ASD, w. 02

Podstawowe techniki algorytmiczne.

- rekurencja
- programowanie dynamiczne
- dziel i rządź.

Rekurencja

- zaprogramuj rozwiązanie dla przypadków bazowych (najprostszych)
- przypadki bardziej skomplikowane sprowadź do prostszych.

Przykład

Dwumian Newtona: $\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$, $n \geq m \geq 0$

$0! = 1$

Fakt

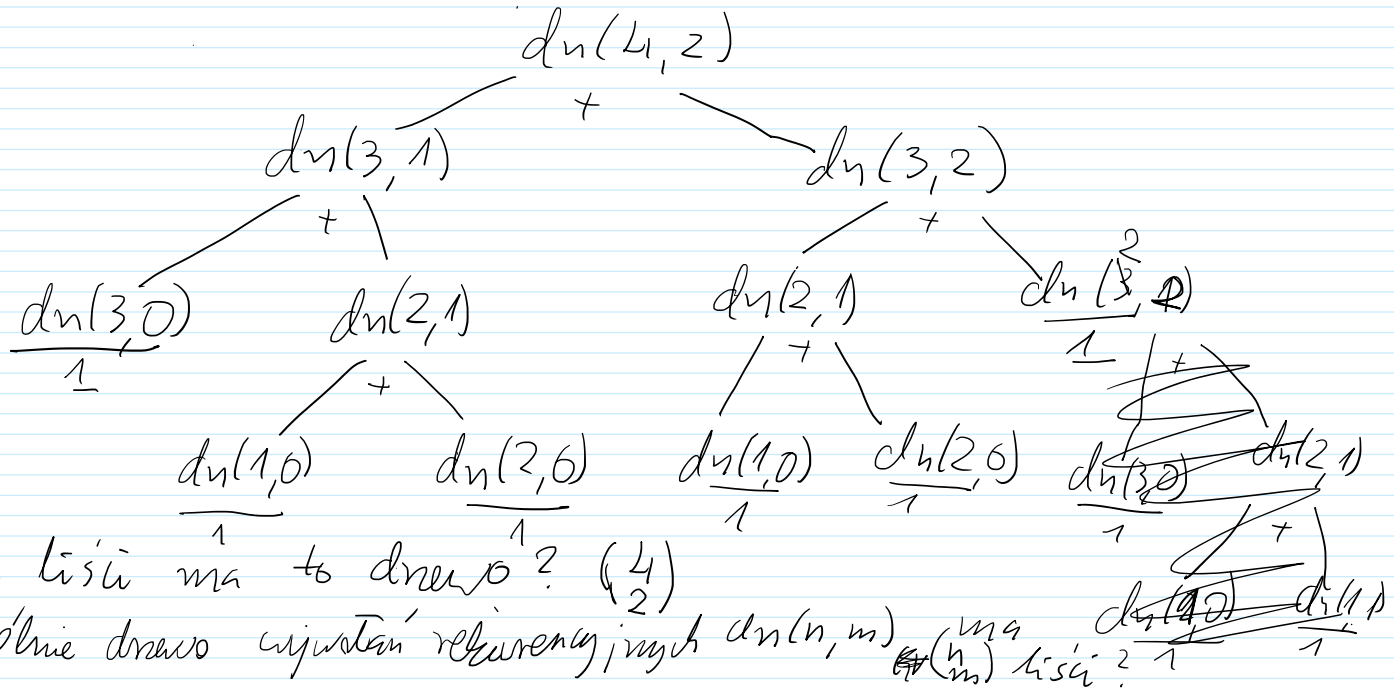
$0 < m < n$, $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$

$\binom{n}{0} = 1$, $\binom{n}{n} = 1$ - przypadki brzegowe.

---

dm(n, m) - funkcja licząca dwumian Newtona

```
dm(n, m){ // 0 <= m <= n - warunki na wejście
    if(m==0 || m==n) return 1;
    a = dm(n-1, m-1);
    b = dm(n-1, m);
    return a+b;
}
```



Ile liści ma to drzewo? $\binom{4}{2}$

Ogólnie drzewo wyjściowych rekurencyjnych $dn(n,m)$ ma $\binom{n}{m}$ liści?

