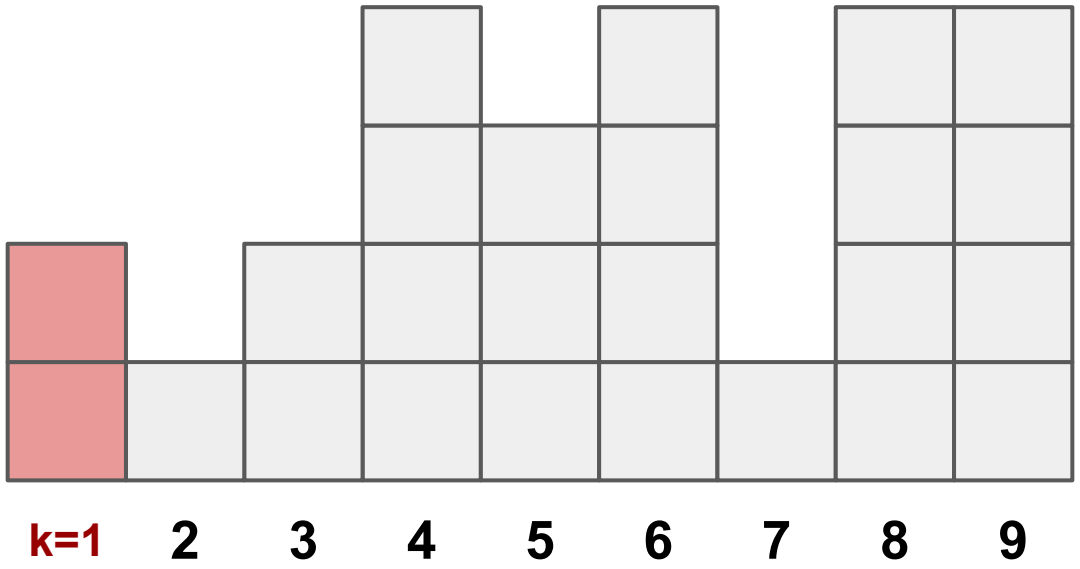


Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.



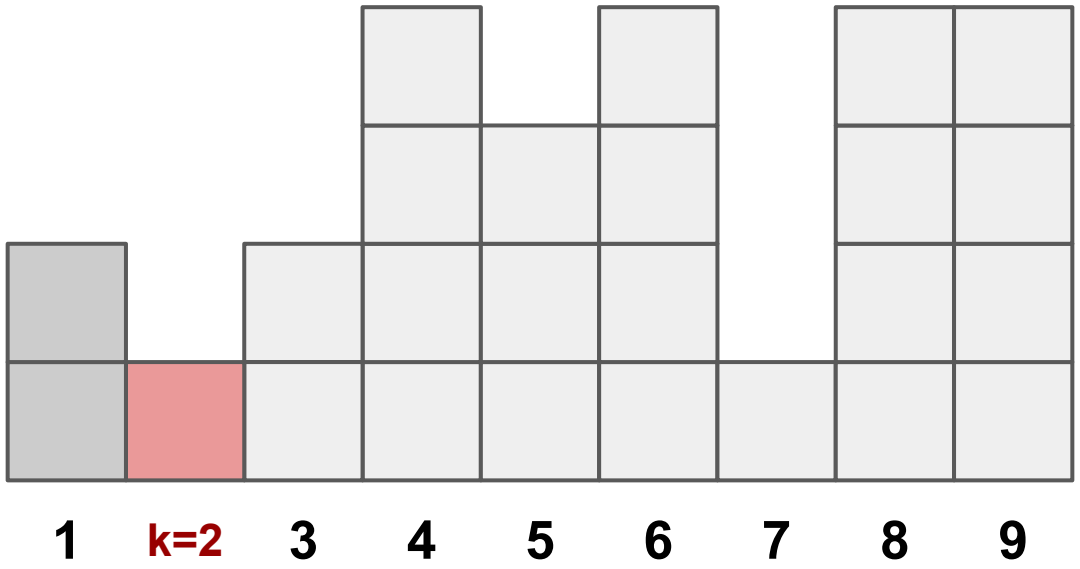
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=1. La stânga nu există nicio altă fâșie, așa că l=1.



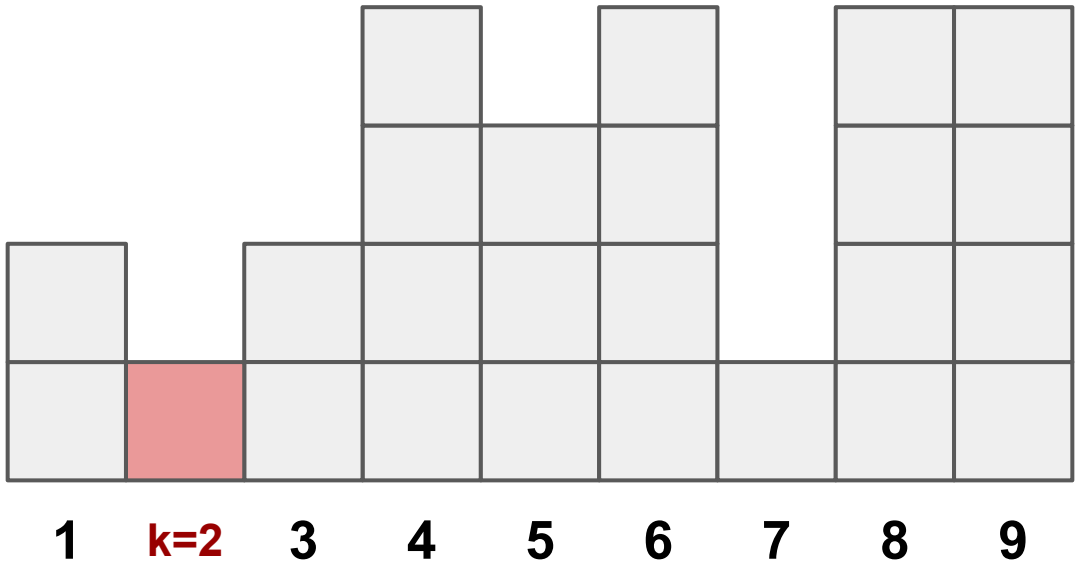
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=2. Fâșia din stânga nu este mai mică decât fâșia k, așa că l=1.



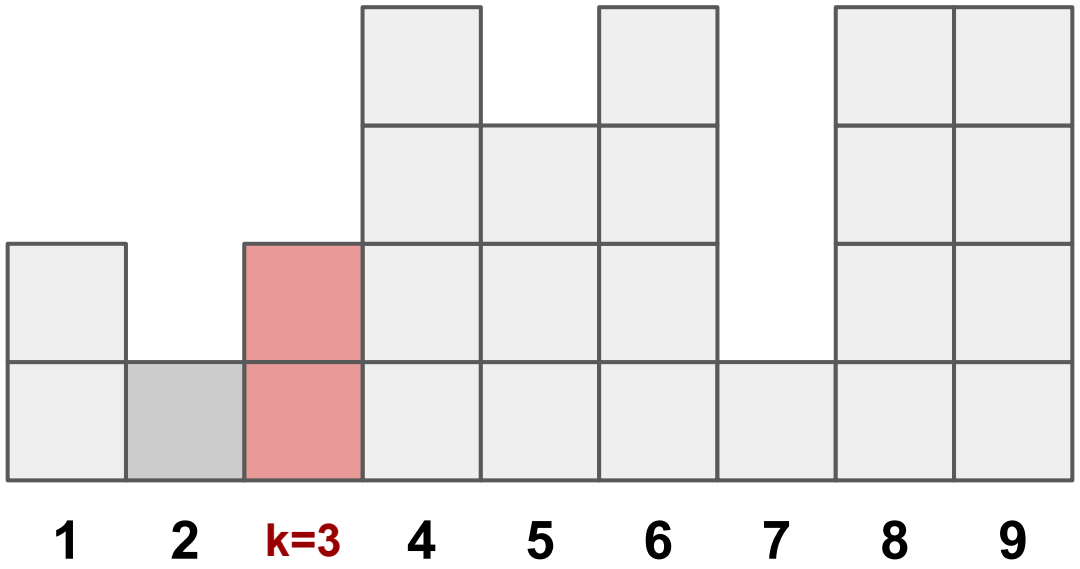
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=2. Fâșia 1 nu este mai mică decât fâșia 2, așa că l=1. De asemenea, fâșia 1 nu mai trebuie să fie considerată de acum înainte.



