ŚCIĄGA - MATEMATYKA DYSKRETNA

Algorytmy i Struktury Danych - Pracownia Specjalistyczna

SPIS TREŚCI

PROBLEM 1 - SYMBOL NEWTONA

TEORIA MATEMATYCZNA

Symbol Newtona (współczynnik dwumianowy): $$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Interpretacja kombinatoryczna:

- Liczba sposobów wyboru k elementów z n-elementowego zbioru (bez zwracania uwagi na kolejność)
- Przykład: $\binom{8}{3} = 56$ - na 56 sposobów można wybrać 3 elementy z 8

Własności:

- $\binom{n}{0} = \binom{n}{n} = 1$
- $\binom{n}{k} = \binom{n}{n-k}$ (symetria)
- **Wzór rekurencyjny (trójkąt Pascala):** $$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Trójkąt Pascala:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

ALGORYTM I - Z DEFINICJI

Idea: Bezpośrednie obliczenie według wzoru $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$

Pseudokod:

```
AlgorytmI(n, k):
    nSilnia = 1
```

```
    for i = 2 to n:
        nSilnia *= i

    kSilnia = 1
    for i = 2 to k:
        kSilnia *= i

    nkSilnia = 1
    for i = 2 to (n-k):
        nkSilnia *= i

    return nSilnia / (kSilnia * nkSilnia)
```

Operacja elementarna: Mnożenie/dzielenie

Liczba operacji:

- Obliczenie n!: (n-1) mnożeń
- Obliczenie k!: (k-1) mnożeń
- Obliczenie (n-k)!: (n-k-1) mnożeń
- Końcowe mnożenie i dzielenie: 2 operacje
- **Razem:** n - 1 + k - 1 + n - k - 1 + 2 = **n + k operacji**

Złożoność czasowa: $O(n)$

Zalety:

- Prosty do zrozumienia
- Bezpośrednio odpowiada definicji matematycznej

Wady:

- Duże wartości n! mogą powodować przepełnienie
- Nieefektywny dla wielokrotnych obliczeń różnych symboli

Implementacja C#:

```
static (long wynik, int operacje) AlgorytmI_SymbolNewtona(int n, int k)
{
    int operacje = 0;

    long nSilnia = 1;
    for (int i = 2; i <= n; i++)
    {
        nSilnia *= i;
        operacje++; // mnożenie
    }

    long kSilnia = 1;
    for (int i = 2; i <= k; i++)
    {
```

```
            kSilnia *= i;
            operacje++;
        }

        long nkSilnia = 1;
        for (int i = 2; i <= (n - k); i++)
        {
            nkSilnia *= i;
            operacje++;
        }

        operacje++; // mnożenie k! * (n-k)!
        operacje++; // dzielenie

        return (nSilnia / (kSilnia * nkSilnia), operacje);
    }
```

ALGORYTM V - TRÓJKĄT PASCALA (TABLICA 2D)

Idea: Wykorzystanie wzoru rekurencyjnego: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Własności matematyczne:

- Każdy element to suma dwóch elementów powyżej
- Brzegi trójkąta zawsze równe 1: $\binom{i}{0} = 1$
- Symetria: możemy budować tylko połowę trójkąta

Pseudokod:

```
AlgorytmV(n, k):
    // Optymalizacja: wykorzystaj symetrię
    if k > n - k:
        k = n - k

    pascal[0..n][0..k]

    // Inicjalizacja pierwszej kolumny
    for i = 0 to n:
        pascal[i][0] = 1

    // Budowanie trójkąta
    for i = 1 to n:
        for j = 1 to min(i, k):
            pascal[i][j] = pascal[i-1][j-1] + pascal[i-1][j]

    return pascal[n][k]
```

Operacja elementarna: Dodawanie

Liczba operacji:

- Budujemy tylko do k-tej kolumny (optymalizacja)
- Dla każdego wiersza i od 1 do n:
  - Wykonujemy min(i, k) dodawań
- **Razem:** $\sum_{i=1}^{n} \min(i, k)$
  - Jeśli k ≥ n: $\frac{n(n+1)}{2} - n = O(n^2)$
  - Jeśli k < n: $O(n \cdot k)$

