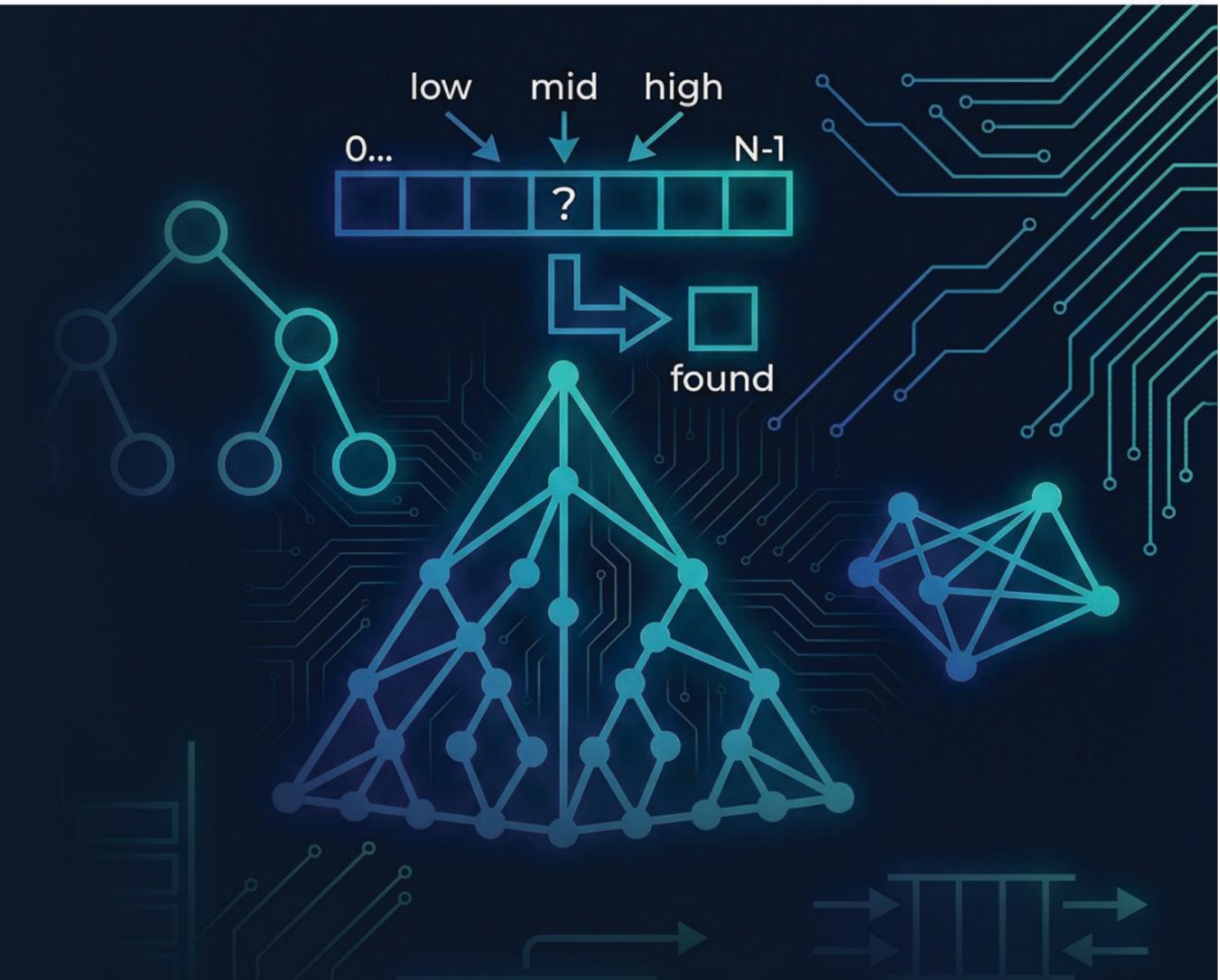
INDEKS W KIESZENI



PODRĘCZNIK TECHNIK ALGORYTMICZNYCH

WOJCIECH RYBAK

INDEKS W KIESZENI

WOJCIECH RYBAK

Podręcznik technik algorytmicznych

Wersja demonstracyjna

WARSZAWA 2026

Wydanie I

Autor: Wojciech Rybak

Projekt graficzny okładki: Grzegorz Przybyło

ISBN: 978-83-68290-11-0

Wydawnictwo Indeks w Kieszeni

IWK MAT sp. z o.o.

www.indekswkieszeni.pl

Słowo od autora

Ten podręcznik jest zbiorem klasycznych technik spotykanych w zadaniach algorytmicznych. Każda sekcja ma podobny schemat, na który składają się: intuicja, motywujący przykład, formalny opis, kod w C++ oraz zadania do ćwiczeń. W każdej sekcji jedno zadanie jest wyróżnione jako **zadanie wiodące** – do niego dołączony został kompletny kod. Dodatkowo przed kodem znajduje się krótkie omówienie treści zadania, aby było jasne, co dokładnie liczymy.

