Programowanie dynamiczne

Programowanie rekurencyjne:

ZALETY: - prostota
- naturalność sformułowania

WADY: - trudność w oszacowaniu zasobów (czasu i pamięci) potrzebnych do realizacji

Czy jest możliwe wykorzystanie korzyści płynących z rekurencyjnego formułowania rozwiązań, bez używania rekurencji ?

Programowanie dynamiczne bazuje na powyższym paradoksalnym postulacie.

Nadaje się ono szczególnie do:

- problemów o charakterze numerycznym
- wyliczania skomplikowanych wartości podanych równaniem rekurencyjnym
- obliczanie najkrótszej drogi w grafach

Trzy etapy konstrukcji programu dynamicznego:

koncepcja:

- dla danego problemu stwórz rekurencyjny model rozwiązania
- stwórz tablicę, w której będzie można zapamiętywać rozwiązania przypadków elementarnych i rozwiązania pod-problemów

inicjacja:

- wpisz do tablicy wartości numeryczne, odpowiadające przypadkom elementarnym

progresja:

- na podstawie wartości numerycznych w tablicy, używając formuły rekurencyjnej, oblicz rozwiązanie problemu wyższego rzędu i wpisz je do tablicy

Porównanie metod „dziel i zwyciężaj" i „programowania dynamicznego"

„dziel i zwyciężaj"

- problem rzędu N rozłóż na pod-problemy i rozwiąż je
- połącz rozwiązania pod-problemów w celu otrzymania rozwiązania globalnego

„programowanie dynamiczne"

- mając dane rozwiązanie problemu elementarnego, policz na jego podstawie
- problem wyższego rzędu i kontynuuj obliczenia, aż do otrzymania rozwiązania rzędu N

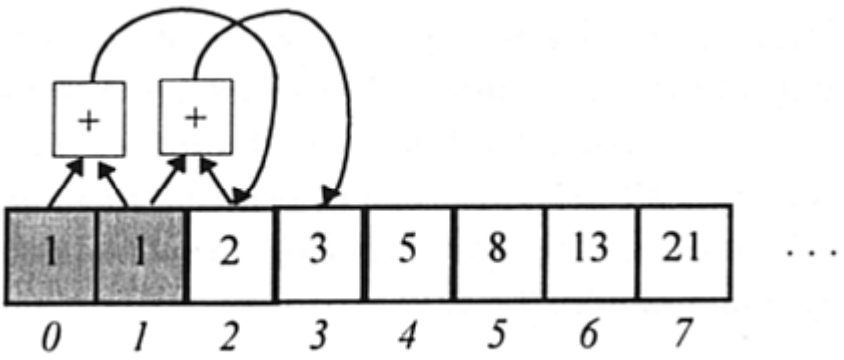
Optymalność rozwiązania - raz znalezione rozwiązanie pod-problemu zostaje zarejestrowane w tablicy i w miarę potrzeb jest wykorzystywane.

Przykład1:

Liczby Fibonacciego
Fib(0)=1
Fib(1)=1
Fib(n)=Fib(n-1) + Fib(n-2) dla n > 1

Rozwiązanie dynamiczne:
koncepcja - wzór rekurencyjny jest znany, pozostaje zadeklarować tablicę Fib(n) do składowania obliczanych wartości
inicjacja - początkowymi wartościami w tablicy Fib są wartości Fib(0)=1 i Fib(1)=1
progresja - wartością Fib(i) w tablicy (i > 1) jest suma dwóch poprzednio obliczonych wartości Fib(i-1) i Fib(i-2) zapamiętanych w tablicy

Rozwiązanie ma bardziej charakter iteracyjny niż rekurencyjny.

