Programowanie dynamiczne, a problemy optymalizacyjne

W procesie opracowywania algorytmu programowania dynamicznego dla problemu optymalizacji wyróżniamy etapy:

Etap1. Określenie właściwości rekurencyjnej, która daje rozwiązanie optymalne dla realizacji problemu.

Etap2. Obliczenie wartości rozwiązania optymalnego w porządku wstępującym.

Etap3. Skonstruowanie rozwiązania optymalnego w porządku wstępującym

Nie jest prawdą, że problem optymalizacji może zawsze zostać rozwiązany przy użyciu programowania dynamicznego. Aby tak było w problemie musi mieć zastosowanie **zasada optymalności.**

Zasada optymalności.

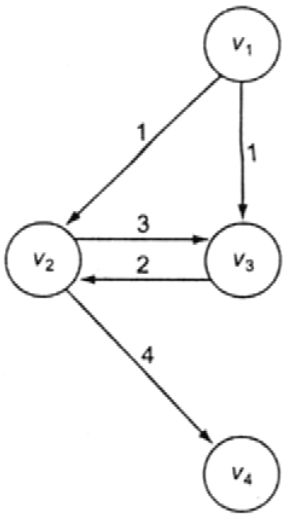
Zasada optymalności ma zastosowanie w problemie wówczas, gdy rozwiązanie optymalne realizacji problemu zawsze zawiera rozwiązania optymalne dla wszystkich podrealizacji.

Zasada optymalności w problemie najkrótszej drogi: jeżeli v_k jest wierzchołkiem należącym do drogi optymalnej z v_i do v_j , to poddrogi z v_i do v_k oraz z v_k do v_j również muszą być optymalne. Optymalne rozwiązanie realizacji zawiera rozwiązania optymalne wszystkich podrealizacji.

Jeżeli zasada optymalności ma zastosowanie w przypadku danego problemu, to można określić właściwość rekurencyjną, która będzie dawać optymalne rozwiązanie realizacji w kontekście optymalnych rozwiązań podrealizacji.

W praktyce zanim jeszcze założy się, że rozwiązanie optymalne może zostać otrzymane dzięki programowaniu dynamicznemu trzeba wykazać, że zasada optymalności ma zastosowanie.

Przykład.
Rozważmy problem znalezienia najdłuższych! prostych dróg wiodących od każdego wierzchołka do wszystkich innych wierzchołków. Ograniczamy się do prostych dróg, ponieważ w przypadku cykli zawsze możemy utworzyć dowolnie długą poprzez powtórne przechodzenie przez cykl.



Optymalna (najdłuższa) prosta droga z v_1 do v_4 to $[v_1 , v_3 , v_2 , v_4]$. Poddroga $[v_1, v_3]$ nie jest optymalną drogą z v_1 do v_3 ponieważ

$$\textit{długość}[v_1, v_3] = 1 \quad < \quad \textit{długość}[v_1, v_2 , v_3] = 4$$

Zatem zasada optymalności nie ma zastosowania. Wynika to z faktu, że optymalne drogi v_1 do v_3 oraz z v_3 do v_4 nie mogą zostać powiązane, aby utworzyć optymalną drogę z v_1 do v_4 . W ten sposób utworzymy cykl, nie optymalną drogę.

Łańcuchowe mnożenie macierzy

Załóżmy, że chcemy pomnożyć macierze o wymiarach 2x3 i 3x4 w następujący sposób:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \text{ x } \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix} = C(2,4)$$

Macierz wynikowa ma rozmiary 2x4.
Każdy element można uzyskać poprzez 3 mnożenia,
przykładowo C(1,1) = 1x7 + 2x2 + 3x6
W iloczynie macierzy występuje 2x4=8 pozycji wiec całkowita liczba elementarnych operacji mnożenia wynosi 2x4x3 = 24.

Ogólnie w celu pomnożenia macierzy o wymiarach i x j przez macierz o wymiarach j x k standardowo musimy wykonać i x j x k elementarnych operacji mnożenia.

Weźmy mnożenie:

A	x	B	x	C	x	D
20x2		2x30		30x12		12x8

Mnożenie macierzy jest łączne. Może być realizowane przykładowo: Ax(Bx(CxD)) lub (AxB)x(CxD).