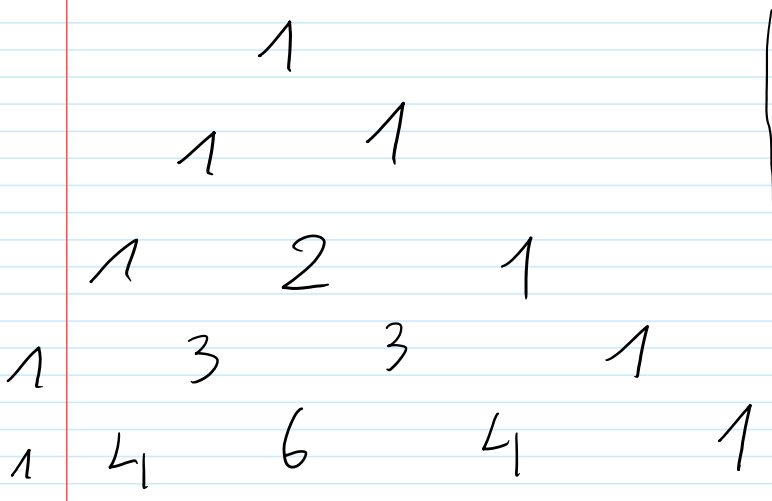
To dużo bo $\binom{n}{m}$ różnie wykładniczo wzgl. n, m.
Czyli podobnie wykładniczo względem $|n|+|m|$
(długości zapisu binarnego n i m).

Programowanie dynamiczne
Warunki wstępne

1) Wasz problem musi dać się podzielić na niewiele (najlepiej wielomianowo wiele) podproblemów.
2) Rozwiązania bardziej złożonych podproblemów można szybko (wielomianowo) złożyć z rozwiązań tych prostszych podproblemów.

>>> Będziemy mogli skonstruować ich rozwiązania

Twierdzenie Newtona i trójkąt Pascala



n	
1	$\binom{n}{0} = \binom{n}{n} = 1$
2	
3	$\binom{n}{k+1} = \binom{n}{k} + \binom{n-1}{k-1}$
4	
5	

Przykład

Obliczymy efektywnie $\binom{n}{m}$ korzystając z techniki prog. dyn.

Trójkat Pascala

$\binom{4}{2}$

		$i=1$	$i=2$				
	0	1	0				
	0	1	1	0			
	0	1	2	1	0		($\binom{4}{2}$)
	0	1	3	3	1	0	
($\binom{4}{i}$)	0	1	4	6	4	1	0
	0	1	5	10	10	5	1	0

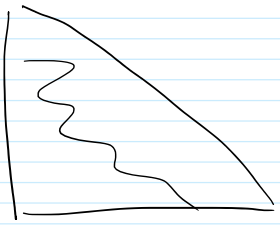
```
dn_dyn(n, m) { // n ≥ m
  int t[0,...,n] = {0,...,0}
  int pom[0,...,n] = {0,...,0}
  t[0] = 1;  (// t[0] = (0 0) = 1)
  pom[0] = 1;
  int i = 1;  // będziemy liczyć (i 0), (i 1), ..., (i i), 0, ...
  while(i ≤ n) {
    for(j = 1; j ≤ i; j++) {
      pom[j] = t[j-1] + t[j];
    } // po tej pętli w pom = {(i 0), (i 1), ..., (i i)}
    i = i+1;
    t = pom;   // przepisz pom na t.
  }
  return t[m];
}
```

	0	1	2	3	4	5
i=1	1	1	0	0	0	0
i=2	1	2	1	0	0	0
i=3	1	3	3	1	0	0
i=4	1	4	6	4	1	0

t[i+1]

Zauważmy, że

- podzieliliśmy nasz problem na $O(n^2)$ podproblemów
  to jest istotnie mniej
  niż liści drzewa
  wywołania dla - rekurencyjnego



- sprytnie liczymy $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$.
- tablicujemy potrzebne wartości.
  żeby szybko dało się do nich odwoływać.