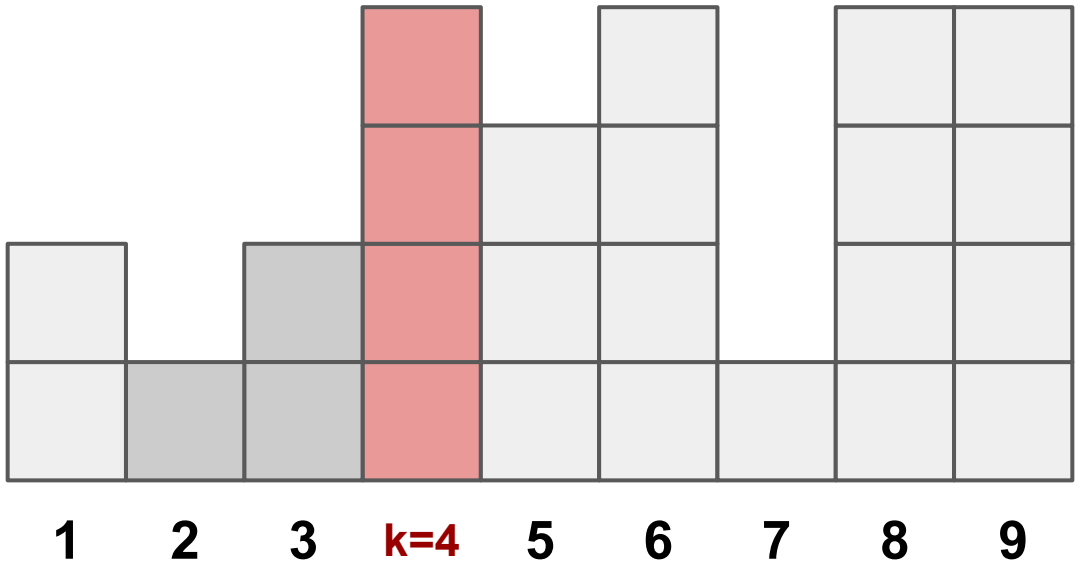
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=3. Fâșia 2 este mai mică decât fâșia 3, așa că l=2+1=3.



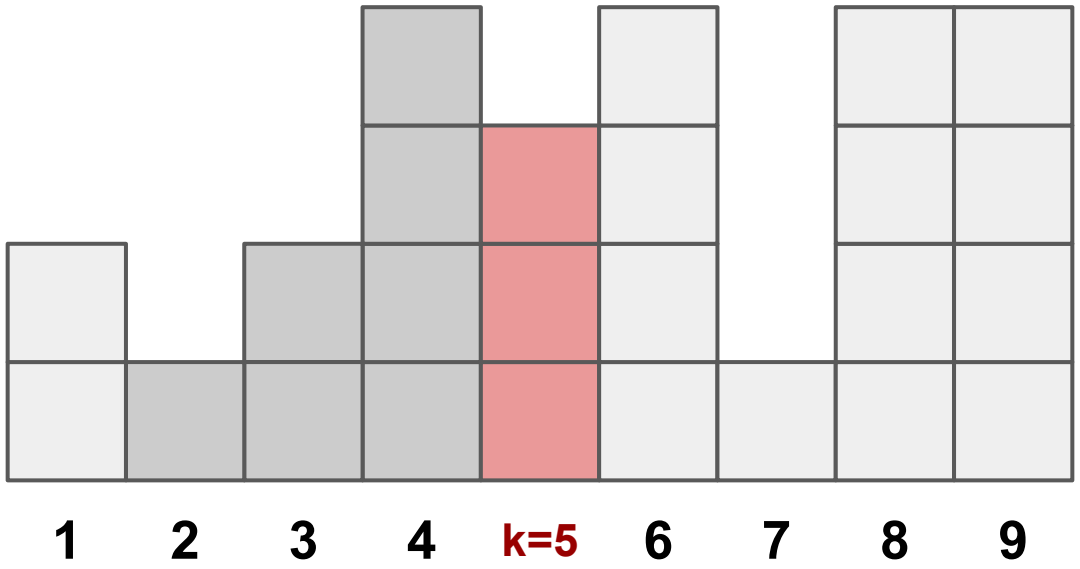
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=4. Fâșia 3 este mai mică decât fâșia 4, așa că l=3+1=4.



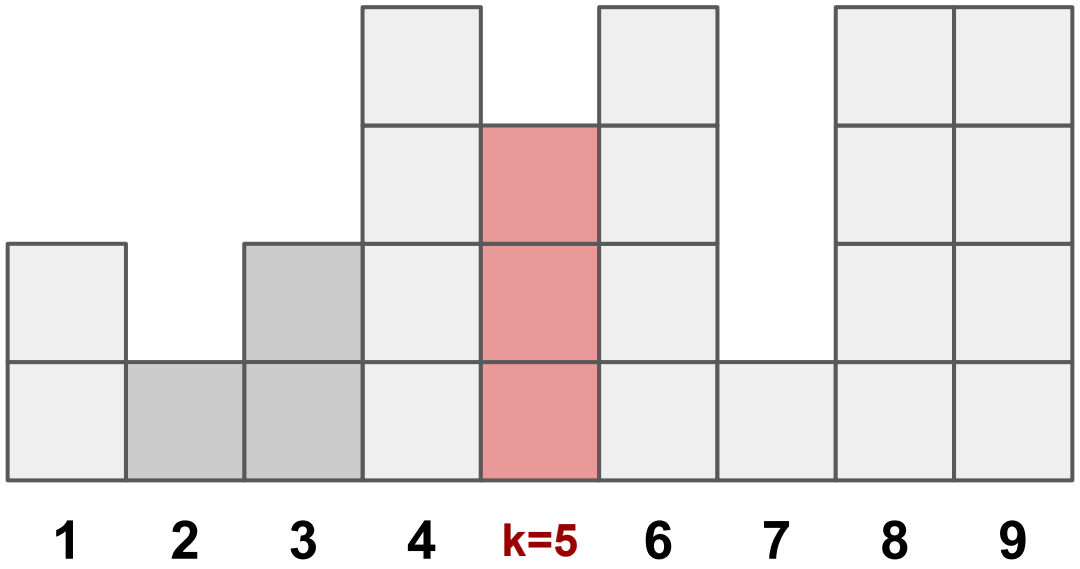
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=5. Fâșia 4 nu este mai mică decât fâșia 5, așa că nu mai trebuie considerată.



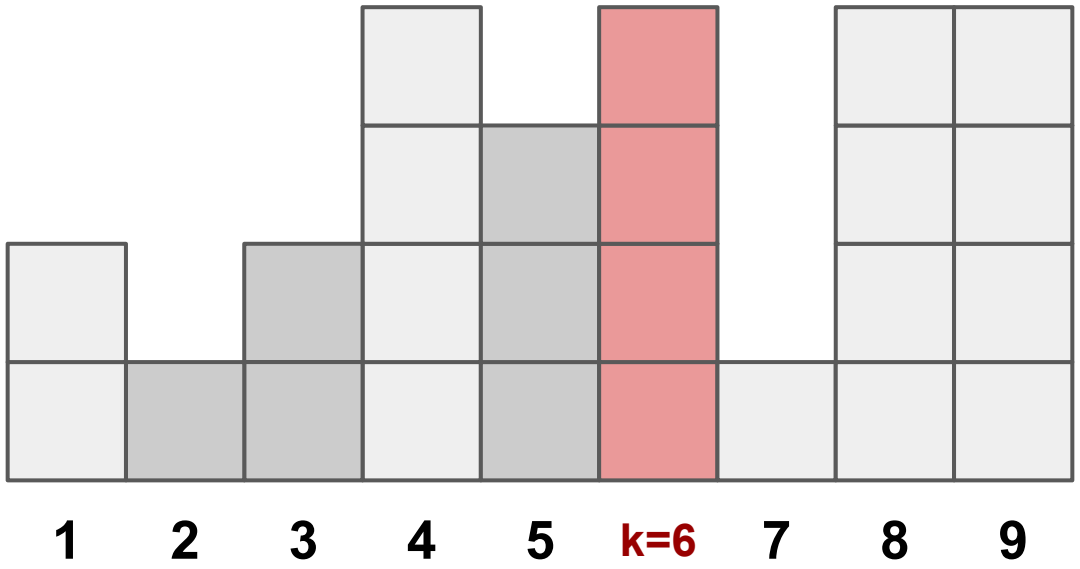
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=5. Fâșia 4 nu este mai mică decât fâșia 5, așa că nu mai trebuie considerată. Fâșia 3 în schimb, este mai mică decât fâșia 5, așa că l=3+1=4.