Złożoność czasowa: $O(n \cdot k)$ lub $O(n^2)$ w najgorszym przypadku

Zalety:

- Tylko operacje dodawania (mniejsze ryzyko przepełnienia)
- Efektywny dla wielokrotnych obliczeń (można wykorzystać tablicę wielokrotnie)
- Numerycznie stabilny

Wady:

- Wymaga dodatkowej pamięci: $O(n \cdot k)$
- Wolniejszy dla pojedynczych obliczeń małych wartości

Implementacja C#:

```
static (long wynik, int operacje) AlgorytmV_SymbolNewtona(int n, int k)
{
    int operacje = 0;

    // Optymalizacja: symetria
    if (k > n - k)
        k = n - k;

    long[,] pascal = new long[n + 1, k + 1];

    // Inicjalizacja C(i,0) = 1
    for (int i = 0; i <= n; i++)
    {
        pascal[i, 0] = 1;
    }

    // Budowanie trójkąta
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= Math.Min(i, k); j++)
        {
            pascal[i, j] = pascal[i - 1, j - 1] + pascal[i - 1, j];
            operacje++; // dodawanie
        }
    }

    return (pascal[n, k], operacje);
}
```

PORÓWNANIE ALGORYTMÓW (PROBLEM 1)

Kryterium	Algorytm I (Definicja)	Algorytm V (Pascal)
Operacja elementarna	Mnożenie/dzielenie	Dodawanie
Liczba operacji	O(n + k)	O(n·k) lub O(n^2)
Złożoność czasowa	O(n)	O(n·k)
Złożoność pamięciowa	O(1)	O(n·k)
Ryzyko przepełnienia	Wysokie (silnie!)	Niskie (dodawanie)
Wielokrotne użycie	Nieefektywne	Efektywne

Przykład (n=8, k=3):

- Algorytm I: 8 + 3 = 11 operacji
- Algorytm V: ~14 operacji (budowanie części trójkąta)

PROBLEM 2 - ZBIORY 2-ELEMENTOWE

TEORIA MATEMATYCZNA

Problem: Mamy n liczb naturalnych z przedziału [1, k]. Chcemy znaleźć minimalną liczbę zbiorów 2-elementowych {x, y} takich, że x + y ≤ k.

Własności:

- Jeśli dla liczby x nie można znaleźć pary y (tj. x + y > k dla wszystkich dostępnych y), to x tworzy zbiór 1-elementowy {x}
- Optymalne rozwiązanie łączy najmniejsze z największymi liczbami

Strategia zachłanna (greedy):

1. Posortuj liczby rosnąco
2. Łącz najmniejszą liczbę z największą
3. Jeśli suma ≤ k, utwórz parę; w przeciwnym razie największa idzie sama

Dowód poprawności strategii zachłannej:

- Najmniejsza liczba x musi być sparowana z jak największą możliwą liczbą
- Jeśli x nie może być sparowana z największą dostępną, to nie może być sparowana z żadną inną
- Największa liczba, która nie może być sparowana z najmniejszą, nie może być sparowana z żadną inną

ALGORYTM I - ZACHŁANNY (OPTYMALNY)

Idea: Sortowanie + technika dwóch wskaźników (two pointers)

Pseudokod:

```
AlgorytmI_Zachłanny(n, k, liczby):
    posortuj liczby rosnąco
    zbiory = []
    left = 0
    right = n - 1

    while left <= right:
        if left == right:
            zbiory.dodaj({liczby[left]})
            break

        if liczby[left] + liczby[right] <= k:
            zbiory.dodaj({liczby[left], liczby[right]})
            left++
            right--
        else:
            zbiory.dodaj({liczby[right]})
            right--

    return zbiory
```

Operacja elementarna: Porównanie

Liczba operacji:

- Sortowanie: O(n log n) porównań (~n $\log_2$ n)
- Parowanie: O(n) porównań (każdy element przetwarzany raz)
- **Razem:** ~n $\log_2$ n + n

Złożoność czasowa: $O(n \log n)$ - zdominowane przez sortowanie

Złożoność pamięciowa: O(n) - kopia tablicy do sortowania

Zalety:

- Gwarantuje optymalną liczbę zbiorów
- Efektywny czasowo
- Prosty do implementacji