Uwaga. $O(n^2)$ podproblemów to jest wykładniczo wiele względem $\ln$
Liczby na wyjściu są duże. (wykładnicze wzgl. n, m.)

Metoda dziel i rządź.

- metoda rekurencyjna
- dzielimy problem na a podproblemów, $a \geq 1$.
  b-razy mniejszych, $b > 1$
- łączymy rozwiązania podproblemów w rozwiązanie naszego problemu w jakimś efektywnym czasie.

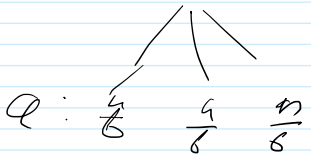
Złożoność tak stworzonego algorytmu to $T(n) = a \cdot T(\lfloor \frac{n}{b} \rfloor) + f(n)$, gdzie $f(n)$ to koszt połączenia

Twierdzenie (o rekursji uniwersalnej)

Niech $1 \leq a$, $1 < b$, c – stałe.

Jeśli $T(n)$ spełnia zależność, dla $n > 1$

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^c),$$



to

$$T(n) = \begin{cases} O(n^c), & \text{gdy } c > \log_b a \\ O(n^c \cdot \log_2(n)), & c = \log_b a \\ O(n^{\log_b a}), & \log_b(a) > c \end{cases}$$

Podobna zależność jest dla Θ w miejsce O

Rekurs ja

$$f(n) = \sum_{i=0}^{n} i$$

$$= \left(\sum_{i=0}^{n-1} i\right) + n$$

$$= f(n-1) + n.$$

```
int foo(int n){
  if (n <= 0) return 0;
  x = foo(n-1);
  return x + n;
}
```

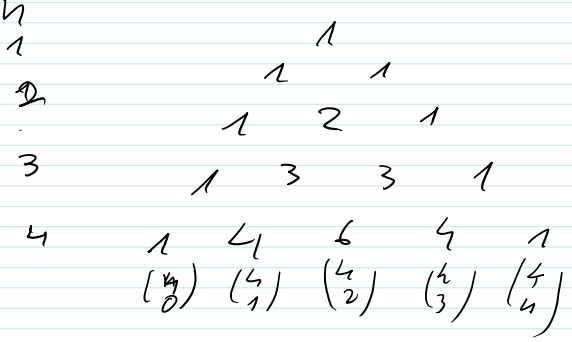
foo(3) ↑ 3+3=6
↓ foo(2) ↑ 1+2=3
↓ foo(1) ↑ 1+0=1
↓ foo(0) ↑ 0
foo(0) = 0