1 Wyszukiwanie binarne

Wprowadzenie

Wyszukiwanie binarne polega na **dzieleniu przestrzeni poszukiwań na pół** w każdej iteracji. Najważniejszym warunkiem jego zastosowania jest **monotoniczność** odpowiedzi na pytanie typu „tak/nie". Jeśli dla pewnego x warunek jest spełniony, to dla wszystkich większych x też (albo odwrotnie).

Motywujący przykład

Gra w „większe/mniejsze" to najprostsza intuicja: pytamy o środek przedziału i eliminujemy połowę możliwości. Ta sama idea działa przy tak zwanym *wyszukiwaniu po odpowiedzi*: szukamy minimalnej wartości, która „już działa".

Opis techniki

Wyszukiwanie binarne stosujemy wtedy, gdy potrafimy zadać pytanie typu *tak/nie* o pewną wartość x, a odpowiedź jest **monotoniczna**. Najczęściej oznacza to istnienie progu x_0 takiego, że:

$$\texttt{dziala}(x) = \begin{cases} \textit{fałsz}, & x < x_0, \\ \textit{prawda}, & x \geq x_0. \end{cases}$$

Wtedy problemem jest znalezienie x_0 (wariant **first true**, czyli odpowiednik `lower_bound`).

Kluczowa jest praca na **inwariancie**. Dla wariantu *first true* utrzymujemy zawsze:

$$\texttt{dziala}(l) = \textit{fałsz}, \qquad \texttt{dziala}(r) = \textit{prawda},$$

oraz przedział, w którym leży odpowiedź. W każdej iteracji bierzemy środek

$$s = l + \left\lfloor \frac{r-l}{2} \right\rfloor,$$

i sprawdzamy `dziala(s)`:

- jeśli `dziala(s)` jest prawdą, to odpowiedź jest w $[l, s]$, więc ustawiamy $r = s$,
- w przeciwnym razie odpowiedź jest w $[s, r]$, więc ustawiamy $l = s$.

Kończymy, gdy $r - l \leq 1$. Wtedy r jest najmniejszym argumentem z prawdą.

Istnieje też wariant **last true** (odpowiednik `upper_bound`-1): utrzymujemy $\texttt{dziala}(l) = \textit{prawda}$ i $\texttt{dziala}(r) = \textit{fałsz}$, a aktualizacje brzegów są „odwrócone".

Typowe zastosowania:

- *szukanie w posortowanej tablicy* (granica wartości),
- *wyszukiwanie po odpowiedzi* (minimalny czas, minimalna pojemność, minimalne X spełniające warunek),
- *problemy z monotonicznym predykatem* (np. „czy da się?" dla danego parametru).

Złożoność: liczba iteracji to $O(\log(r-l))$, a koszt całości to $O(\log(r-l) \cdot C)$, gdzie C to koszt policzenia `dziala(x)`.

Omówienie zadania wiodącego: *CSES – Factory Machines*

W zadaniu dostajemy n maszyn. Maszyna i produkuje jeden produkt co k_i jednostek czasu, czyli w czasie T wytworzy dokładnie $\left\lfloor \frac{T}{k_i} \right\rfloor$ produktów. Dane jest też t – liczba produktów, które musimy łącznie wyprodukować.

Naszym celem jest znalezienie **minimalnego** czasu T, takiego że:

$$\sum_{i=1}^{n} \left\lfloor \frac{T}{k_i} \right\rfloor \geq t.$$

To typowy przykład „wyszukiwania po odpowiedzi": dla małego T warunek jest fałszywy, a po przekroczeniu pewnej granicy staje się prawdziwy i pozostaje prawdziwy dla większych czasów.

Kod w C++ (zadanie wiodące: CSES – Factory Machines)

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

static vector<long long> k;
static long long goal;

bool can(long long time) {
    __int128 produced = 0; // ochrona przed przepełnieniem sumy
    for (long long x : k) {
        produced += time / x;
        if (produced >= goal) return true;
    }
    return produced >= goal;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n >> goal;
    k.resize(n);
    for (int i = 0; i < n; i++) cin >> k[i];