Istnieje 5 różnych kolejności w których można pomnożyć 4 macierze :
Ax(Bx(CxD)) = 30x12x8 + 2x30x8 +20x2x8 = 3680

(AxB)x(CxD) = 20x2x30 + 30x12x8 +20x30x8 = 8880

Ax((BxC)xD) = 2x30x12 + 2x12x8 + 20x2x8 = 1232

((AxB)xC)xD = 20x2x30 + 20x30x12 + 20x12x8 = 10320

(Ax(BxC))xD = 2x30x12 + 20x2x12 + 20x12x8 = 3120 .

Każde mnożenie 4 macierzy wymaga innej liczby elementarnych operacji mnożenia. Trzecia kolejność jest optymalna.

Zadanie:
Opracować algorytm określający optymalną kolejność mnożenia *n* macierzy.

Kolejność mnożenia macierzy zależy tylko od rozmiarów macierzy.

Dane:
Ilość macierzy *n* oraz rozmiary macierzy.

Algorytm metodą siłową - rozważenie wszystkich kolejności i wybranie minimum.

Czas wykonania algorytmu siłowego.
Niech t_n będzie liczbą różnych kolejności mnożenia *n* macierzy: $A_1, A_2, \ldots, A_n$.
Weźmy podzbiór kolejności dla których macierz A_1 jest ostatnią mnożoną macierzą. W podzbiorze tym mnożymy macierze od A_2 do A_n, liczba różnych kolejności w tym podzbiorze wynosi t_{n-1} :

$$A_1 \text{ x } (A_2 \ldots A_n)$$
$$\uparrow\_\ t_{n-1} \text{ różnych możliwości}$$

Drugi podzbiór jest zbiorem kolejności, w przypadkach w których macierz A_n jest ostatnią mnożoną macierzą. Liczba kolejności w tym podzbiorze również wynosi t_{n-1}.

Zatem dla *n* macierzy: $t_n \geq t_{n-1} + t_{n-1} = 2\ t_{n-1}$

Natomiast dla 2 macierzy: $t_2 = 1$

Korzystając z rozwiązania równania rekurencyjnego: $t_n \geq 2^{n-2}$

Dla tego problemu ma zastosowanie zasada optymalności, tzn. optymalna kolejność mnożenia *n* macierzy zawiera optymalną kolejność mnożenia dowolnego podzbioru zbioru *n* macierzy.

Przykładowo, jeżeli optymalna kolejność mnożenia 6 macierzy jest:

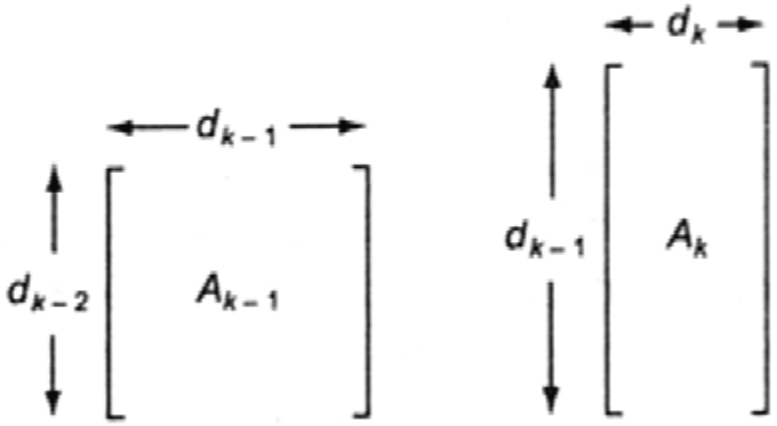
$$A_1((((A_2A_3)A_4)A_5)A_6)$$

to

$$(A_2A_3)A_4$$

Musi być optymalną kolejnością mnożenia macierzy od A_2 do A_4 .

Ponieważ mnożymy $(k\text{-}1)$-szą macierz, A_{k-1}, przez k-tą macierz, A_k, liczba kolumn w A_{k-1} musi być równa liczbie wierszy w A_k . Przyjmując, że d_0 jest liczbą wierszy w A_1, zaś d_k jest liczbą kolumn w A_k dla $1 \le k \le n$, to wymiary A_k będą wynosić d_{k-1} x d_k .