Dwumian Newtona, trójkąt Pascala

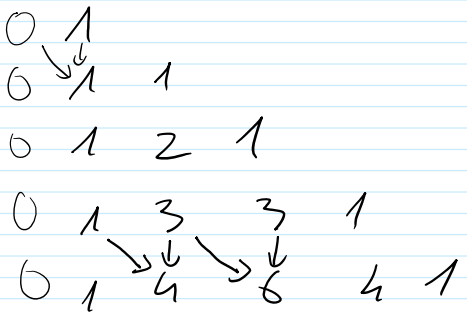
$\binom{n}{0} = \binom{n}{n} = 1$

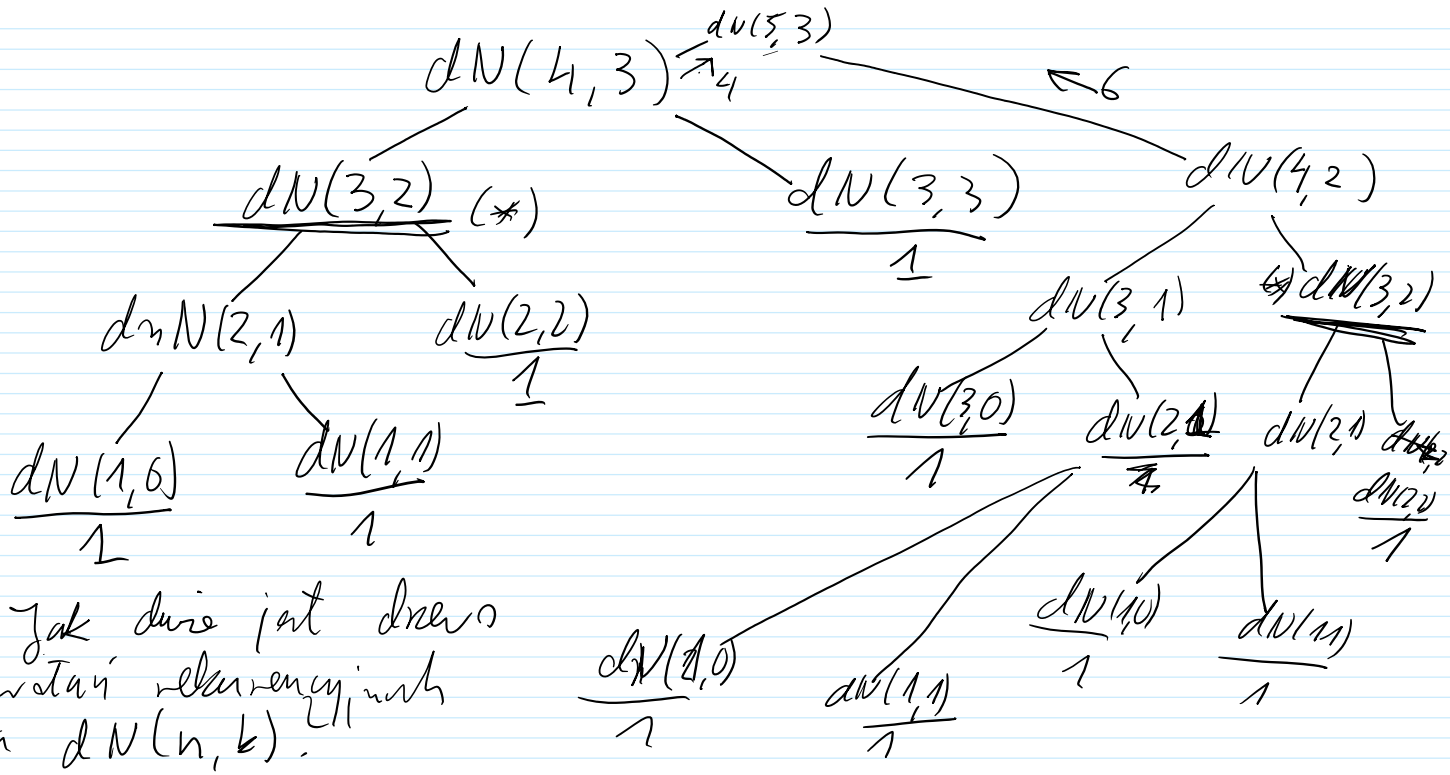
$\binom{n}{k+1} = \binom{n}{k} + \binom{n-1}{k}$, $1 \leq k+1 \leq n$

$\binom{n}{k} = |\mathcal{P}^{=k}(\{0,\ldots,n-1\})|$



```
int dwumianN(int n, int k){
    if(n<0 || k<0 || n<k)
        return 0;
    if(n==k || k==0)
        return 1;
    x = dwumianN(n-1, k-1);
    y = dwumianN(n-1, k);
    return x+y;
}
```





Jak duże jest drzewo wywołań rekurencyjnych dla $dN(n,k)$?

Takie drzewo ma $\binom{n}{k}$-liści.

Tzn. czas działania tego algorytmu jest wykładniczy wzgl. ~~n+k~~ $\max(n, k)$.

Problem

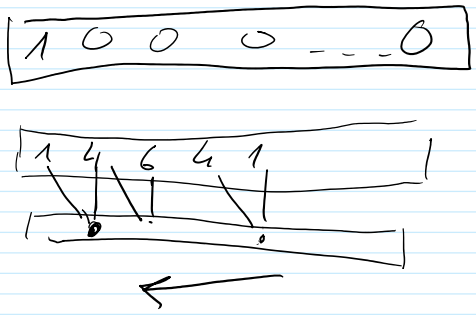
Wywołania rekurencyjne liczą wiele razy tą samą, nie przenoszą między sobą informacji.