Implementacja C#:

```csharp
static (List<List<int>> zbiory, int operacje) AlgorytmI_Zbiory(int n, int k, int[] liczby)
{
    int operacje = 0;
    List<List<int>> zbiory = new List<List<int>>();

    int[] sorted = new int[n];
```

```
    Array.Copy(liczby, sorted, n);
    Array.Sort(sorted);
    operacje += (int)(n * Math.Log(n, 2)); // sortowanie

    int left = 0;
    int right = n - 1;

    while (left <= right)
    {
        operacje++; // porównanie left <= right

        if (left == right)
        {
            zbiory.Add(new List<int> { sorted[left] });
            break;
        }

        operacje++; // porównanie sumy
        if (sorted[left] + sorted[right] <= k)
        {
            zbiory.Add(new List<int> { sorted[left], sorted[right] });
            left++;
            right--;
        }
        else
        {
            zbiory.Add(new List<int> { sorted[right] });
            right--;
        }
    }

    return (zbiory, operacje);
}
```

Przykład działania:

```
Dane: n=8, k=140
Liczby: [60, 70, 80, 56, 67, 78, 81, 68]

Po sortowaniu: [56, 60, 67, 68, 70, 78, 80, 81]

Krok 1: left=56, right=81  → 56+81=137 ≤ 140 → {56, 81}
Krok 2: left=60, right=80  → 60+80=140 ≤ 140 → {60, 80}
Krok 3: left=67, right=78  → 67+78=145 > 140  → {78}
Krok 4: left=67, right=70  → 67+70=137 ≤ 140 → {67, 70}
Krok 5: left=68, right=68  → {68}

Wynik: 5 zbiorów
```

ALGORYTM II - NAIWNY

Idea: Dla każdej liczby szukaj pierwszej możliwej pary liniowo

Pseudokod:

```
AlgorytmII_Naiwny(n, k, liczby):
    zbiory = []
    uzyte[1..n] = false

    for i = 1 to n:
        if uzyte[i]:
            continue

        znalezionoPare = false
        for j = i+1 to n:
            if uzyte[j]:
                continue

            if liczby[i] + liczby[j] <= k:
                zbiory.dodaj({liczby[i], liczby[j]})
                uzyte[i] = true
                uzyte[j] = true
                znalezionoPare = true
                break

        if not znalezionoPare:
            zbiory.dodaj({liczby[i]})
            uzyte[i] = true

    return zbiory
```

Operacja elementarna: Porównanie

Liczba operacji: $O(n^2)$ - w najgorszym przypadku dla każdej liczby sprawdzamy wszystkie pozostałe

Złożoność czasowa: $O(n^2)$

Wady:

- Nieefektywny
- Nie gwarantuje optymalnego rozwiązania
- Wynik zależy od kolejności liczb w wejściu

Implementacja C#:

```csharp
static (List<List<int>> zbiory, int operacje) AlgorytmII_Zbiory(int n, int k,
int[] liczby)
{
    int operacje = 0;
    List<List<int>> zbiory = new List<List<int>>();
```

```
        bool[] uzyte = new bool[n];

        for (int i = 0; i < n; i++)
        {
            operacje++;
            if (uzyte[i]) continue;

            bool znalezionoPare = false;

            for (int j = i + 1; j < n; j++)
            {
                operacje++;
                if (uzyte[j]) continue;

                operacje++;
                if (liczby[i] + liczby[j] <= k)
                {
                    zbiory.Add(new List<int> { liczby[i], liczby[j] });
                    uzyte[i] = true;
                    uzyte[j] = true;
                    znalezionoPare = true;
                    break;
                }
            }

            if (!znalezionoPare && !uzyte[i])
            {
                zbiory.Add(new List<int> { liczby[i] });
                uzyte[i] = true;
            }
        }

        return (zbiory, operacje);
    }
```

PORÓWNANIE ALGORYTMÓW (PROBLEM 2)

Kryterium	Algorytm I (Zachłanny)	Algorytm II (Naiwny)
Złożoność czasowa	O(n log n)	O(n^2)
Optymalizacja	TAK - minimalna liczba zbiorów	NIE
Zależność od kolejności	NIE (sortuje)	TAK
Liczba operacji (n=8)	~27	~64 (w najgorszym)
Gwarancja poprawności	TAK	NIE

PROBLEM 4 - SEJF KRÓLA BAJTDOCJI

TEORIA MATEMATYCZNA

Problem: Korytarz szerokości n metrów z m prętami laserowymi. Każdy pręt zajmuje przedział $[y_1, y_2]$ wysokości. Znaleźć wszystkie bezpieczne pasma (gdzie nie ma prętów).