Do rozwiązania problemu wykorzystamy sekwencje tablic dla $1 \le i \le j \le n$:

M[*i*][*j*] = minimalna liczba mnożeń wymaganych do pomnożenia macierzy od A_i do A_j , jeżeli $i < j$

M[*i*][*i*] = 0

Przykład (6 macierzy):

A_1	x	A_2	x	A_3	x	A_4	x	A_5	x	A_6
5x2		2x3		3x4		4x6		6x7		7x8
$d_0\, d_1$		$d_1\, d_2$		$d_2\, d_3$		$d_3\, d_4$		$d_4\, d_5$		$d_5\, d_6$

Dla pomnożenia macierzy A_4 , A_5 , A_6 możemy określić dwie kolejności oraz liczby elementarnych operacji mnożenia:

$(A_4\, A_5)A_6$ Liczba operacji mnożenia $= d_3$ x d_4 x $d_5 + d_3$ x d_5 x d_6
$= 4$ x 6 x $7 + 4$ x 7 x $8 = 392$

$A_4\, (A_5\, A_6)$ Liczba operacji mnożenia $= d_4$ x d_5 x $d_6 + d_3$ x d_4 x d_6
$= 6$ x 7 x $8 + 4$ x 6 x $8 = 528$

Stąd: M[*4*][*6*] = *minimum*(392,528) = 392

Optymalna kolejność mnożenia 6 macierzy musi mieć jeden z rozkładów:

1. $A_1(A_2A_3A_4A_5A_6)$
2. $(A_1A_2)(A_3A_4A_5A_6)$
3. $(A_1A_2A_3)(A_4A_5A_6)$
4. $(A_1A_2A_3A_4)(A_5A_6)$
5. $(A_1A_2A_3A_4A_5)A_6$

gdzie iloczyn w nawiasie jest uzyskiwany zgodnie z optymalna kolejnością.

Liczba operacji mnożenia dla *k*-tego rozkładu jest minimalną liczba potrzebną do otrzymania każdego czynnika, powiększoną o liczbę potrzebną do pomnożenia dwóch czynników:

$$M[1][k] + M[k+1][6] + d_0\, d_k\, d_6$$

Zatem:

$$M[1][6] = \underset{1 \le k \le 5}{minimum(}M[1][k] + M[k+1][6] + d_0\, d_k\, d_6\,)$$

Uogólniając ten rezultat w celu uzyskania właściwości rekurencyjnej, związanej z mnożeniem macierzy dostajemy (dla $1 \le i \le j \le n$) :

$$M[i][j] = \underset{i \le k \le j-1}{minimum(}M[i][k] + M[k+1][j] + d_{i-1}\, d_k\, d_j\,)$$

$$M[i][i] = 0$$

Algorytm typu *dziel i zwyciężaj* oparty na tej właściwości jest wykonywany w czasie wykładniczym.

Można jednak przedstawić wydajniejszy algorytm dynamiczny liczący M[*i*][*j*] w kolejnych etapach.

Używamy siatkę podobną do trójkąta Pascala.
Element M[*i*][*j*] jest obliczany:
- na podstawie wszystkich wpisów ze swojego wiersza znajdujących się po jego lewej stronie
- wpisów ze swojej kolumny, znajdujących się poniżej niego