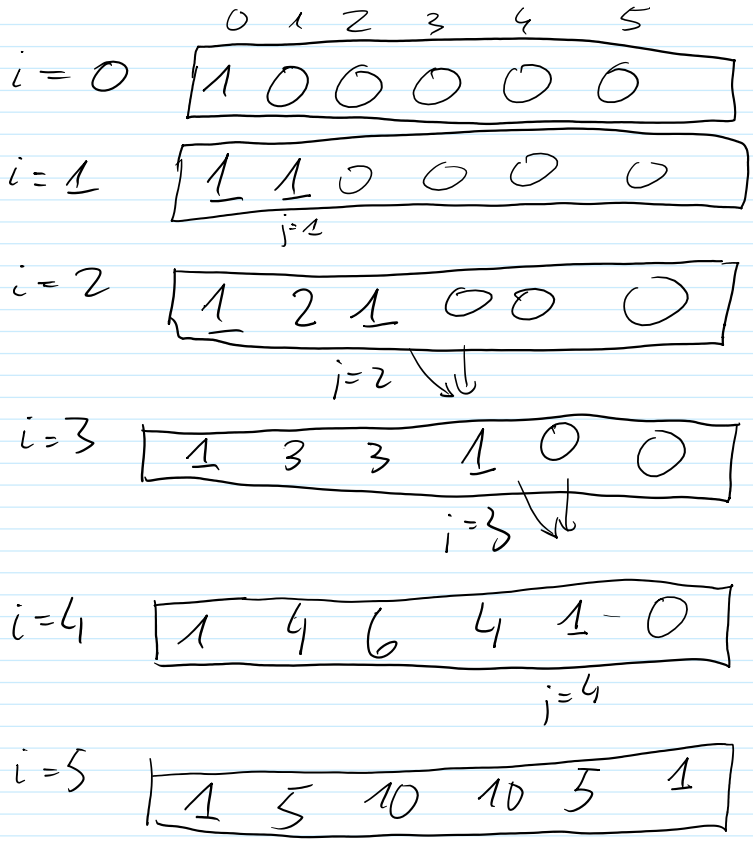
Programowanie dynamiczne

Dla których rozwiązywany problem ma trzy cechy:

1. Rozwiązanie dla przypadku rozmiaru n wymaga dużej (tzn. $poly(n)$) ilości podproblemów.
2. Redukcja, podział na podproblemy daje się wykonać szybko i podobnie skalenie.
3. Rozwiązania podproblemów można ~~stablicować~~ zapamiętać.

```
int dN_tab[0...N];
int dN(int n, int k) {
O(1)  { if(n<0 || k<0 || n<k) return 0;
        if(n==k || k==0) return 1;
        if(n>N) return -1;
O(n)  { for(i=0; i<=n; i++) dN_tab[i]=0;
O(1)    dN_tab[0]=1;
O(n^2)  for(i=1; i<=n; i++) // ta pętla liczy i-ty wiersz Δ-kwadratu
O(n)    {for(j=i; j>0; j--)
            dN_tab[j]:=dN_tab[j-1]+dN_tab[j];
        return dN_tab[k];
}
```





Koszt

czasowy $O(n^2)$

pamięciowy $O(n)$

Dziel i rządź

Podziel problem rozmiaru n na a - podproblemów rozmiaru $\frac{n}{b}$.
(gdzie $b > 1$, $a > 0$).
Następnie scal otrzymane rozwiązania w rozwiązanie oryginalnego problemu.

Szacowanie kosztu działania tej metody

Niech $T(n)$ - funkcja kosztu dla wejścia rozmiaru n.

$$\begin{cases} T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \\ T(1) = c \end{cases}$$

gdzie $f(n)$ to koszt podziału i połączenia rozwiązań.

Twierdzenie (Master Theorem)

Niech $a \geq 1$, $b > 1$, $c \geq 0$.

$$\begin{cases} T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c) \\ T(1) = \text{const.} \end{cases}$$

Teza:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \log_b(a) > c \\ \Theta(n^c \cdot \log(n)) & \log_b(a) = c \\ \Theta(n^c) & \log_b(a) < c \end{cases}$$

Twierdzenie jest też prawdziwe, gdy zamiast Θ wpiszemy $O(\cdot)$.