Model matematyczny:

- Korytarz: przedział [0, n]
- Pręt i: przedział $[y_{1i}, y_{2i}]$
- Bezpieczne pasmo: przedział [a, b] taki, że nie przecina się z żadnym prętem

Własności:

- Pręty mogą się nakładać lub dotykać
- Pręt może być zamocowany wzdłuż korytarza $(0 \le x_1 \le x_2 \le n)$
- Oś Y jest krytyczna (0Y - dół, S - góra)

Kluczowa obserwacja: Jeśli scalimy wszystkie nakładające się przedziały zajęte, to bezpieczne pasma to luki między nimi.

ALGORYTM I - SCALANIE PRZEDZIAŁÓW (OPTYMALNY)

Idea:

1. Zbierz wszystkie zajęte przedziały $[y_1, y_2]$
2. Posortuj według punktu początkowego
3. Scal nakładające się/stykające się przedziały
4. Znajdź luki między scalonymi przedziałami

Matematyka scalania:

- Dwa przedziały $[a_1, b_1]$ i $[a_2, b_2]$ $(a_1 \le a_2)$ nakładają się, gdy $a_2 \le b_1$
- Scalony przedział: $[a_1, \max(b_1, b_2)]$

Pseudokod:

```
AlgorytmI_Scalanie(n, prety):
    zajete = []
    for pret in prety:
        zajete.dodaj([pret.y1, pret.y2])

    posortuj zajete według punktu początkowego

    // Scalanie
    scalone = []
    current = zajete[0]
    for i = 1 to |zajete|-1:
        if zajete[i].y1 <= current.y2:
            current.y2 = max(current.y2, zajete[i].y2)
        else:
            scalone.dodaj(current)
```

```
            current = zajete[i]
    scalone.dodaj(current)

    // Szukanie luk
    bezpieczne = []
    pozycja = 0
    for zajety in scalone:
        if pozycja < zajety.y1:
            bezpieczne.dodaj([pozycja, zajety.y1])
        pozycja = max(pozycja, zajety.y2)

    if pozycja < n:
        bezpieczne.dodaj([pozycja, n])

    return bezpieczne
```

Operacja elementarna: Porównanie

Liczba operacji:

- Sortowanie: O(m log m) ≈ m $\log_2$ m porównań
- Scalanie: O(m) porównań
- Szukanie luk: O(m) porównań
- **Razem:** ~m $\log_2$ m + 2m

Złożoność czasowa: $O(m \log m)$ - zdominowane przez sortowanie

Złożoność pamięciowa: O(m) - listy przedziałów

Implementacja C#:

```
static (List<(int y1, int y2)> pasma, int operacje) AlgorytmI_Sejf(
    int n, (int x1, int y1, int x2, int y2)[] prety)
{
    int operacje = 0;
    List<(int y1, int y2)> bezpieczne = new List<(int, int)>();

    // Zbierz zajęte przedziały
    List<(int y1, int y2)> zajete = new List<(int, int)>();
    for (int i = 0; i < prety.Length; i++)
    {
        zajete.Add((prety[i].y1, prety[i].y2));
    }

    if (zajete.Count == 0)
    {
        if (n > 0) bezpieczne.Add((0, n));
        return (bezpieczne, operacje);
    }

    // Sortowanie
    zajete.Sort((a, b) => a.y1.CompareTo(b.y1));
```

```
        operacje += (int)(prety.Length * Math.Log(prety.Length, 2));

        // Scalanie
        List<(int y1, int y2)> scalone = new List<(int, int)>();
        var current = zajete[0];

        for (int i = 1; i < zajete.Count; i++)
        {
            operacje++;
            if (zajete[i].y1 <= current.y2)
            {
                operacje++;
                current = (current.y1, Math.Max(current.y2, zajete[i].y2));
            }
            else
            {
                scalone.Add(current);
                current = zajete[i];
            }
        }
        scalone.Add(current);

        // Szukanie luk
        int pozycja = 0;
        foreach (var z in scalone)
        {
            operacje++;
            if (pozycja < z.y1)
            {
                bezpieczne.Add((pozycja, z.y1));
            }
            operacje++;
            pozycja = Math.Max(pozycja, z.y2);
        }

        operacje++;
        if (pozycja < n)
        {
            bezpieczne.Add((pozycja, n));
        }

        return (bezpieczne, operacje);
    }
```