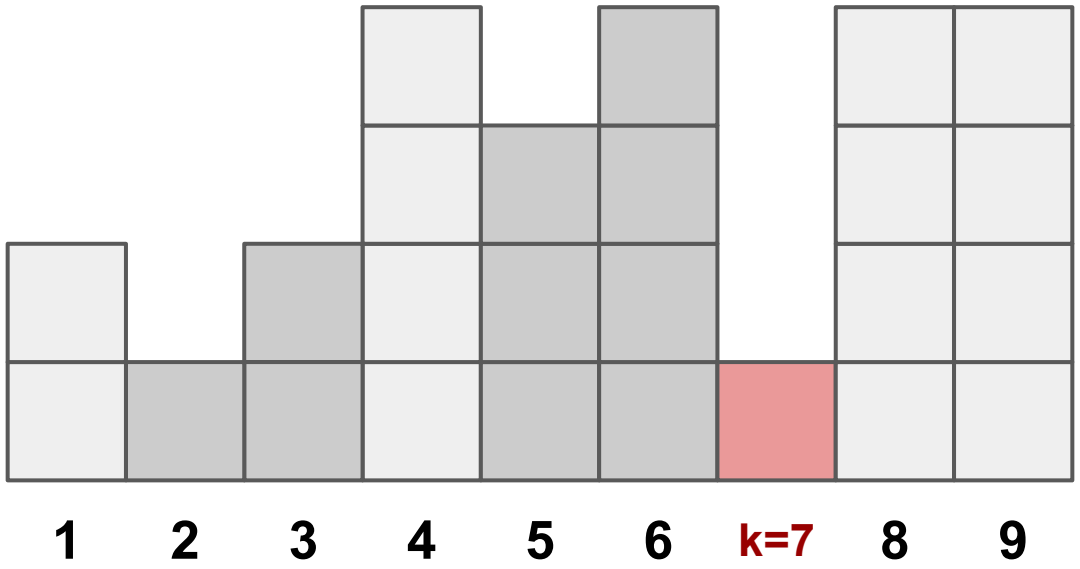
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=6. Fâșia 5 este mai mică decât fâșia 6, așa că l=5+1=6.



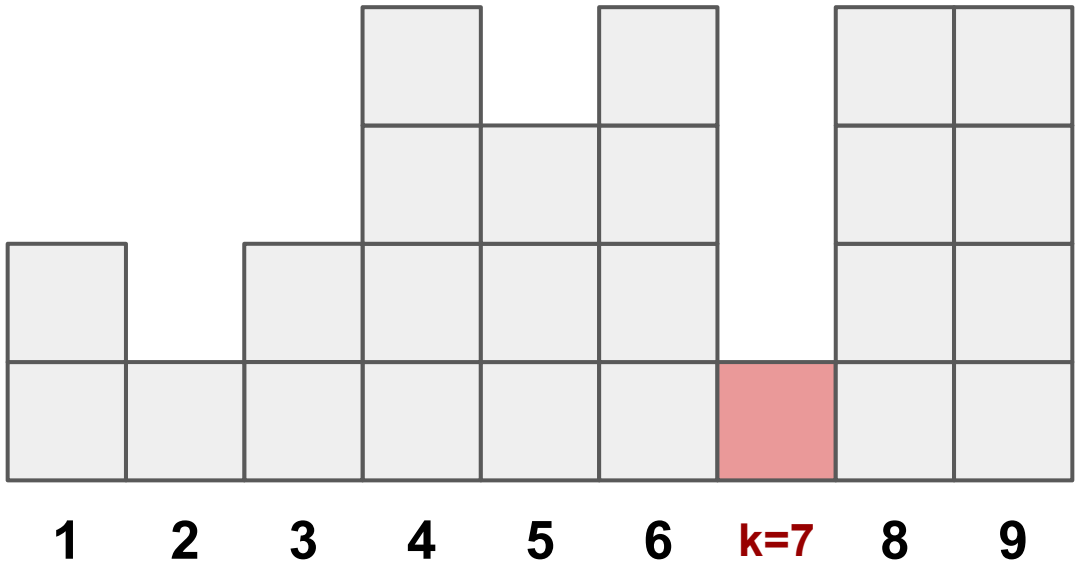
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=7. Niciuna dintre fâșiile 2, 3, 5 sau 6 nu sunt mai mici decât fâșia 7. Așadar, nu vor mai fi considerate în continuare, iar l=1.



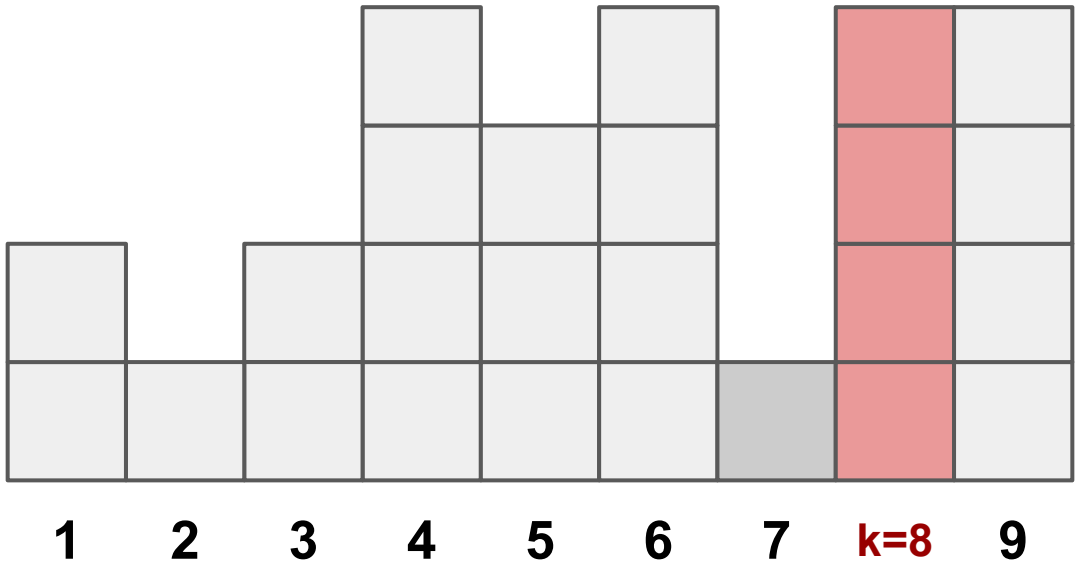
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=7. Niciuna dintre fâșiile 2, 3, 5 sau 6 nu sunt mai mici decât fâșia 7. Așadar, nu vor mai fi considerate în continuare, iar l=1.



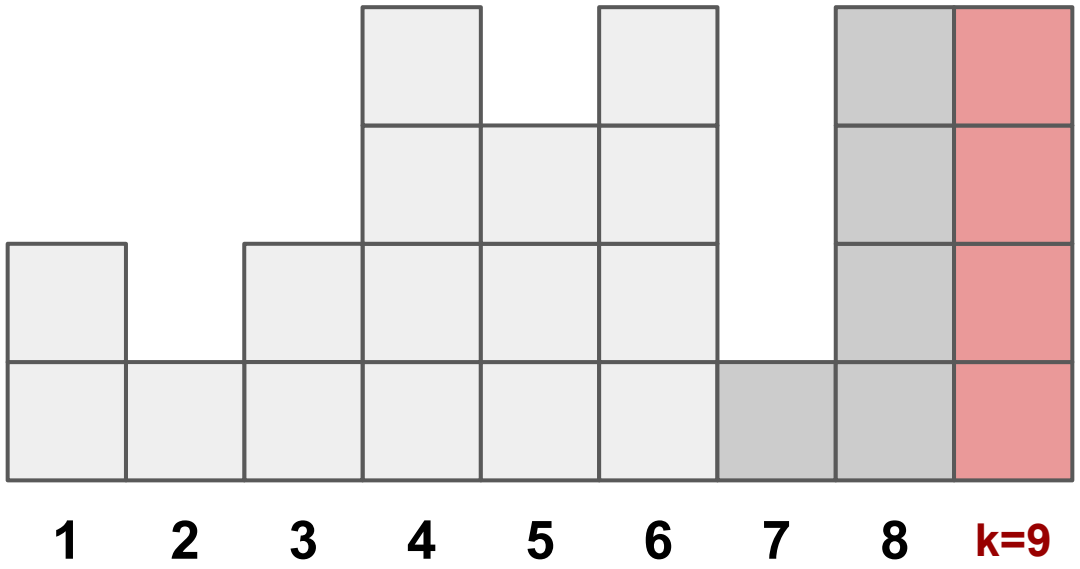
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=8. Fâșia 7 este mai mică decât fâșia 8, așa că l=7+1=8.



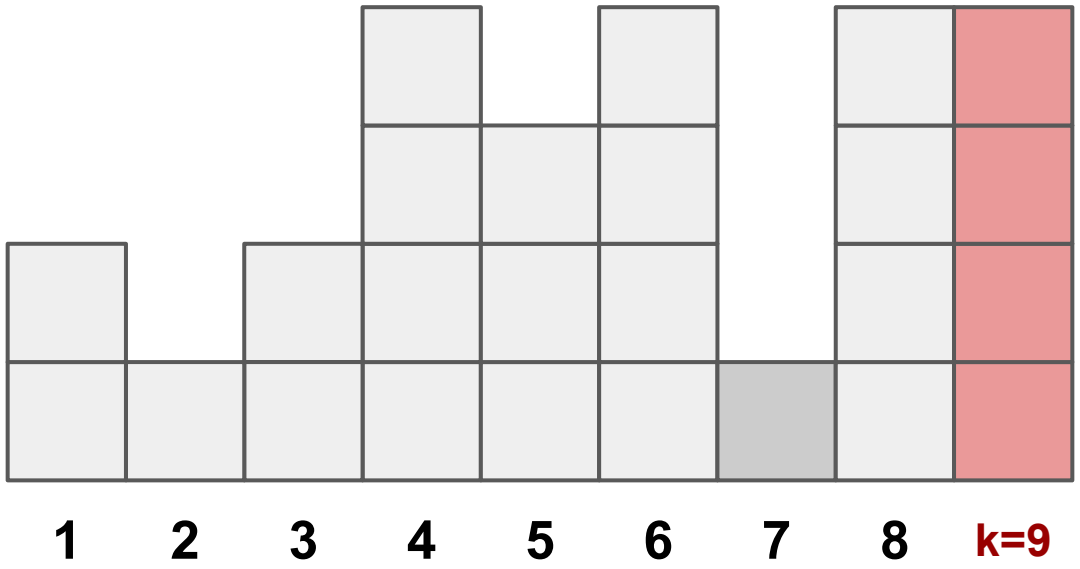
Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=9. Fâșia 8 nu este mai mică decât fâșia 9, așa că nu mai trebuie considerată.