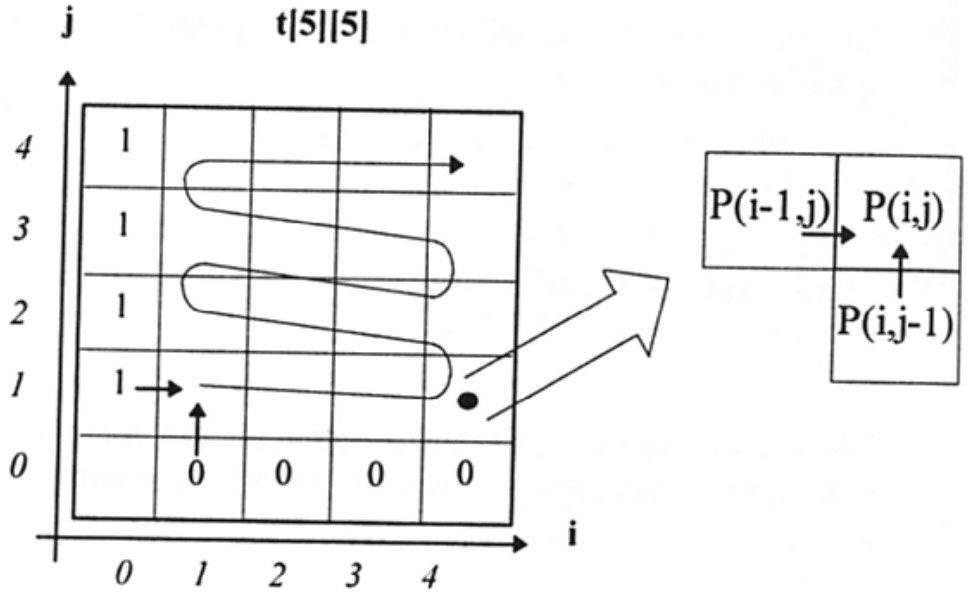



```
void fibonaci (int n , matrix fib)
{
int i;
fib[0]=1;
fib[1]=1;
for (i=1;i<=n;i++)
fib[i]=fib[i-1]+fib[i-2];
}
```

Przykład2:
Równanie rekurencyjne z więcej niż jedną zmienną:

$$P(i,j)=\begin{cases}1 & \text{jeśli } i=0 \text{ oraz } j>0,\\ 0 & \text{jeśli } i>0 \text{ oraz } j=0,\\ \dfrac{P(i-1,j)+P(i,j-1)}{2} & \text{jeśli } i>0 \text{ oraz } j>0.\end{cases}$$

Liczenie parametru P dla dwóch zmiennych i, j. Należy użyć tablicy dwuwymiarowej, której indeksy i, j oznaczają kolumny i wiersze.



Do obliczenia wartości P(i,j) potrzebna jest znajomość dwóch sąsiednich komórek: dolnej - P(i, j-1) oraz tej znajdującej się z lewej strony - P(i-1,j). Zatem naturalnym sposobem obliczania wartości P(i,j) jest posuwanie się „zygzakiem”.

```
void dynam (int n, matrix P)
{
int i,j;
  for (i=1;i<=n;i++) //inicjacja
  {
  P[i,0]=0;
  P[0,i]=1;
  }

  for (j=1;j<=n;j++)  //progresja
       for (i=1;i<=n,i++)
       P[i,j]=(P[i-1,j]+P[i,j-1])/2.0;
}
```

Program jest odbiciem wzoru rekurencyjnego, należy jedynie znaleźć prawidłowy sposób wypełniania tablicy.

Dla rekurencji dwu- i więcej wymiarowych można łatwo popełnić błąd próbując skorzystać z wartości w tablicy , które nie są jeszcze policzone.

Współczynnik dwumianowy

Współczynnik dwumianowy:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \quad \text{dla } 0 \le k \le n$$

Dla dużych wartości n i k, wartości występujące w definicji są bardzo duże.

Możemy jednak wykorzystać inną zależność:

$$D^n_k = \begin{Bmatrix} n \\ k \end{Bmatrix} = \begin{cases} \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k \end{Bmatrix} & 0 < k < n \\ 1 & k = 0 \text{ lub } k = n \end{cases}$$

Algorytm liczenia współczynnika dwumianowego metodą *dziel i zwyciężaj.*

Problem: liczenie współczynnika dwumianowego.