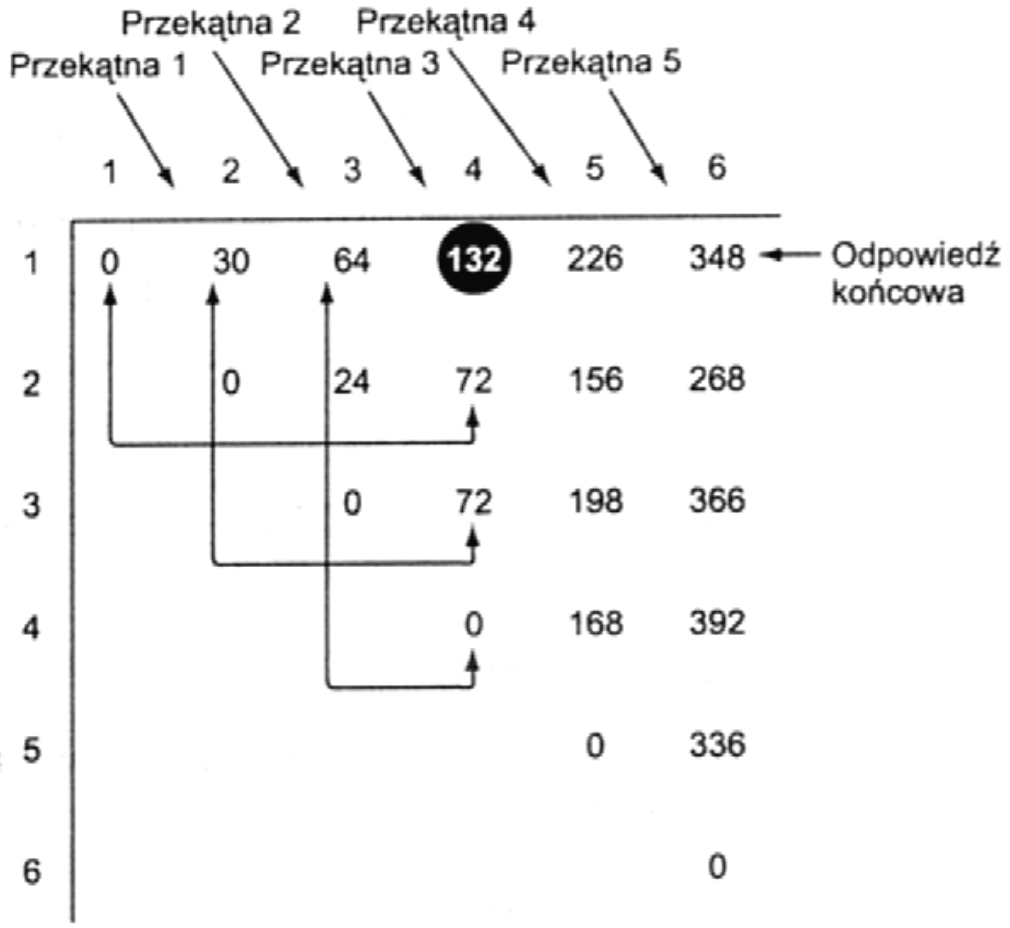
Algorytm:
- ustawiamy wartość elementów na głównej przekątnej na 0

- obliczamy wszystkie elementy na przekątnej powyżej (przekątna 1)

- obliczamy wszystkie wartości na przekątnej 2

- kontynuujemy obliczenia aż do uzyskania jedynej wartości na przekątnej 5, która jest odpowiedzią końcową M[*1*][*6*]

Przykład (6 macierzy)



Obliczamy przekątną 0:

$$M[i][i] = 0 \quad \text{dla} \quad 1 \le i \le 6$$

Obliczamy przekątną 1:

$$M[1][2] = \underset{1 \le k \le 1}{minimum}(M[1][k] + M[k+1][2] + d_0\, d_k\, d_2)$$

$$= M[1][1] + M[2][2] + d_0\, d_1\, d_2$$
$$= 0 + 0 + 5 \text{ x } 2 \text{ x } 3 = 30$$

Wartości $M[2][3]$, $M[3][4]$, $M[4][5]$, $M[5][6]$ liczymy podobnie.

Obliczamy przekątną 2:

$$M[1][3] = \underset{1 \le k \le 2}{minimum}(M[1][k] + M[k+1][3] + d_0\, d_k\, d_3)$$

$$= minimum(M[1][1] + M[2][3] + d_0\, d_1\, d_3 ,$$
$$M[1][2] + M[3][3] + d_0\, d_2\, d_3)$$
$$= minimum(0 + 24 + 5 \text{ x } 2 \text{ x } 4,\ 30 + 0 + 5 \text{ x } 3 \text{ x } 4) = 64$$

Wartości M[*2*][*4*] , M[*3*][*5*] , M[*4*][*6*] liczymy podobnie.