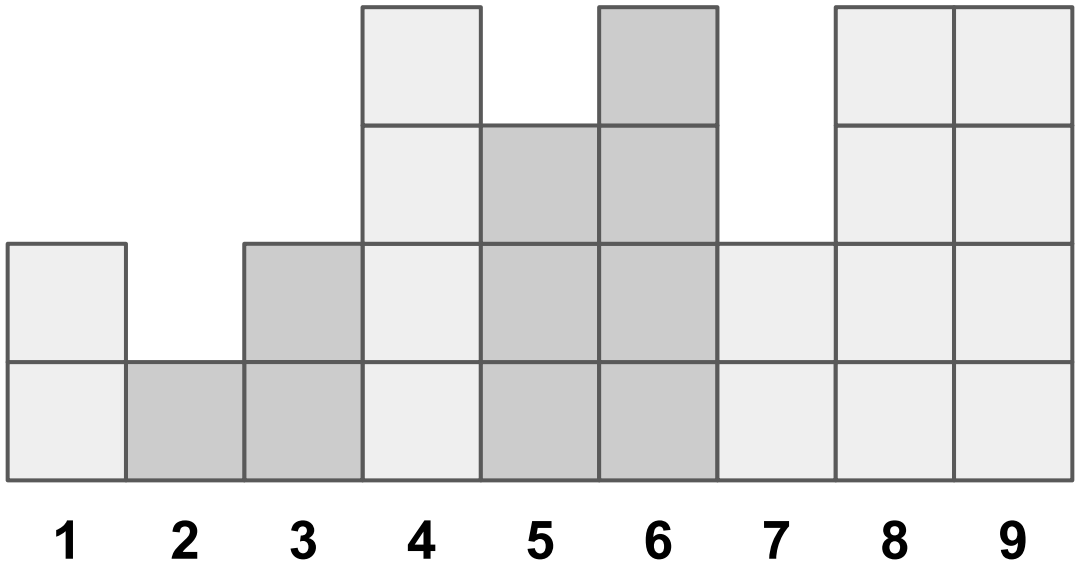


Este greu să determinăm fâșia l-1 independent pentru fiecare fâșie k, așa că vom parcurge cu un for fâșiile k de la prima la ultima.

k=9. Fâșia 8 nu este mai mică decât fâșia 9, așa că nu mai trebuie considerată. Fâșia 7 în schimb, este mai mică decât fâșia 9, așa că l=7+1=8.

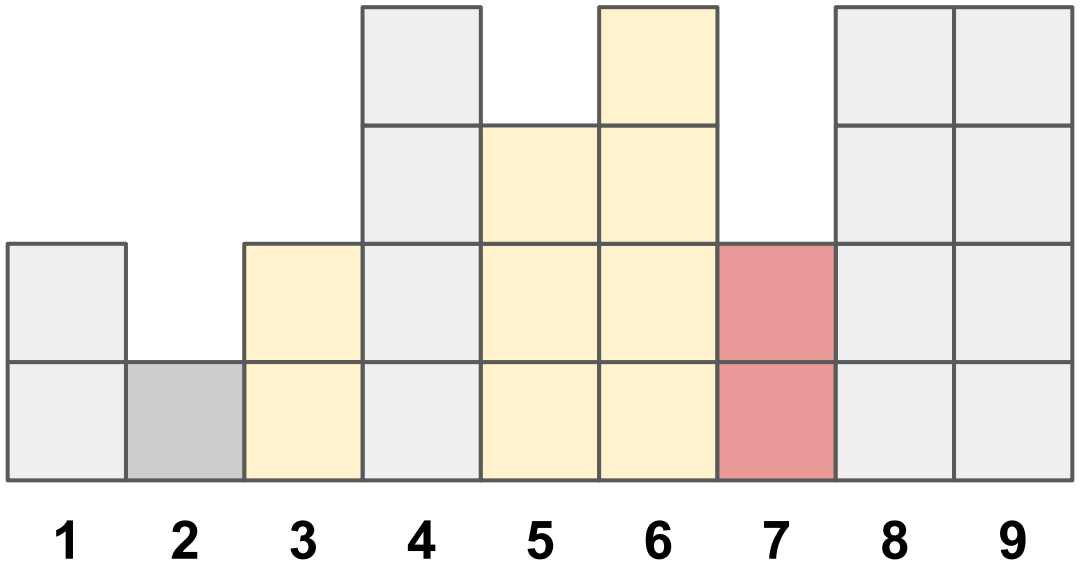


Fâșiile luate în considerare la orice moment dat vor forma un șir stir crescător. Pentru a menține acest șir, vom folosi o stivă (un vector în care introducem și ștergem mereu de la final).



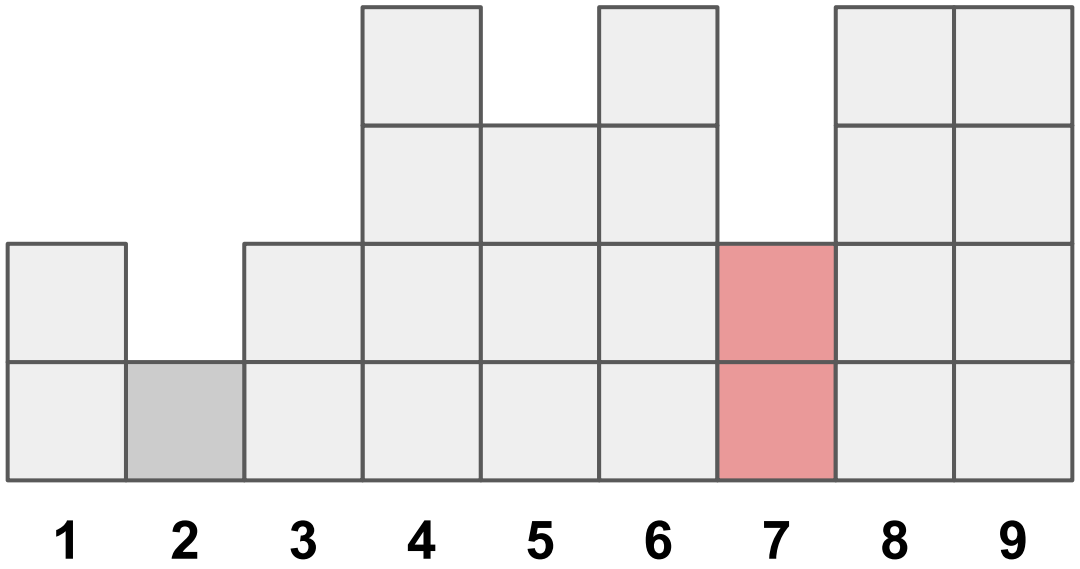
Fâșiile luate în considerare la orice moment dat vor forma un șir stir crescător. Pentru a menține acest șir, vom folosi o stivă (un vector în care introducem și ștergem mereu de la final).

Înainte să introducem o nouă fâșie în stivă, ștergem mai întâi toate fâșiile mai mici sau egale din stivă.