Dane: nieujemne liczby całkowite n i k, gdzie $k \le n$

Wynik: *bin* – wartość współczynnika dwumianowego

```
int bin(int n, int k)
{
  if(k == 0 || n == k)
     return 1;
  else
     return bin(n-1,k-1)+bin(n-1,k);
}
```

Algorytm ten wylicza $2*D^n_k - 1$ wyrazów.

Można również sformułować algorytm dynamiczny.

Wykorzystujemy tablice B taką, że $B[i][j] = D^i_j$.

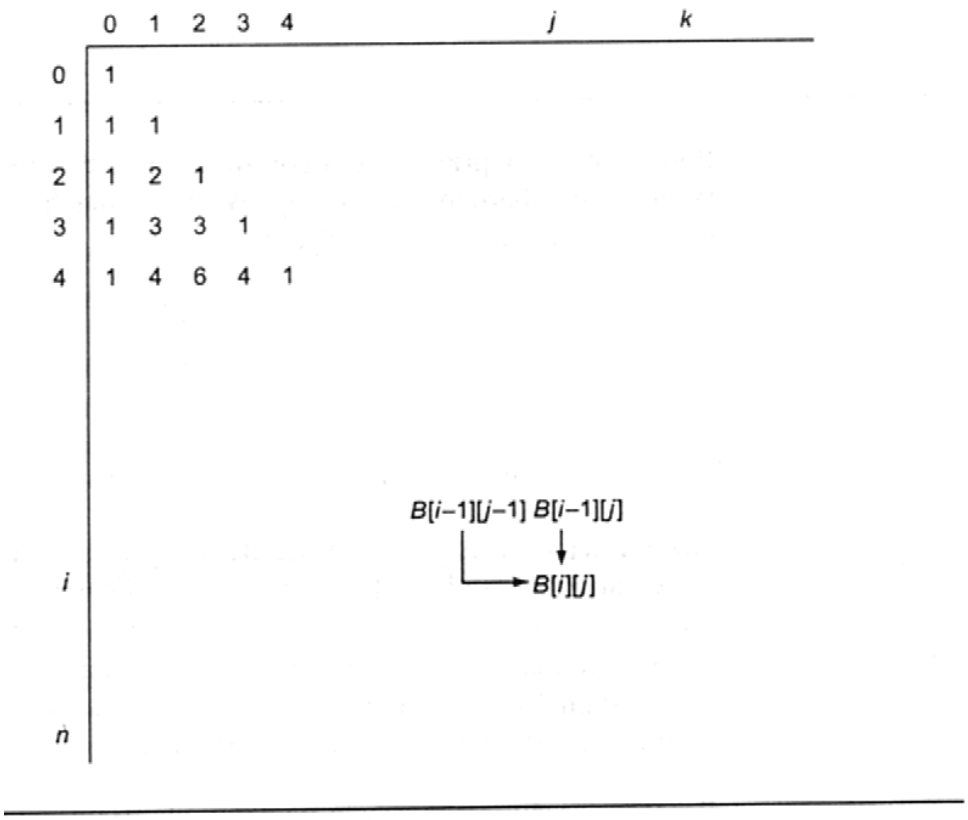
Etapy rozwiązania:

1. Określamy właściwość rekurencyjną i zapisujemy za pomocą tablicy

$$B[i][j] = \begin{cases} B[i-1][j-1]+B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ lub } j = i \end{cases}$$

2. Rozwiązujemy realizację problemu w porządku wstępującym, rozpoczynając od pierwszego wiersza i wyliczając po kolei wartości w wierszach tablicy B.

Etap 2 można zilustrować trójkątem Pascala :



Obliczenia dla B[4][2] = D^4_2

B[0][0] = 1

Wiersz 1: B[1][0] = 1
B[1][1] = 1

Wiersz 2: B[2][0] = 1
B[2][1] = B[1][0]+B[1][1] = 1+1 = 2
B[2][2] = 1

Wiersz 3: B[3][0] = 1
B[3][1] = B[2][0]+B[2][1] = 1+2 = 3
B[3][2] = B[2][1]+B[2][2] = 2+1 = 3

Wiersz 4: B[4][0] = 1
B[4][1] = B[3][0]+B[3][1] = 1+3 = 4
B[4][2] = B[3][1]+B[3][2] = 3+3 = 6

Algorytm liczenia współczynnika dwumianowego przy użyciu programowania dynamicznego.