Przykład działania:

```
Dane: n=11, m=5
Pręty (y1, y2): [2,5], [2,2], [6,6], [4,1], [4,4], [10,10], [10,10], [1,6], [5,9]

Krok 1: Zbierz zajęte: [2,5], [2,2], [6,6], [1,4], [4,4], [10,10], [10,10], [1,6],
[5,9]
```

```
Krok 2: Sortuj: [1,4], [1,6], [2,2], [2,5], [4,4], [5,9], [6,6], [10,10], [10,10]

Krok 3: Scalaj:
  [1,4] + [1,6] → [1,6]
  [1,6] + [2,2] → [1,6] (zawiera się)
  [1,6] + [2,5] → [1,6] (zawiera się)
  [1,6] + [4,4] → [1,6] (zawiera się)
  [1,6] + [5,9] → [1,9]
  [1,9] + [6,6] → [1,9] (zawiera się)
  [1,9] + [10,10] → [1,9], [10,10]
  [10,10] + [10,10] → [10,10]

Scalone: [1,9], [10,10]

Krok 4: Luki:
  0 < 1 → [0,1]
  9 < 10 → [9,10]
  10 < 11 → [10,11]

UWAGA: [10,11] jest błędne bo [10,10] zajęte!
Poprawnie: [0,1], [9,10], pasmo końcowe jeśli 10 < 11
```

ALGORYTM II - BRUTE FORCE

Idea: Dla każdego punktu y ∈ [0, n] sprawdź czy jest zajęty przez jakiś pręt

Pseudokod:

```
AlgorytmII_BruteForce(n, prety):
    bezpieczny[0..n] = true

    for pret in prety:
        for y = pret.y1 to pret.y2:
            bezpieczny[y] = false

    // Zbierz przedziały
    pasma = []
    poczatek = -1
    for y = 0 to n:
        if bezpieczny[y]:
            if poczatek == -1:
                poczatek = y
        else:
            if poczatek != -1:
                pasma.dodaj([poczatek, y])
                poczatek = -1

    if poczatek != -1:
        pasma.dodaj([poczatek, n])

    return pasma
```

Operacja elementarna: Porównanie/przypisanie

Liczba operacji:

- Dla każdego pręta: (y_2 - y_1 + 1) operacji
- Razem dla wszystkich prętów: O(m · średnia_długość)
- Przejście po tablicy: O(n)
- **Razem:** O(m · długość_pręta + n) ≈ O(n · m) w najgorszym

Złożoność czasowa: $O(n \cdot m)$ lub $O(n + m \cdot L)$ gdzie L to średnia długość pręta

Wady:

- Nieefektywny dla dużych n
- Wymaga tablicy rozmiaru n (problem pamięciowy dla n=50000)

Implementacja C#:

```
static (List<(int y1, int y2)> pasma, int operacje) AlgorytmII_Sejf(
    int n, (int x1, int y1, int x2, int y2)[] prety)
{
    int operacje = 0;
    List<(int y1, int y2)> bezpieczne = new List<(int, int)>();

    bool[] bezp = new bool[n + 1];
    for (int i = 0; i <= n; i++)
    {
        bezp[i] = true;
    }

    // Oznacz zajęte punkty
    foreach (var pret in prety)
    {
        for (int y = pret.y1; y <= pret.y2; y++)
        {
            operacje++;
            if (y <= n)
                bezp[y] = false;
        }
    }

    // Zbierz przedziały
    int poczatek = -1;
    for (int y = 0; y <= n; y++)
    {
        operacje++;
        if (bezp[y])
        {
            if (poczatek == -1)
                poczatek = y;
        }
```

```
        else
        {
            if (poczatek != -1)
            {
                bezpieczne.Add((poczatek, y));
                poczatek = -1;
            }
        }
    }

    if (poczatek != -1)
    {
        bezpieczne.Add((poczatek, n));
    }

    return (bezpieczne, operacje);
}
```