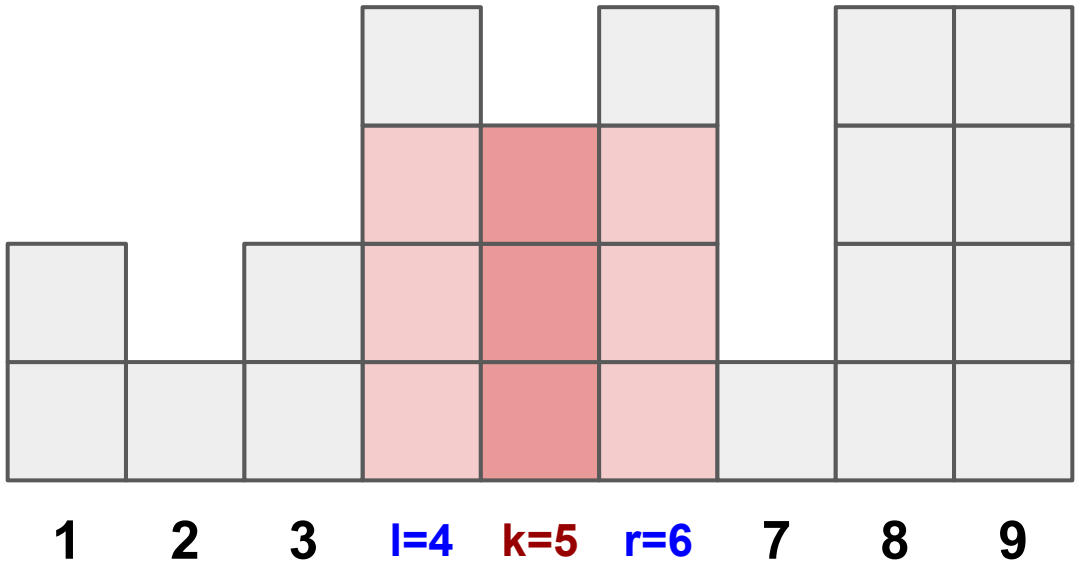


Fâșiile luate în considerare la orice moment dat vor forma un șir stir crescător. Pentru a menține acest șir, vom folosi o stivă (un vector în care introducem și ștergem mereu de la final).

Înainte să introducem o nouă fâșie în stivă, ștergem mai întâi toate fâșiile mai mici sau egale din stivă. Ultima fâșie din stivă va fi mereu cea pe care o căutăm.



Odată determinat intervalul [l,r] pentru fiecare fâșie k, putem determina aria maximă de sub histogramă.



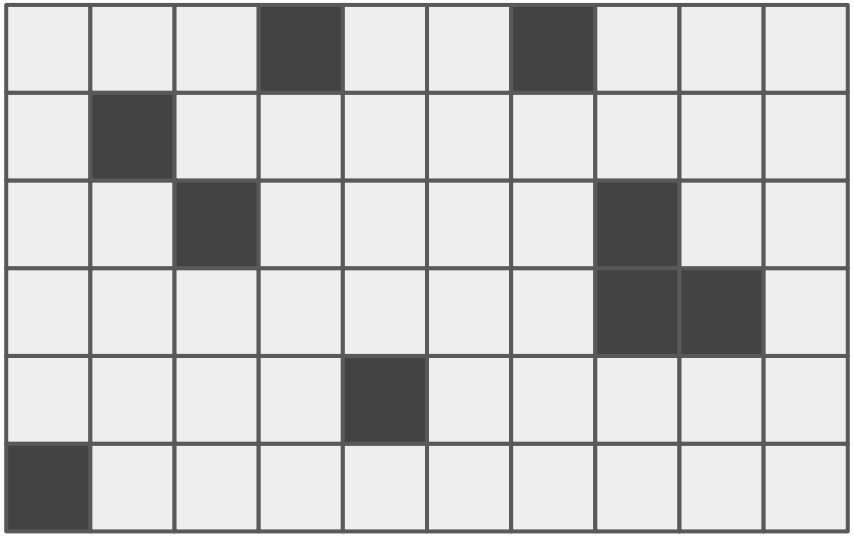
```cpp
#include <bits/stdc++.h>
using namespace std;
const int N_MAX = 1000000;
int n;
int h[N_MAX + 2];
int st[N_MAX + 2], cnt;
int L[N_MAX + 2], R[N_MAX + 2];
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> h[i];
    }
    for (int i = 1; i <= n; i++) {
        while (cnt > 0 && h[st[cnt]] >= h[i]) {
            cnt--;
        }
        L[i] = st[cnt] + 1;
        cnt++; st[cnt] = i;
    }
    cnt = 0; st[0] = n + 1;
    for (int i = n; i >= 1; i--) {
        while (cnt > 0 && h[st[cnt]] >= h[i]) {
            cnt--;
        }
        R[i] = st[cnt] - 1;
        cnt++; st[cnt] = i;
    }
    long long mx = 0;
    for (int i = 1; i <= n; i++) {
        mx = max(mx, (long long) h[i] * (R[i] - L[i] + 1));
    }
    cout << mx << "\n";
    return 0;
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N_MAX = 1000000;
int n;
int h[N_MAX + 2];
int st[N_MAX + 2], cnt;
int L[N_MAX + 2], R[N_MAX + 2];
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> h[i];
    }
    for (int i = 1; i <= n; i++) {
        while (cnt > 0 && h[st[cnt]] >= h[i]) {
            cnt--;
        }
        L[i] = st[cnt] + 1;
        cnt++; st[cnt] = i;
    }
    cnt = 0; st[0] = n + 1;
    for (int i = n; i >= 1; i--) {
        while (cnt > 0 && h[st[cnt]] >= h[i]) {
            cnt--;
        }
        R[i] = st[cnt] - 1;
        cnt++; st[cnt] = i;
    }
    long long mx = 0;
    for (int i = 1; i <= n; i++) {
        mx = max(mx, (long long) h[i] * (R[i] - L[i] + 1));
    }
    cout << mx << "\n";
    return 0;
}
```

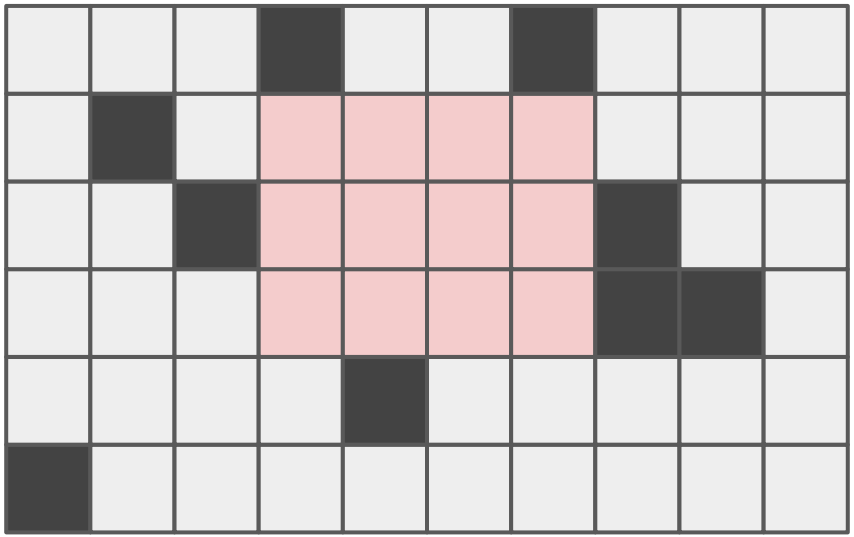
Soluția 3 face O(n) operații pentru a ajunge la rezultat. Așadar, este eficientă pentru n ≤ 1000000, sau chiar mai mare.

Submatrice de arie maximă

Enunț: Se dă o matrice cu valori de 1 și 0. Celulele care conțin valoarea 1 sunt blocate, restul sunt libere. Să se determine submatricea liberă de arie maximă.

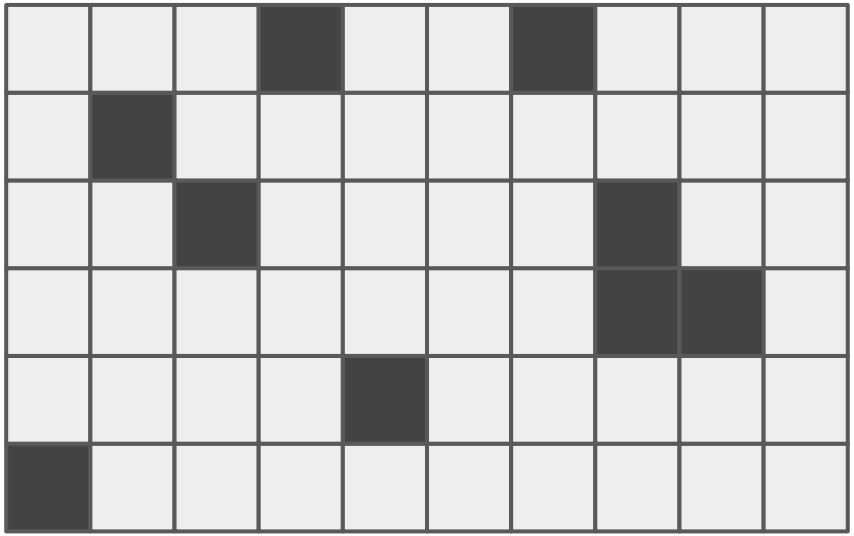


Enunț: Se dă o matrice cu valori de 1 și 0. Celulele care conțin valoarea 1 sunt blocate, restul sunt libere. Să se determine submatricea liberă de arie maximă.