Problem: obliczanie współczynnika dwumianowego.

Dane: nieujemne liczby całkowite *n* i *k*

Wyniki: *bin* – wartość współczynnika dwumianowego

```
int bin2(int n, int k)
{
  index i, j;
  int B[0..n][0..k];

  for(i=0; i <= n; i++)
    for(j=0; j <= minimum(i,k); j++)
       if(j == 0 || j == i)
          B[i][j] = 1;
       else
          B[i][j] = B[i-1][j-1] + B[i-1][j];
  return B[n][k];
}
```

Rozmiarem danych wejściowych jest liczba symboli użytych do ich zakodowania. Musimy zatem określić ilość wykonywanych działań w funkcji zmiennych *n* i *k*. Dla danego *n* i *k*. obliczymy liczbę przebiegów pętli `for-j`.

i	0	1	2	3	… *k*	*k+1*	…	*n*
Liczba przebiegów	0	2	3	4	… *k+1*	*k+1*	…	*k+1*

Całkowita liczba przebiegów:

$$C = 1+2+3+4+\ldots+k \quad + \quad \underbrace{(k+1)+(k+1)+\ldots+(k+1)}_{n-k+1 \text{ razy}}$$

Zatem:

$$C = k(k+1)/2 + (n-k+1)(k+1) = (2n-k+2)(k+1)/2 \Rightarrow O(nk)$$

Algorytm ten jest znacznie wydajniejszy niż zapisany metodą *dziel i zwyciężaj*.

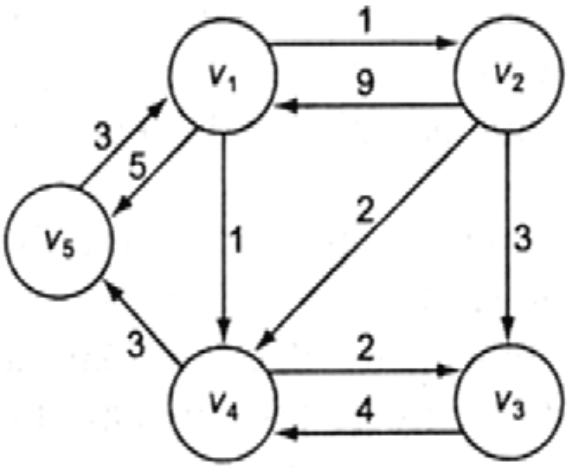
W algorytmie wykorzystano tablicę dwuwymiarową, ale realizacja z tablicą jednowymiarową $B[0\ldots k]$ jest również możliwa.

Algorytm Floyda
określanie najkrótszej drogi w grafie

Chcemy znaleźć najkrótszą drogę z jednego miejsca w drugie, gdy nie istnieje połączenie bezpośrednie.

Wprowadzenie do teorii grafów.

Przykład



Ważony graf skierowany.
Kółka – **wierzchołki grafu**.
Linie łączące wierzchołki - **krawędzie grafu**.

Wszystkie krawędzie posiadają przypisany kierunek– **graf skierowany** (digraf).
W digrafie mogą istnieć dwie krawędzie między dwoma wierzchołkami, każda biegnąca w innym kierunku.

Jeżeli krawędzie posiadają związane ze sobą wartości, są one nazywane wagami (nieujemne). Graf jest zwany **grafem ważonym**.

Droga w grafie ważonym to sekwencja wierzchołków, taka że istnieje krawędź z każdego wierzchołka do jego następnika – [v_1, v_4, v_3] jest drogą, [v_3, v_4, v_1] nie jest drogą.

Droga jest nazywana **prostą**, jeżeli nie przechodzi dwa razy przez ten sam wierzchołek. Droga prosta nigdy nie zawiera poddrogi, która byłaby cykliczna.

Długością drogi w grafie skierowanym jest suma wag krawędzi należących do drogi.

