78)

Algorytm:

Na potrzeby algorytmu założymy, że wszystkie liczby są różne. (W przypadku takich samych postępujemy indeksy).

Wiemy teraz zauważyć, że znalezienie mediany jest równoważne ze znalezieniem elementu, który jest większy od $\lfloor \frac{n}{2} \rfloor$ elementów.

- Dla każdej z k list:
  - p = 0 , k = n - 1
  - if k == p:
    break
  - kandydat = $\frac{k+p}{2}$
  - W pozostałych listach bin-search znajdujemy dla każdej listy pierwszą taką liczbę, która jest $>$ od kandydata. W ten sposób uzyskujemy po zsumowaniu liczbę wszystkich liczb $<$ kandydat.
  - if suma = $\lfloor \frac{n}{2} \rfloor$:
    return kandydat
    elif suma $< \lfloor \frac{n}{2} \rfloor$:
    p = kandydat + 1
    elif suma $> \lfloor \frac{n}{2} \rfloor$:
    k = kandydat - 1 (wróć do • powyżej (*))

Z3|

Szukamy: $\binom{n}{k} \bmod p$, czyli

$$\binom{n}{k} = n! \cdot (k!)^{-1} \cdot ((n-k)!)^{-1} \bmod p$$

Możemy najpierw stablicować sobie wartości $n! \% p$ od 1 do $\max(n_i)$. $O(n)$

->

```
sil[0] = 1
for(int i = 1; i <= max(n_i); i++)
    sil[i] = sil[i-1] * i % p;
```

Teraz moglibyśmy wyliczać odwrotności modulo dla $(k!)$ i $(n-k)!$ w czasie $O(\log n)$ korzystając z małego tw. Fermata:

$$a^{m-1} \equiv 1 \bmod m$$

$$a^{m-2} \equiv a^{-1} \bmod m$$

Możemy jednak wyliczyć sobie te odwrotności wcześniej innym sposobem w czasie $O(n)$ i otrzymać wtedy wszystkie wyniki w czasie $O(1)$. (n = $\max(n_i)$)

Lemat:

$$i^{-1} \equiv -\left\lfloor \frac{m}{i} \right\rfloor \cdot (m \bmod i)^{-1} \bmod m$$

D-d.

$$m \bmod i = m - \left\lfloor \frac{m}{i} \right\rfloor \cdot i \qquad | \bmod m$$

$$m \bmod i \equiv -\left\lfloor \frac{m}{i} \right\rfloor \cdot i \pmod m \qquad | \cdot i^{-1} \cdot (m \bmod i)^{-1}$$

$$(m \bmod i) \cdot i^{-1} \cdot (m \cdot \bmod i)^{-1} \equiv -\left\lfloor \frac{m}{i} \right\rfloor \cdot i \cdot i^{-1} \cdot (m \bmod i)^{-1}$$

$$\Downarrow$$

$$i^{-1} \equiv -\left\lfloor \frac{m}{i} \right\rfloor \cdot (m \bmod i)^{-1} \pmod m \quad \text{o}$$

Teraz mogiemy analogicznie policzyć odwrotności:

```
odw[1] = 1;
for(int i = 2; i <= max(n,i); i++)
    //odw[i] = p - (p/i) * (p mod i) * odw[p%i] %p;
```

Mając policzone odwrotności możemy wykorzystać tę konwencję: $(x!)^{-1} \equiv ((x-1)!)^{-1} \cdot x^{-1}$ oby wyliczyć odwrotności silni:

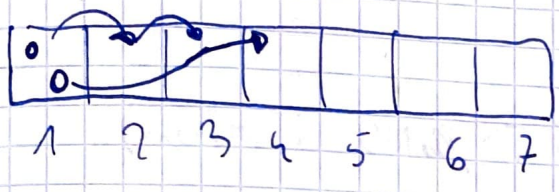
```
odw_sil[1] = 1;
for (int i = 2; i <= max(n); i++)
    odw_sil[i] = odw_sil[i-1] * odw[i] % p;
```

I finalnie mamy:

```
def npok(n, k):
    return sil[n] * odw_sil[k] % m * odw_sil[n-k] % m;
```

75|

Zauważmy, że nasz graf to tak na prawdę tablica jednowymiarowa, w której możemy skakać z dowolnego pola na dowolne inne leżące po prawej stronie.



rys. 1

Zauważmy, że jeśli wykonamy jakimś planem skok i omijamy w ten sposób nieodwiedzone pole, to drugi plan musi przejść po każdym z nich po kolei. (patrz rys. 1)

Możemy teraz zauważyć, że interesują nas tak na prawdę tylko dwie sytuacje:

- gdy dwa plany leżą na tym samym polu (1)
- gdy dwa plany leżą na dwóch sąsiednich polach (2)

Niech: $TS[i]$ - oznacza koszt dotarcia do i-tego pola przez oba plany jak w (1)

$NS[i]$ - oznacza koszt dotarcia do i-tego pola przez dalej wysunięty plan, jak w (2)

w obu - dotarcie wszystkie pola przed i są odwiedzone.

- Jak wyliczyć $TS[i]$ i $NS[i]$?

skok poprzedniego / dalszy dojazd skoczka

$$NS[i] = \min\Big(\min_{2 \le j < i} \big(NS[j] + d_{j-1,i} + \sum_{k=j}^{i-2} d_{k,k+1} \big), \min\big(TS[j] + d_{j,i} + \sum_{k=j}^{i-2} d_{k,k+1} \big) \Big)$$

$$TS[i] = NS[i] + d_{i-1,i}$$

4

Zauważmy, że wyliczanie kosztu grodzenia po kolei od i do j możemy sprepocesować korzystając z sum prefiksowych

Dzięki temu otrzymamy złożoność:

- czasowa - $O(n^2)$
- pamięciowa - $O(n)$ (TS, BONUS : prefsum)

27|

Chcemy podzielić zbiór na trzy podzbiory, których sumy są równe.

Możemy podejść do problemu dynamicznie (podobnie do wydawania reszty).

Niech $dp[i][j] = \{0,1\}$ gdzie my możemy uzyskać
↑ suma pierwszego zbioru ↑ suma drugiego

podział na sumy i, j i $S-i-j$, gdzie S to całość. Wtedy początkowo $dp[0][0] = 1$ i przechodzimy po kolei przez wartości/elementy zbioru.

Jeśli $dp[i][j] = true$ to $dp[i+a_i][j] = true$ i
$dp[i][j+a_i] = true$

Algorytm:

```
dp[0][0] = 1
for ai in zbiór:
    if ai < 0:
(*)     for i in range(-S, S):
            for j in range(-S, S):
                if dp[i][j] == true:
                    dp[i+ai][j] = true
                    dp[i][j+ai] = true
    else:
        (*) ale range(S, -S)
return dp[S/3][S/3]
```

złożoność czasowa: $O(n^3)$
pamięciowa: $O(n^2)$