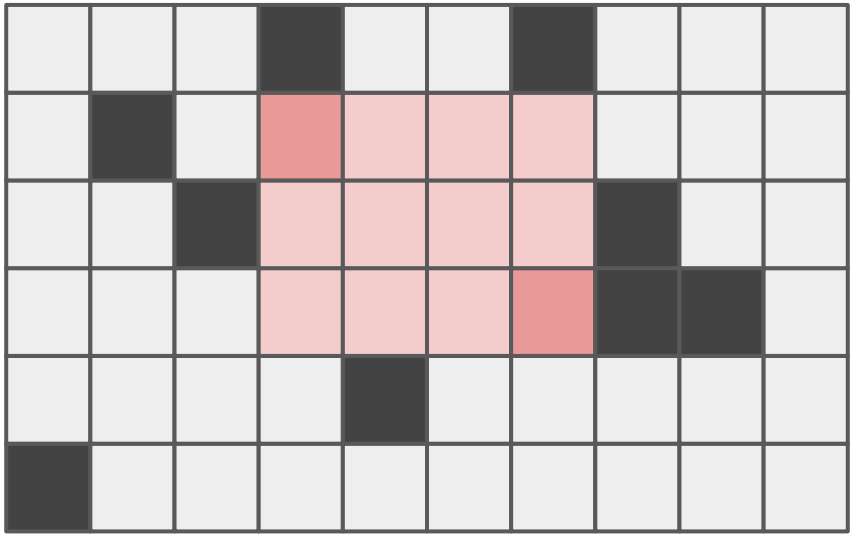


Arie maximă: 3x4=12

Soluții ineficiente



Soluția 1: Fixăm folosind două for-uri colțul stânga-sus al submatricei, iar cu alte două for-uri, colțul din dreapta-jos. Apoi, parcurgem submatricea și verificăm dacă este liberă. Dacă este liberă, actualizăm aria maximă cu aria submatricei.

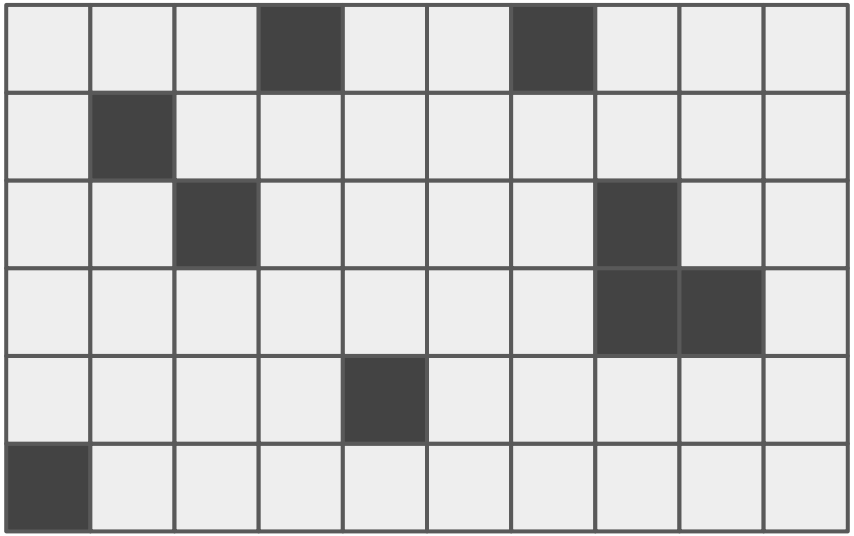


```cpp
#include <bits/stdc++.h>
using namespace std;
const int NM_MAX = 20;
int n, m;
bool mat[NM_MAX + 2][NM_MAX + 2];
int main () {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mat[i][j];
        }
    }
    int mx = 0;
    for (int i1 = 1; i1 <= n; i1++) {
        for (int j1 = 1; j1 <= m; j1++) {
            for (int i2 = i1; i2 <= n; i2++) {
                for (int j2 = j1; j2 <= m; j2++) {
                    bool ok = true;
                    for (int i = i1; i <= i2; i++) {
                        for (int j = j1; j <= j2; j++) {
                            if (mat[i][j] == 1) {
                                ok = false;
                            }
                        }
                    }
                    if (ok == true) {
                        mx = max(mx, (i2 - i1 + 1) * (j2 - j1 + 1));
                    }
                }
            }
        }
    }
    cout << mx << "\n";
    return 0;
}
```

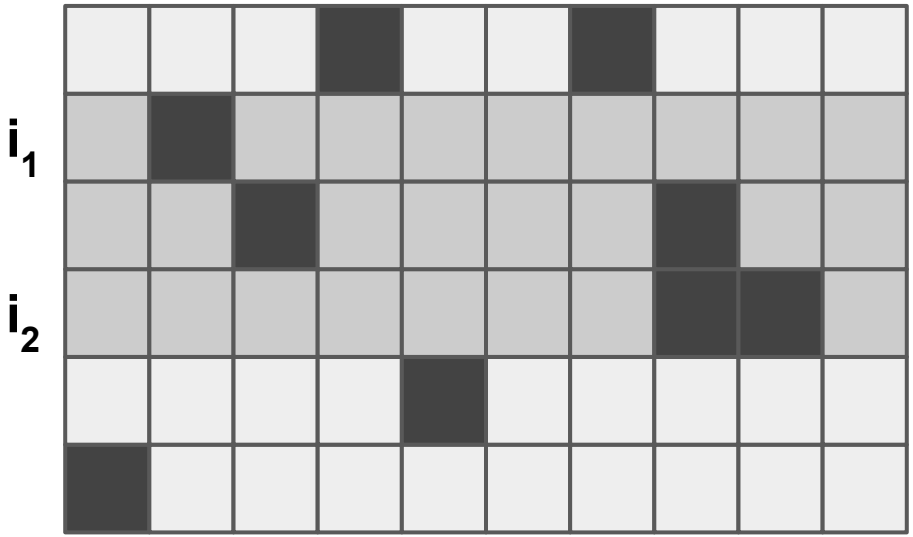
```cpp
#include <bits/stdc++.h>
using namespace std;
const int NM_MAX = 20;
int n, m;
bool mat[NM_MAX + 2][NM_MAX + 2];
int main () {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mat[i][j];
        }
    }
    int mx = 0;
    for (int i1 = 1; i1 <= n; i1++) {
        for (int j1 = 1; j1 <= m; j1++) {
            for (int i2 = i1; i2 <= n; i2++) {
                for (int j2 = j1; j2 <= m; j2++) {
                    bool ok = true;
                    for (int i = i1; i <= i2; i++) {
                        for (int j = j1; j <= j2; j++) {
                            if (mat[i][j] == 1) {
                                ok = false;
                            }
                        }
                    }
                    if (ok == true) {
                        mx = max(mx, (i2 - i1 + 1) * (j2 - j1 + 1));
                    }
                }
            }
        }
    }
    cout << mx << "\n";
    return 0;
}
```

Soluția 1 face $O(n^3m^3)$ operații pentru a ajunge la rezultat. Așadar, este eficientă pentru n, m ≤ 20.