    // Inwariant: can(l) == false, can(r) == true
    long long l = -1;
    long long r = (long long)4e18;

    while (r - l > 1) {
        long long s = l + (r - l) / 2;
        if (can(s)) r = s;  // szukamy minimalnego czasu
        else l = s;
    }

    cout << r << "\n";
    return 0;
}
```

Zadania do ćwiczeń

1. ***CSES – Factory Machines*** – minimalny czas wytworzenia t produktów przez n maszyn (wyszukiwanie po odpowiedzi).
2. *Codeforces 706B – Interesting Drink* – dla zapytań policz, ile cen $\leq q$. (lower_bound).
3. *AtCoder ABC146C – Buy an Integer* – maksymalne n spełniające warunek budżetu (wyszukiwanie po odpowiedzi).

4. *SPOJ – AGGRCOW* – maksymalizacja minimalnej odległości (wyszukiwanie po odpowiedzi).
5. *LeetCode 278 – First Bad Version* – pierwszy wadliwy indeks (granica).

2 Two Pointers oraz Sliding Window

Wprowadzenie

Metoda dwóch wskaźników utrzymuje przedział $[l, r]$ i przesuwa oba wskaźniki *tylko w prawo*. Jeśli własność okna jest „naprawialna" przez przesuwanie l, to cały algorytm działa w $O(n)$.

Motywujący przykład

Mamy czasy czytania książek i limit czasu T. Chcemy otrzymać najdłuższy spójny fragment o sumie $\leq T$. Zwiększamy r, a gdy suma przekroczy limit – przesuwamy l, aż znów będzie dobrze.

Opis techniki

Metoda **two pointers** utrzymuje dwa indeksy l i r, opisujące aktualny przedział (okno) $[l, r]$. Najważniejsza własność tej techniki polega na tym, że oba wskaźniki przesuwamy **tylko w jedną stronę** (zwykle w prawo), dzięki czemu łączna liczba przesunięć jest $O(n)$.

Sliding window to szczególny przypadek two pointers, gdy utrzymujemy inwariant:

"okno $[l, r]$ spełnia warunek".

Typowy schemat wygląda tak:
- zwiększamy r i aktualizujemy strukturę opisującą okno (np. sumę, liczniki, maksimum),
- jeśli warunek jest złamany, przesuwamy l dopóki warunek nie wróci.
- W każdej chwili możemy aktualizować najlepszą odpowiedź na podstawie bieżącego okna.

Żeby ta logika była poprawna, warunek musi być **naprawialny przez przesuwanie** l: gdy okno jest „za duże" (np. suma $> T$, liczba różnych wartości $> K$), to usuwanie elementów z lewej strony musi przybliżać nas do spełnienia warunku.

Klasyczne warianty:
- suma w oknie $\leq T$ (zwykle wymaga nieujemnych liczb);
- co najwyżej K różnych wartości (liczniki częstotliwości);
- okno spełniające ograniczenia na liczbę „złych" elementów;
- maksymalny/minimalny element w oknie (czasem z deque, tzw. monotonic queue).

Złożoność: $O(n)$ przesunięć wskaźników, a więc zwykle $O(n)$ czasu + koszt aktualizacji stanu okna.

Omówienie zadania wiodącego: *Codeforces 279B – Books*

Dostajemy n książek ustawionych w kolejności oraz czas czytania każdej z nich: $a_0, a_1, \ldots, a_{n-1}$. Mamy również limit czasu T. Chcemy przeczytać **pewien spójny fragment** książek (czyli kolejne książki), tak aby suma czasów czytania nie przekroczyła T.

Należy wypisać **maksymalną liczbę** książek w takim fragmencie. To klasyczny przypadek sliding window: rozszerzamy okno w prawo, a gdy suma robi się za duża, zwężamy je z lewej strony.

Kod w C++ (zadanie wiodące: Codeforces 279B – Books)

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    long long T;
```

```
    cin >> n >> T;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];

    long long sum = 0;
    int l = 0, best = 0;

    for (int r = 0; r < n; r++) {
        sum += a[r];
        while (sum > T) {      // zwężamy okno, aż wróci warunek
            sum -= a[l];
            l++;
        }
        best = max(best, r - l + 1);
    }

    cout << best << "\n";
    return 0;
}
```

Zadania do ćwiczeń

1. ***Codeforces 279B – Books*** –najdłuższy spójny fragment o sumie $\leq T$ (sliding window).
2. *LeetCode 3* – najdłuższy podciąg bez powtórzeń (okno + częstości).
3. *LeetCode 904* – najdłuższy fragment z co najwyżej dwiema różnymi wartościami.
4. *CSES – Subarray Sums I* – liczba podtablic o sumie x (często prefix sum + mapa).

3 Sumy prefiksowe (1D i 2D)

Wprowadzenie

Suma prefiksowa to sposób na odpowiadanie na zapytania o sumy w $O(1)$ po wstępnym obliczeniu tablicy pomocniczej.

Motywujący przykład

Dysponujesz mapą lasu i dużą liczbą zapytań: ile drzew znajduje się w danym prostokącie? Bez prefiksów każde zapytanie kosztowałoby $O(n^2)$, a z prefiksami $O(1)$.