Przykład metody dz. i zw.

Mnożenie liczb algorytmem Karatsuby

```
  1 0 1 0 1 1 0
*   1 1 0 0 1 0 1
-------------------
    1 0 1 0 1 1 0
  0 0 0 0 0 0 0
1 0 1 0 1 1 0
0 0 0 0 0 0 0
0 0 0 0 0 0
1 0 1 0 1 1 0
+1 0 1 0 1 1 0
---------------
```

$O(n^2)$

Weźmy liczbę $x = (x_{n-1}\ x_{n-2}\ \dots\ x_0)_2$

$x_i \in \{0, 1\}$ — i-ty bit liczby x.

Załóżmy, że mnożymy dwie liczby o parzystej ilości bitów równej $n = 2k$

$y = (y_{n-1}\ y_{n-2}\ \dots\ y_0)$.

$(132)_{10} = 2 \cdot 10^0 + 3 \cdot 10^1 + 1 \cdot 10^2$

$x = \sum_{i=0}^{n-1} x_i \cdot 2^i = x_0 \cdot 2^0 + \dots + x_{k-1} 2^{k-1} + (\ )$

2^k dzieli tę część

$y = \ -1- y_i \ -1- \ $

$$x = \sum_{i=0}^{n-1} x_i 2^i = \sum_{i=0}^{k-1} x_i \cdot 2^i + 2^k \left(\sum_{i=0}^{k-1} x_{k+i} \cdot 2^i \right)$$

Np

$$x = (\overset{x_7}{1}\,0\,1\,1\,\overset{x_3}{0}\,1\,1\,\overset{x_0}{0})_2 =$$

$$= \underbrace{0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3}_{} + 1 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7$$

$$= 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 2^4 \cdot (1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3)$$

$$= (0110)_2 + (10110000)_4$$

$$= (0110)_2 + 2^4 \cdot (1011)_2 = 6 + 11 \cdot 2^4$$

$x = (x_{n-1} \dots x_0)$

$k \in \mathbb{N}$, $k \leq n$

$2^k \cdot x = (x_{n-1} \dots x_0 \underbrace{0 \dots 0}_{k\text{-zer}})_2$

Mnożenie przez potęgę 2 jest obliczane w czasie liniowym (ze względu $|x|_{lin}$ i k)

$y = (y_{n-1} \dots y_0)_2$

$x = (x_{n-1} \dots x_0)_2$

$n = 2k$

Niech $A = (x_{k-1} \dots x_0)_2$ $\quad C = (y_{k-1} \dots y_0)_2$

$B = (x_{n-1} \dots x_k)_2$ $\quad D = (y_{n-1} \dots y_k)_2$

$x = B \cdot 2^k + A \qquad y = D \cdot 2^k + C$

$x \cdot y = (B \cdot 2^k + A) \cdot (D \cdot 2^k + C) = B \cdot D \cdot 2^{2k} + 2^k (BC + AD) + AC.$

$T(2k) = 2 \cdot T(k) + O(n)$, $\rightarrow$ $a = 4$, $b = 2$, $c = 1$ $\rightarrow T(n) = O(n^2)$.

$$x \cdot y = B \cdot D \cdot 2^{2k} + 2^k \cdot ((A+B)(C+D) - B \cdot D - AC) + A \cdot C$$

$$(A+B)(C+D) - B \cdot D - AC = AC + AD + BC + BD - AC - BD = AD + BC$$

Potrzebujemy tylko 3 mnożeń: AC, $B \cdot C$, $(A+B) \cdot (C+D)$.

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n) = O(n^{\log_2 3})$$

① Złożoność mnożenia

górne ograniczenie znane (od paru lat)

$O(n \cdot \log n)$ (alg. galaktyczny)

"W praktyce lepsze" algorytmy działają w czasie

$O(n \cdot \log n \cdot f(n))$, $f(n) \simeq 2^{\log^* n} \cdot \log\log(n)$

② Nie znamy ograniczeń dolnych na mnożenie.

③ Metoda Karatsuby do zastosowań do mnożenia macierzy.