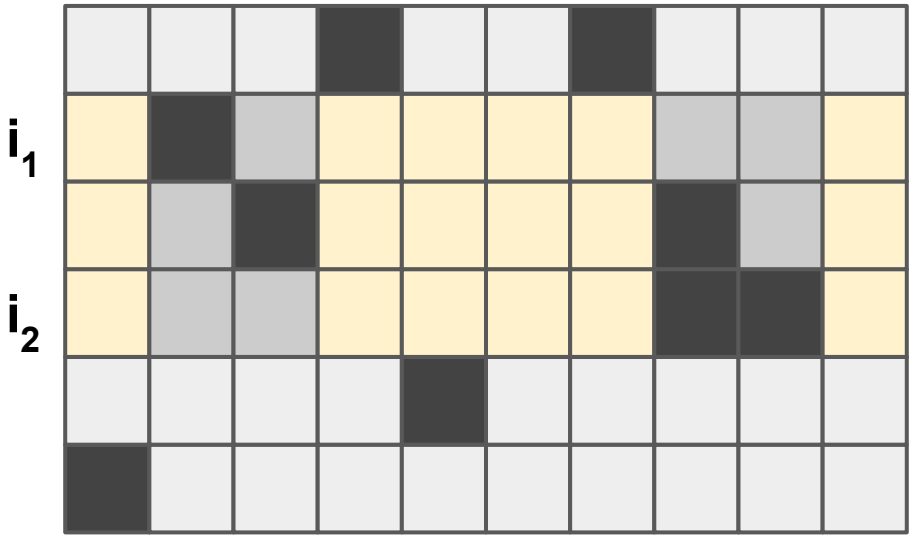
Soluții ineficiente



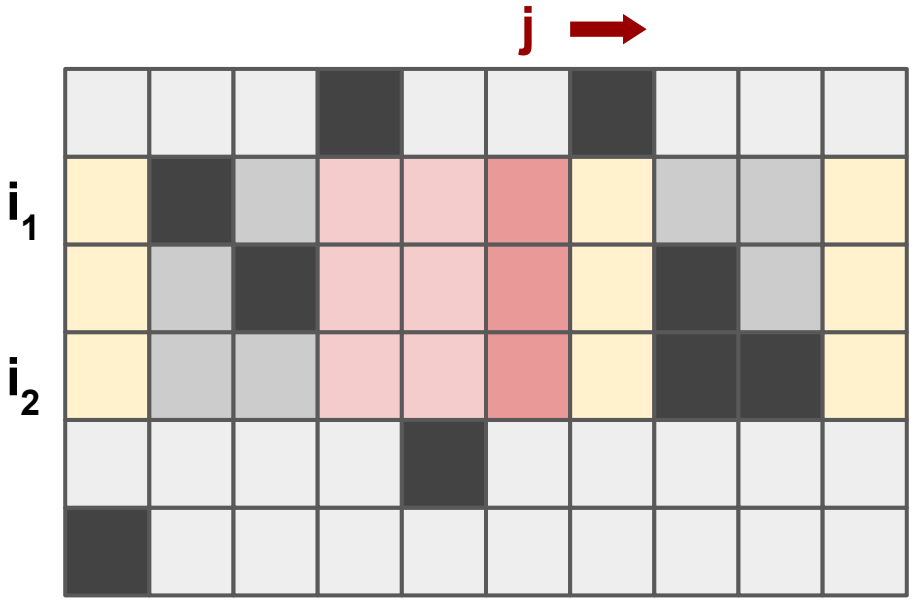
Soluția 2: Fixăm cele două rânduri i_1 și i_2 care ar putea încadra submatricea de arie maximă.



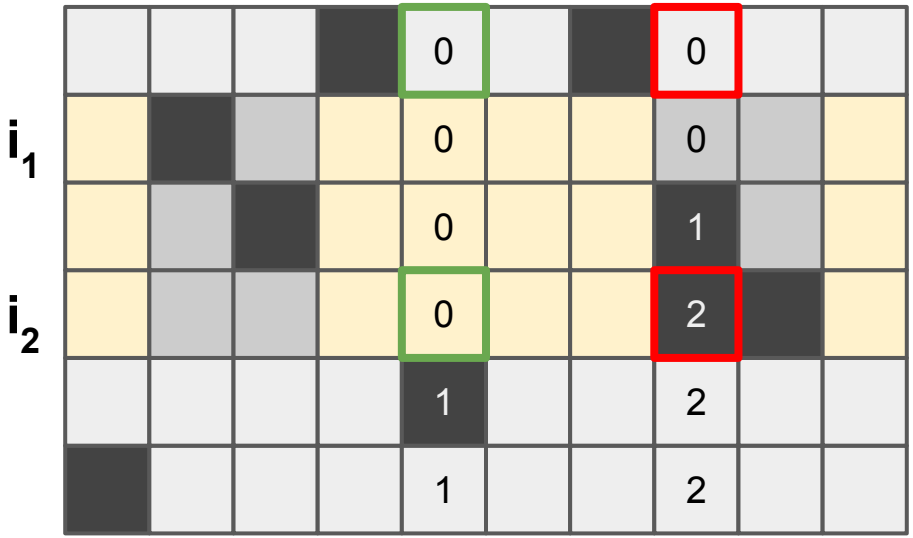
Soluția 2: Fixăm cele două rânduri i_1 și i_2 care ar putea încadra submatricea de arie maximă. Pentru fiecare coloană, verificăm dacă este complet liberă între cele două rânduri.



Soluția 2: Fixăm cele două rânduri i_1 și i_2 care ar putea încadra submatricea de arie maximă. Pentru fiecare coloană, verificăm dacă este complet liberă între cele două rânduri. Apoi, parcurgem coloanele în ordine de la prima la ultima și menținem secvența maximală de coloane libere. Când întâlnim o coloană blocată, resetăm contorul, altfel, îl creștem cu 1.



Pentru a verifica rapid dacă o coloană este liberă, putem precalcula sume parțiale pe coloane. Pentru o anumită coloană j, verificăm dacă sum[i_2][j] == sum[i_1-1][j].

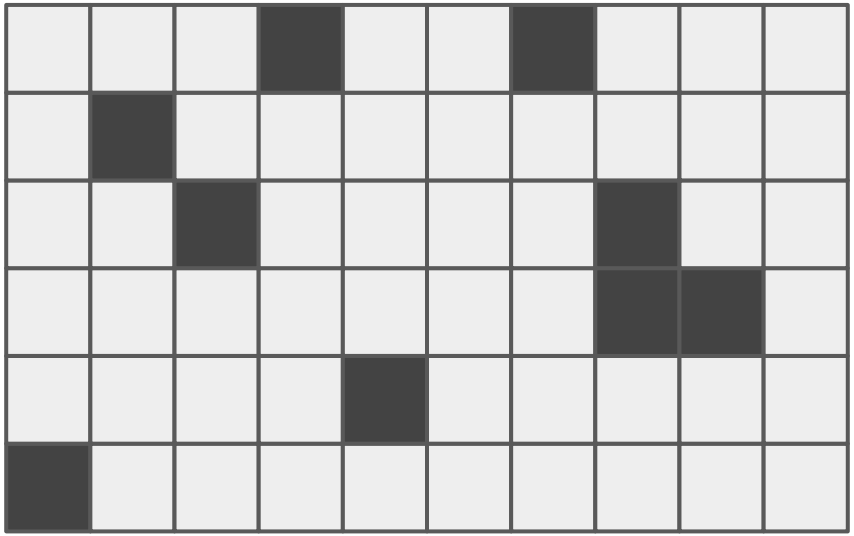


```cpp
#include <bits/stdc++.h>
using namespace std;
const int NM_MAX = 400;
int n, m;
bool mat[NM_MAX + 2][NM_MAX + 2];
int sum[NM_MAX + 2][NM_MAX + 2];
int main () {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mat[i][j];
            sum[i][j] = sum[i - 1][j] + mat[i][j];
        }
    }
    int mx = 0;
    for (int i1 = 1; i1 <= n; i1++) {
        for (int i2 = i1; i2 <= n; i2++) {
            int len = 0;
            for (int j = 1; j <= m; j++) {
                if (sum[i2][j] == sum[i1 - 1][j]) {
                    len++; mx = max(mx, len * (i2 - i1 + 1));
                } else {
                    len = 0;
                }
            }
        }
    }
    cout << mx << "\n";
    return 0;
}
```

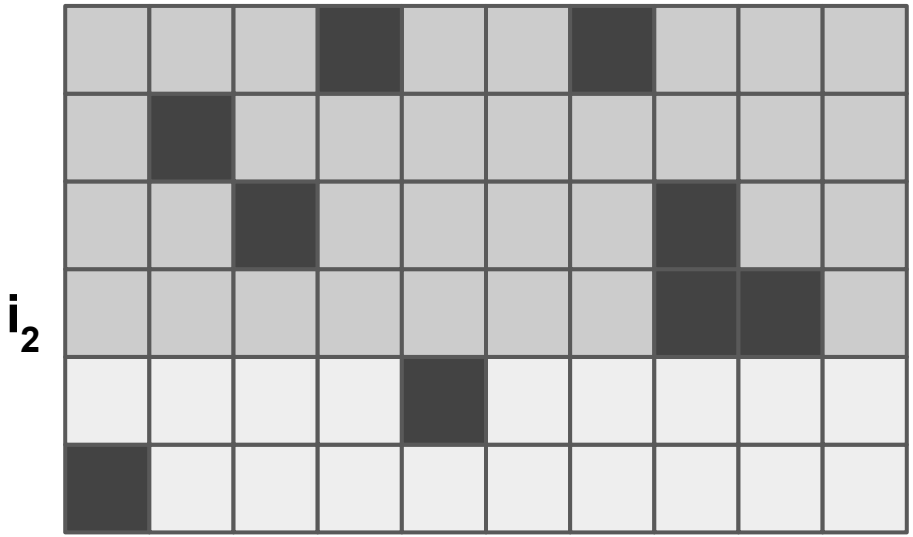
```cpp
#include <bits/stdc++.h>
using namespace std;
const int NM_MAX = 400;
int n, m;
bool mat[NM_MAX + 2][NM_MAX + 2];
int sum[NM_MAX + 2][NM_MAX + 2];
int main () {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mat[i][j];
            sum[i][j] = sum[i - 1][j] + mat[i][j];
        }
    }
    int mx = 0;
    for (int i1 = 1; i1 <= n; i1++) {
        for (int i2 = i1; i2 <= n; i2++) {
            int len = 0;
            for (int j = 1; j <= m; j++) {
                if (sum[i2][j] == sum[i1 - 1][j]) {
                    len++; mx = max(mx, len * (i2 - i1 + 1));
                } else {
                    len = 0;
                }
            }
        }
    }
    cout << mx << "\n";
    return 0;
}
```

Soluția 2 face $O(n^2m)$ operații pentru a ajunge la rezultat. Așadar, este eficientă pentru n, m ≤ 400.