Opis techniki (1D i 2D)

Prefiksy 1D. Dla tablicy $a[0..n-1]$ budujemy:

$$pref[0] = 0, \qquad pref[i+1] = pref[i] + a[i].$$

Wtedy suma na przedziale $[l, r]$ (indeksy 0-based, włącznie) wynosi:

$$\sum_{i=l}^{r} a[i] = pref[r+1] - pref[l],$$

czyli odpowiedź na zapytanie jest w $O(1)$ po wcześniejszym przygotowaniu prefiksów w $O(n)$.

Prefiksy 2D. Dla planszy $n \times n$ (lub ogólnie $H \times W$) budujemy tablicę:

$$pref[i][j] = \text{wartość w prostokącie } [1..i] \times [1..j]$$

(typowo używa się indeksowania 1-based w prefiksach, żeby łatwo obsłużyć brzeg). Wzór to klasyczne włączenia/wyłączenia:

$$pref[i][j] = pref[i-1][j] + pref[i][j-1] - pref[i-1][j-1] + val(i, j).$$

Suma w prostokącie (x_1, y_1) do (x_2, y_2) to:

$$pref[x_2][y_2] - pref[x_1 - 1][y_2] - pref[x_2][y_1 - 1] + pref[x_1 - 1][y_1 - 1].$$

Po co to działa? Każdy punkt w środku jest dodany dokładnie raz: sumujemy dwa większe prostokąty, odejmujemy ich część wspólną, a na końcu dodajemy bieżące pole.

Złożoność:
- budowa 1D: $O(n)$, zapytanie: $O(1)$,
- budowa 2D: $O(HW)$, zapytanie: $O(1)$,
- pamięć 2D: $O(HW)$ (warto o tym pamiętać przy dużych limitach).

Omówienie zadania wiodącego: *CSES – Forest Queries*

Dostajemy planszę $n \times n$ złożoną ze znaków:
- `'.'` – puste pole,
- `'*'` – pole z drzewem.

Następnie dostajemy q zapytań. Każde zapytanie podaje współrzędne prostokąta (zwykle 1-based): (x_1, y_1) jako lewy górny róg oraz (x_2, y_2) jako prawy dolny róg.

Dla każdego zapytania należy wypisać, ile znaków `'*'` znajduje się wewnątrz tego prostokąta. Ponieważ zapytań jest dużo, chcemy mieć odpowiedź w $O(1)$ po jednorazowym przygotowaniu prefiksów 2D.

Kod w C++ (zadanie wiodące: CSES – Forest Queries)

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, q;
    cin >> n >> q;
    vector<string> g(n);
    for (int i = 0; i < n; i++) cin >> g[i];

    vector<vector<int>> pref(n + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            int add = (g[i - 1][j - 1] == '*');
            // pref 2D: włączenia/wyłączenia
            pref[i][j] = pref[i - 1][j] + pref[i][j - 1] - pref[i - 1][j - 1] + add;
        }
    }

    while (q--) {
        int x1, y1, x2, y2;
        cin >> x1 >> y1 >> x2 >> y2;
        int ans = pref[x2][y2] - pref[x1 - 1][y2] - pref[x2][y1 - 1] + pref[x1 - 1][y1 - 1];
        cout << ans << "\n";
    }
    return 0;
}
```

Zadania do ćwiczeń

1. ***CSES – Forest Queries*** – liczba gwiazdek w prostokącie na planszy (prefix sum 2D).
2. *CSES – Range Sum Queries I* – statyczne sumy przedziałów (prefix sum 1D).
3. *LeetCode 304* – suma w prostokącie (2D prefix).

Ta książka to praktyczny przegląd **klasycznych technik** niezbędnych do rozwiązywania **zadań algorytmicznych**. Publikacja przeprowadzi Cię przez **19 kluczowych zagadnień** – od podstawowych metod przeszukiwania, przez algorytmy grafowe i programowanie dynamiczne, aż po zaawansowane struktury danych i algorytmy tekstowe.

Każdy rozdział został opracowany według **czytelnego schematu:**

- **Intuicja i przykład:** służą one zrozumieniu idei stojącej za algorytmem;
- **Opis formalny:** polega na precyzyjnym zdefiniowaniu techniki;
- **Kod w C++:** gotowa implementacja omawianego zagadnienia;
- **Zadania:** ćwiczenia do samodzielnego rozwiązania, pochodzące z popularnych platform takich jak CSES, Codeforces czy LeetCode.

W każdym dziale znajdziesz również **szczegółowe omówienie zadania wiodącego wraz z kompletnym kodem źródłowym**, co pozwala dokładnie prześledzić działanie algorytmu w praktyce. To konkretna pomoc dla osób przygotowujących się do różnego rodzaju zawodów programistycznych.

Wydawnictwo INDEKS W KIESZENI

Warszawa 2026
ISBN: 978-83-68290-11-0