Droga z wierzchołka do niego samego – **cykl**. Graf zawierający cykl jest **grafem cyklicznym**, gdy nie zawiera cyklu jest **grafem acyklicznym**.

Najczęstsze zadanie dla grafów to znalezienie najkrótszych dróg z każdego wierzchołka do wszystkich innych wierzchołków. Najkrótsza droga musi być drogą prostą.

Przykład.
Drogi proste z v_1 do v_3 :

długość$[v_1, v_2, v_3] = 1 + 3 = 4$
długość$[v_1, v_4, v_3] = 1 + 2 = 3$ - najkrótsza
długość$[v_1, v_2, v_4, v_3] = 1 + 2 + 2 = 5$

Problem najkrótszej drogi to **problem optymalizacji**.
Dla problemu najkrótszej drogi ***rozwiązaniem kandydującym*** jest droga z jednego wierzchołka do drugiego, ***wartością*** jest długość tej drogi, ***wartością optymalną*** jest minimum tych długości. Może istnieć kilka rozwiązań.

Algorytm siłowy.
Określenie dla każdego wierzchołka długości wszystkich dróg z niego do innych wierzchołków i znalezienie minimum – czas wykonania gorszy od wykładniczego.
Załóżmy, że istnieje krawędź z jednego wierzchołka do wszystkich innych wierzchołków. Podzbiór wszystkich dróg, które rozpoczynają się od pierwszego wierzchołka i kończą na innych wierzchołkach, przechodząc przez wszystkie pozostałe. Całkowita ilość dróg wynosi wówczas – $(n-2)(n-3)\ldots1 = (n-2)!$

W wielu problemach optymalizacji algorytm siłowy jest wykładniczy lub gorszy. Zadaniem jest znalezienie bardziej wydajnego algorytmu.

Algorytm (programowanie dynamiczne):

- znajdowanie najkrótszej drogi

Reprezentacja grafu za pomocą tablicy W :

$$W[i][j] = \begin{cases} \text{waga krawędzi}, & \text{jeżeli istnieje krawędź z } v_i \text{ do } v_j \\ \infty, & \text{jeżeli nie istnieje krawędź z } v_i \text{ do } v_j \\ 0, & \text{jeżeli } i = j \end{cases}$$

Wierzchołek v_j jest **przyległy** do wierzchołka v_i , gdy istnieje krawędź z v_i do v_j .

Macierz W jest **macierzą przyległości**.

Przykład:

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

tablica W

Długości najkrótszych dróg w grafie tworzą tablicę D.

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

tablica D

Przykładowo D[*3*][*5*] = 7 – długość najkrótszej drogi z ν_3 do ν_5 .
Algorytm otrzymamy, gdy znajdziemy sposób obliczania wartości w tablicy D na podstawie wartości w tablicy W.

W tym celu tworzymy sekwencje n+1 tablic $D^{(k)}$, gdzie $0 \leq k \leq n$ oraz

$D^{(k)}[i][j]$ = długość najkrótszej drogi z ν_i do ν_j , zawierającej jako wierzchołki pośrednie tylko wierzchołki należące do zbioru $\{ \nu_1 , \nu_2 ,\ldots, \nu_k \}$

Przykładowe obliczenia wartości w tablicy D:

$D^{(0)}[2][5]$ = *długość*$[\nu_2 , \nu_5] = \infty$

$D^{(1)}[2][5]$ = *minimum*(*długość*$[\nu_2 , \nu_5]$, *długość*$[\nu_2 , \nu_1 , \nu_5]$)
= *minimum*$(\infty, 14) = 14$

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14$ {Najkrótsza droga z ν_2 nie może przechodzić przez ν_2}

$D^{(3)}[2][5] = D^{(2)}[2][5] = 14$ {Dla tego grafu są równe, bo uwzględnienie wierzchołka ν_3 nie daje nowej drogi z ν_2 do ν_5 która nie przechodzi przez ν_4}

$D^{(4)}[2][5] = minimum(długość[v_2,v_1,v_5], długość[v_2,v_4,v_5],$
$długość[v_2,v_1,v_4,v_5], długość[v_2,v_3,v_4,v_5])$
$= minimum(14,5,13,10) = 5$

$D^{(5)}[2][5] = D^{(4)}[2][5] = 5$ {Dla tego grafu są równe, bo najkrótsza droga kończąca się w v_5 nie może przechodzić przez v_5}

Wartość $D^{(5)}[2][5]$ jest długością najkrótszej drogi z v_2 do v_5, która może przechodzić przez dowolny inny wierzchołek.