PORÓWNANIE ALGORYTMÓW (PROBLEM 4)

Kryterium	Algorytm I (Scalanie)	Algorytm II (Brute Force)
Złożoność czasowa	O(m log m)	O(n·m) lub O(n + m·L)
Złożoność pamięciowa	O(m)	O(n)
Operacje	~$m \log_2 m$ + 2m	~suma_długości_prętów + n
Skalowalność	Doskonała	Słaba dla dużych n
Dla n=10000, m=50	~390 op.	~500050 op. (worst)

Optymalizacja: Algorytm I jest optymalny - O(m log m + n) przy wypisywaniu wyników.

ZŁOŻONOŚCI CZASOWE - PODSUMOWANIE

Notacja O (Big O)

Definicja: f(n) = O(g(n)) gdy istnieje c > 0 i n_0 takie, że dla wszystkich n ≥ n_0: $$f(n) \leq c \cdot g(n)$$

Hierarchia złożoności: $$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

Porównanie wszystkich algorytmów

Problem	Algorytm	Złożoność	Operacja	Typ
1	Algorytm I	O(n)	Mnożenie	Iteracyjny
1	Algorytm V	O(n·k)	Dodawanie	Programowanie dynamiczne

Problem	Algorytm	Złożoność	Operacja	Typ
2	Algorytm I	O(n log n)	Porównanie	Zachłanny
2	Algorytm II	$O(n^2)$	Porównanie	Naiwny
4	Algorytm I	O(m log m)	Porównanie	Zachłanny + sortowanie
4	Algorytm II	O(n·m)	Porównanie	Brute force

Przykładowe czasy dla n=1000

Złożoność	Operacje	Czas (1GHz)
O(n)	1,000	1 µs
O(n log n)	~10,000	10 µs
$O(n^2)$	1,000,000	1 ms
$O(n^3)$	1,000,000,000	1 s
$O(2^n)$	10^{300}	niemożliwe

TECHNIKI ALGORYTMICZNE

1. Programowanie dynamiczne (DP)

- **Idea:** Rozwiązuj mniejsze podproblemy i wykorzystuj ich wyniki
- **Przykład:** Trójkąt Pascala (Algorytm V)
- **Wzór rekurencyjny:** $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
- **Kiedy stosować:** Optymalne podstruktury + nakładające się podproblemy

2. Algorytmy zachłanne (Greedy)

- **Idea:** Wybieraj lokalnie optymalne rozwiązanie w każdym kroku
- **Przykłady:**
  - Problem 2: parowanie najmniejszej z największą
  - Problem 4: scalanie przedziałów
- **Kiedy stosować:** Własność zachłannego wyboru + optymalne podstruktury
- **Dowód poprawności:** Pokazać że zachłanny wybór prowadzi do optymalnego rozwiązania

3. Dziel i zwyciężaj (Divide and Conquer)

- **Idea:** Podziel problem na mniejsze, rozwiąż je, połącz wyniki
- **Przykład:** Sortowanie przez scalanie (Merge Sort) - O(n log n)
- **Wzór rekurencyjny:** T(n) = 2T(n/2) + O(n)

4. Two Pointers (Dwa wskaźniki)

- **Idea:** Dwa wskaźniki poruszają się po posortowanej strukturze
- **Przykład:** Problem 2 - left i right w posortowanej tablicy

- **Złożoność:** O(n) po sortowaniu
- **Kiedy stosować:** Pary elementów, przedziały, posortowane dane

---

WSKAZÓWKI DO ODPOWIEDZI USTNEJ

Jak odpowiadać na pytania o algorytmy:

1. **Przedstaw problem matematycznie**
   - Zdefiniuj notację (n, k, m)
   - Podaj wzory matematyczne
   - Wyjaśnij własności
2. **Opisz ideę algorytmu**
   - Jaka technika (greedy, DP, brute force)?
   - Dlaczego tak działa?
   - Kluczowe kroki
3. **Analiza złożoności**
   - Operacja elementarna
   - Liczba operacji (dokładna lub asymptotyczna)
   - Notacja O
   - Złożoność pamięciowa
4. **Przykład działania**
   - Małe dane wejściowe
   - Krok po kroku
   - Wynik
5. **Porównanie z innymi algorytmami**
   - Zalety i wady
   - Kiedy użyć którego

Typowe pytania:

Q: Dlaczego Algorytm V jest lepszy mimo większej złożoności? A: Używa tylko dodawania (vs mnożenie w Alg. I), co zmniejsza ryzyko przepełnienia. Dla wielokrotnych obliczeń symboli Newtona, tablica może być wykorzystana ponownie.