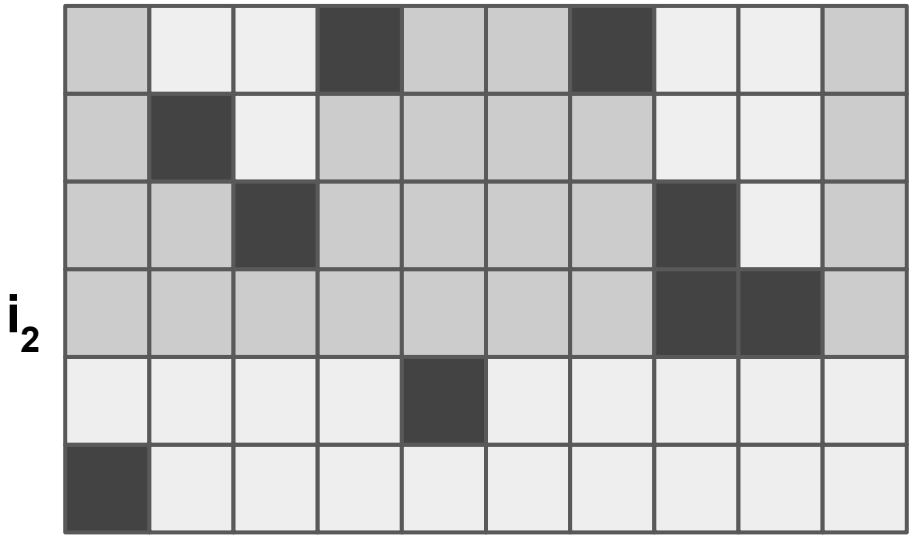
Soluția eficientă



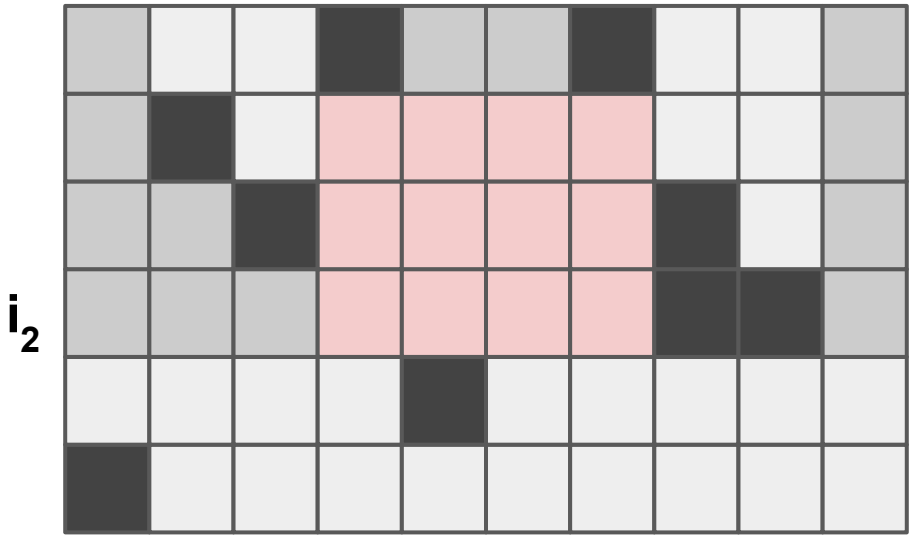
Soluția 3: Fixăm rândul i_2 de data aceasta.



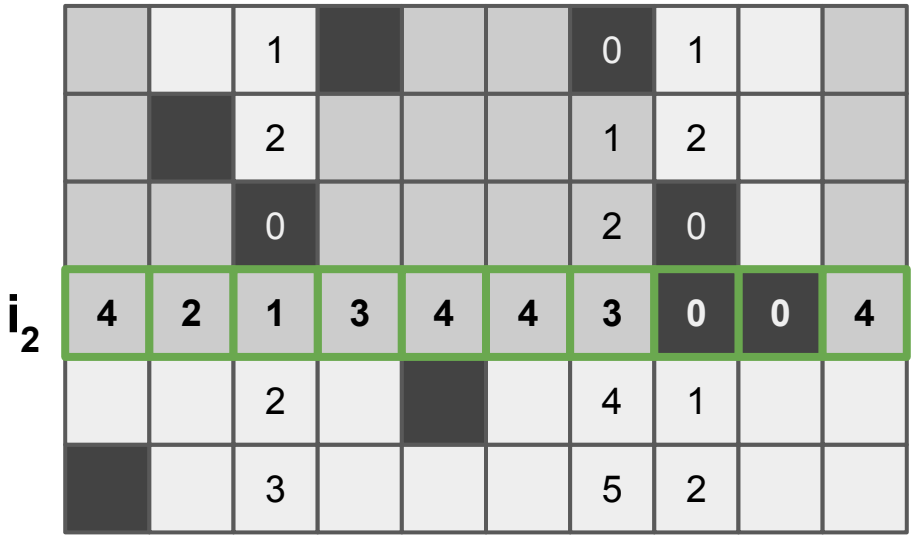
Soluția 3: Fixăm doar rândul i_2 de data aceasta. Pentru fiecare coloană, găsim cel mai jos blocaj, care este totuși mai sus de rândul i_2. Astfel, se formează o histogramă.



Soluția 3: Fixăm doar rândul i_2 de data aceasta. Pentru fiecare coloană, găsim cel mai jos blocaj, care este totuși mai sus de rândul i_2. Astfel, se formează o histogramă. Găsim dreptunghiul de arie maximă de sub histogramă.



Pentru a determina rapid histograma, precalculam pentru fiecare celulă, secvența maximă de celule libere de deasupra ei. Pentru a face asta, parcurgem fiecare coloană de sus în jos. Dacă întâlnim un obstacol, resetăm contorul, altfel, îl creștem cu 1.



```cpp
#include <bits/stdc++.h>
using namespace std;
const int NM_MAX = 400;
int n, m;
bool mat[NM_MAX + 2][NM_MAX + 2];
int up[NM_MAX + 2][NM_MAX + 2];
int h[NM_MAX + 2];
int st[NM_MAX + 2], cnt;
int L[NM_MAX + 2], R[NM_MAX + 2];
int main () {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mat[i][j];
            if (mat[i][j] == 1) {
                up[i][j] = 0;
            } else {
                up[i][j] = up[i - 1][j] + 1;
            }
        }
    }
int mx = 0;
    for (int i2 = 1; i2 <= n; i2++) {
        for (int j = 1; j <= m; j++) {
            h[j] = up[i2][j];
        }
        cnt = 0;
        for (int j = 1; j <= m; j++) {
            while (cnt > 0 && h[st[cnt]] >= h[j]) {
                cnt--;
            }
            L[j] = st[cnt] + 1;
            cnt++; st[cnt] = j;
        }
        cnt = 0; st[0] = m + 1;
        for (int j = m; j >= 1; j--) {
            while (cnt > 0 && h[st[cnt]] >= h[j]) {
                cnt--;
            }
            R[j] = st[cnt] - 1;
            cnt++; st[cnt] = j;
        }
        for (int j = 1; j <= m; j++) {
            mx = max(mx, h[j] * (R[j] - L[j] + 1));
        }
    }
    cout << mx << "\n";
    return 0;
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;
const int NM_MAX = 400;
int n, m;
bool mat[NM_MAX + 2][NM_MAX + 2];
int up[NM_MAX + 2][NM_MAX + 2];
int h[NM_MAX + 2];
int st[NM_MAX + 2], cnt;
int L[NM_MAX + 2], R[NM_MAX + 2];
int main () {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mat[i][j];
            if (mat[i][j] == 1) {
                up[i][j] = 0;
            } else {
                up[i][j] = up[i - 1][j] + 1;
            }
        }
    }
```

Soluția 3 face O(nm) operații pentru a ajunge la rezultat. Așadar, este eficientă pentru n, m ≤ 5000.

```cpp
int mx = 0;
    for (int i2 = 1; i2 <= n; i2++) {
        for (int j = 1; j <= m; j++) {
            h[j] = up[i2][j];
        }
        cnt = 0;
        for (int j = 1; j <= m; j++) {
            while (cnt > 0 && h[st[cnt]] >= h[j]) {
                cnt--;
            }
            L[j] = st[cnt] + 1;
            cnt++; st[cnt] = j;
        }
        cnt = 0; st[0] = m + 1;
        for (int j = m; j >= 1; j--) {
            while (cnt > 0 && h[st[cnt]] >= h[j]) {
                cnt--;
            }
            R[j] = st[cnt] - 1;
            cnt++; st[cnt] = j;
        }
        for (int j = 1; j <= m; j++) {
            mx = max(mx, h[j] * (R[j] - L[j] + 1));
        }
    }
    cout << mx << "\n";
    return 0;
}
```

Probleme propuse

https://www.pbinfo.ro/probleme/877/cuburi2

https://www.pbinfo.ro/probleme/1267/plaja

https://www.pbinfo.ro/probleme/3453/jungla

https://www.pbinfo.ro/probleme/2728/skyline

https://www.pbinfo.ro/probleme/2429/matrice9

https://www.pbinfo.ro/probleme/2665/dreptunghi1