$D^{(n)}[i][j]$ jest długością najkrótszej drogi z v_i do v_j , która może przechodzić przez dowolne wierzchołki, więc jest długością najkrótszej drogi z v_i do v_j .

$D^{(0)}[i][j]$ jest długością najkrótszej drogi, która nie może przechodzić przez inne wierzchołki, jest wagą przypisaną do krawędzi od v_i do v_j .

$$D^{(0)} = W \quad \text{oraz} \quad D^{(n)} = D$$

Etapy w algorytmie „dynamicznym” (otrzymywanie $D^{(n)}$ na podstawie $D^{(0)}$) :

Etap1. Określamy właściwość rekurencyjną, dzięki której możemy obliczyć $D^{(k)}$ na podstawie $D^{(k-1)}$.

Etap2. Realizujemy problem w porządku wstępującym poprzez powtarzanie procesu z **etapu1** dla $k = 1,\ldots,n$.

$$\underset{\uparrow}{D^{(0)}}, D^{(1)}, D^{(2)}, \ldots, \underset{\uparrow}{D^{(n)}}$$
$$W \qquad\qquad D$$

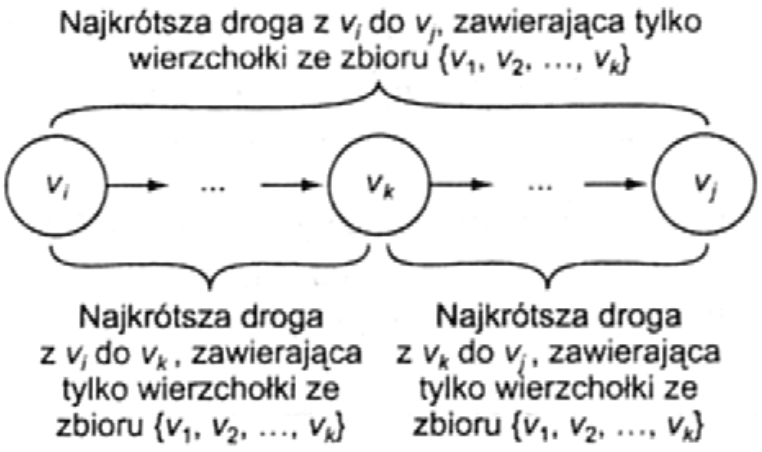
Etap1 wykonujemy, rozpatrując dwa przypadki:

Przypadek 1.
Przynajmniej jedna najkrótsza droga z ν_i do ν_j , zawierająca jako wierzchołki pośrednie tylko wierzchołki ze zbioru $\{ \nu_1 , \nu_2 ,\ldots, \nu_k \}$, nie zawiera wierzchołka ν_k .

$$D^{(k)}[i][j] = D^{(k-1)}[i][j]$$

np. $D^{(5)}[1][3] = D^{(4)}[1][3] = 3 \quad \{ \nu_1 , \nu_4 , \nu_3 \}$

Przypadek 2.
Wszystkie najkrótsze drogi z ν_i do ν_j , zawierające jako wierzchołki pośrednie tylko wierzchołki ze zbioru $\{\nu_1 , \nu_2 ,\ldots, \nu_k\}$, zawierają wierzchołek ν_k .
Dowolna najkrótsza droga ma postać:



ν_k nie może być wierzchołkiem pośrednim na poddrodze z ν_i do ν_k , wiec taka poddroga zawiera tylko wierzchołki ze zbioru $\{\nu_1 , \nu_2 ,\ldots, \nu_{k-1}\}$.