Q: Jak działa strategia zachłanna w Problemie 2? A: Łączymy najmniejszą liczbę z największą. Jeśli się mieszczą w limicie k, tworzymy parę. W przeciwnym razie największa nie może mieć pary (bo jeśli nie pasuje do najmniejszej, nie pasuje do żadnej). Gwarantuje minimalną liczbę zbiorów.

Q: Dlaczego sortujemy w Problemie 4? A: Aby efektywnie scalić nakładające się przedziały. Po sortowaniu wystarczy jedno przejście O(m), aby połączyć wszystkie nakładające się pręty.

Q: Co to jest operacja elementarna? A: To podstawowa operacja, której liczbę chcemy minimalizować. Zależy od algorytmu: mnożenie/dzielenie (Alg. I Problem 1), dodawanie (Alg. V), porównanie (Problem 2, 4).

Q: Różnica między O(n) a Θ(n)? A:

- O(n) - górne ograniczenie (≤)
- Θ(n) - dokładne ograniczenie (=)
- Ω(n) - dolne ograniczenie (≥)

WZORY I DEFINICJE DO ZAPAMIĘTANIA

Symbol Newtona

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Własności

- $\binom{n}{0} = \binom{n}{n} = 1$
- $\binom{n}{k} = \binom{n}{n-k}$
- $\binom{n}{1} = n$

Silnia

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$ $$0! = 1$$

Złożoności standardowych operacji

- Sortowanie (merge sort, heap sort): O(n log n)
- Sortowanie (quick sort średnio): O(n log n)
- Sortowanie (bubble sort): $O(n^2)$
- Wyszukiwanie binarne: O(log n)
- Wyszukiwanie liniowe: O(n)

Wzory sumy

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$ $$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$$

STRUKTURA KODU - WYJAŚNIENIA

Format plików wejściowych/wyjściowych

Problem 1 (In0101.txt):

```
8 3
```

Pierwsza linia: n=8, k=3 (gdzie k ≤ n)

Problem 1 (Out0101.txt):

```
n=8 k=3
SN1 = 56, liczba operacji = 14
SN5 = 56, liczba operacji = 4
```

Problem 2 (In0102.txt):

```
8 140
60
70
80
56
67
78
81
68
```

Pierwsza linia: n (liczba elementów), k (limit sumy) Następne n linii: poszczególne liczby

Problem 2 (Out0102.txt):

```
56 81
60 80
78
67 70
68
5
```

Każda linia: zbiór (1 lub 2 elementy) Ostatnia linia: liczba zbiorów

Problem 4 (In0104.txt):

```
11 5
2 5 2 6
4 1 4 4
4 10 10 10
1 6 5 9
3 8 7 9
```

Pierwsza linia: n (szerokość), m (liczba prętów) Następne m linii: x_1 y_1 x_2 y_2 (współrzędne pręta)

Problem 4 (Out0104.txt):

```
0 1
4 5
```

```
9 10
10 11
liczba bezpiecznych pasm: 4
```

Każda linia: y_1 y_2 (bezpieczne pasmo) Ostatnia linia: liczba pasm

---

POWODZENIA NA EGZAMINIE!

Pamiętaj:

- Matematyka + implementacja = pełne zrozumienie
- Złożoność czasowa to klucz do oceny algorytmu
- Przykłady pomagają w wyjaśnieniu
- Zachłanne algorytmy wymagają dowodu poprawności
- Programowanie dynamiczne = rekurencja + zapamiętywanie

Ściąga opracowana na podstawie implementacji w C# oraz teorii matematyki dyskretnej Algorytmy i Struktury Danych - Pracownia Specjalistyczna