Zatem: $D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$

Przykładowo $D^{(2)}[5][3] = 7 = 4 + 3 = D^{(1)}[5][2] + D^{(1)}[2][3]$

Przypadek 1 lub 2 musi zajść wiec:

$$D^{(k)}[i][j] = minimum(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

Przypadek1 (↑ $D^{(k-1)}[i][j]$) Przypadek2 (↑ $D^{(k-1)}[k][j]$)

W celu wykonania **etapu2** wykorzystujemy właściwość rekurencyjną z **etapu1**.

Przykładowo (pamiętając że $D^{(0)} = W$):

$$D^{(1)}[2][4] = minimum(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4])$$
$$= minimum(2,9+1) = 2$$

$$D^{(1)}[5][2] = minimum(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2])$$
$$= minimum(\infty,3+1) = 4$$

$$D^{(1)}[5][4] = minimum(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4])$$
$$= minimum(\infty,3+1) = 4$$

Po wyliczeniu całej tablicy $D^{(1)}$ obliczamy tablicę $D^{(2)}$.

$$D^{(2)}[5][4] = minimum(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$$
$$= minimum(4,4+2) = 4$$

Po obliczeniu wszystkich wyrazów w tablicy $D^{(2)}$ obliczamy tablicę $D^{(3)}$.
Końcowa tablica D, zawiera długości najkrótszych dróg.

Algorytm Floyda określania najkrótszej drogi.

Dane. Ilość wierzchołków *n*, tablica W reprezentująca graf.

Wynik. Tablica D zawierająca długości najkrótszych dróg.

```
void floyd (int n, const number W[][], number D[][])
{
 index i,j,k;
 D = W;
 for (k=1; k <= n; k++)
    for (i=1; i <= n; i++)
       for (j=1; j <= n; j++)
         D[i][j] = minimum(D[i][j], D[i][k]+D[k][j])
}
```

Obliczenia wykonujemy z jedna tablica D, bo wartości w *k*-tym wierszu oraz w *k*-tej kolumnie nie są wymieniane w czasie *k*-tego przebiegu pętli.

W *k*-tym przebiegu mamy:

$$D[i][k]=minimum(D[i][k], D[i][k] + D[k][k])= D[i][k]$$

oraz

$$D[k][j]=minimum(D[k][j], D[k][k] + D[k][j])= D[k][j]$$

Złożoność czasowa algorytmu: $T(n) = n \times n \times n = n^3 \in \Theta(n^3)$

Algorytm Floyda określania najkrótszej drogi 2
(bardziej wydajny pod względem zajętości pamięci)

Dane. Ilość wierzchołków *n*, tablica W reprezentująca graf.

Wynik. Tablica D zawierająca długości najkrótszych dróg.
Tablica P:

$$P(i,j) = \begin{cases} \text{- najwyższy indeks wierzchołka pośredniego w najkrótszej drodze} \\ \text{od } v_i \text{ do } v_j \text{ , jeżeli istnieje co najmniej 1 wierzchołek pośredni} \\ \text{- 0 , jeżeli nie istnieje żaden wierzchołek pośredni} \end{cases}$$

```
void floyd2 (int n,
             const number W[][],
                   number D[][],
                   index P[][])
{
 index i,j,k;
 for (i=1; i <= n; i++)
    for (j=1; j <= n; j++)
        P[i][j] = 0;
 D = W;
 for (k=1; k <= n; k++)
    for (i=1; i <= n; i++)
       for (j=1; j <= n; j++)
         if (D[i][k]+D[k][j] < D[i][j]) {
             P[i][j] = k;
             D[i][j] = D[i][k]+D[k][j];
         }
}
```

Tablica P dla przykładu:

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

tablica P

Algorytm wyświetlania najkrótszej drogi
Problem: wyświetlanie wierzchołków pośrednich w najkrótszej drodze od jednego wierzchołka do innego w grafie ważonym.

Dane: tablica P, dwa indeksy (q, r) wierzchołków w grafie

Wynik: wierzchołki pośrednie w najkrótszej drodze z v_q do v_r

```
void path(index q, r)
{
   if (P[q][r] != 0 {
       path(q, P[q][r]);
       cout << ”v” << P[q][r];
       path(P[q][r], r);
   }
}
```