Obliczamy przekątną 3:

$$M[1][4] = \underset{1 \le k \le 3}{minimum}(M[1][k] + M[k+1][4] + d_0\, d_k\, d_4)$$

$$= minimum(M[1][1] + M[2][4] + d_0\, d_1\, d_4 ,$$
$$M[1][2] + M[3][4] + d_0\, d_2\, d_4 ,$$
$$M[1][3] + M[4][4] + d_0\, d_3\, d_4)$$

$$= minimum(0 + 72 + 5 \text{ x } 2 \text{ x } 6,\ 30 + 72 + 5 \text{ x } 3 \text{ x } 6,$$
$$64 + 0 + 5 \text{ x } 4 \text{ x } 6) = 132$$

Wartości M[*2*][*5*] , M[*3*][*6*] liczymy podobnie.

Przekątną 4 liczymy podobnie : M[*1*][*5*] = 226
Przekątną 5 liczymy podobnie : M[*1*][*6*] = 348

Algorytm: minimalna liczba operacji mnożenia

Problem : określić minimalną liczbę elementarnych operacji mnożenia, wymaganych w celu pomnożenia *n* macierzy oraz kolejność wykonywania mnożeń, która zapewnia minimalną liczbę operacji.

Dane: liczba macierzy *n* oraz tablica liczb całkowitych *d*, indeksowana od 0 do *n*, gdzie *d*[*i-1*] x *d*[*i*] jest rozmiarem *i*-tej macierzy.

Wynik : *minmult* – minimalna liczba elementarnych operacji mnożenia, wymaganych w celu pomnożenia *n* macierzy; dwuwymiarowa tablica *P*, na podstawie której można określić optymalną kolejność. *P*[*i*][*j*] jest punktem, w którym macierze od *i* do *j* zostaną rozdzielone w kolejności optymalnej dla mnożenia macierzy.

```
int minmult(int n, const int d[], index P[][])
{
 index i,j,k,diagonal;
 int M[1..n][1..n];

 for (i=1; i ≤ n; i++)
     M[i][i]=0;
 for (diagonal = 1; diagonal ≤ n-1; diagonal++)
    for (i=1; i ≤ n - diagonal; i++)
    {
       j=i+diagonal;
      M[i][j]= minimum (M[i][k]+M[k+1][j]+
               i ≤ k ≤ j-1  d[i-1]*d[k]*d[j] )

      P[i][j] = wartość k, która dała minimum
    }
return M[1][n];
}
```

Złożoność czasowa – minimalna liczba operacji mnożenia.

Operacją podstawową są instrukcje wykonywane dla każdej wartości *k*, w tym sprawdzenie czy wartość jest minimalna.

Rozmiar danych: *n*- liczba macierzy do pomnożenia.
Mamy do czynienia z pętlą w pętli. Ponieważ $j = i + diagonal$ dla danych *diagonal* oraz *i*.
Liczba przebiegów pętli *k* wynosi

$$j - 1 - i + 1 = i + diagonal - 1 - i + 1 = diagonal$$

Dla danej wartości *diagonal* liczba przebiegów pętli *for-i* wynosi *n-diagonal*.
Diagonal może przyjmować wartości od 1 do *n - 1* , całkowita liczba powtórzeń operacji podstawowej wynosi

$$\sum_{diagonal=1}^{n-1} [(n - diagonal)\ x\ diagonal] = n(n-1)(n+1)/6 \in \Theta(n^3)$$

Przykład:

$P[2][5] = 4$ oznacza optymalną kolejność mnożenia $(A_2\ A_3\ A_4)\ A_5$
Punkt 4 jest punktem rozdzielenia macierzy w celu otrzymania czynników.

Mając tablicę *P:*

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

możemy odczytać:

$P[1][6] = 1$ -> $A_1\ (A_2\ A_3\ A_4\ A_5\ A_6)$
$P[2][6] = 5$ -> $A_1\ ((A_2\ A_3\ A_4\ A_5)\ A_6)$
$P[2][5] = 4$ -> $A_1\ (((A_2\ A_3\ A_4)\ A_5)\ A_6)$
$P[2][4] = 3$ -> $A_1\ ((((A_2\ A_3)\ A_4)\ A_5)\ A_6)$

Algorytm: wyświetlanie optymalnej kolejności

Problem: wyświetlić optymalną kolejność dla mnożenia *n* macierzy.
Dane: dodatnia liczba całkowita *n* oraz tablica *P*
Wynik: optymalna kolejność mnożenia macierzy

```
void order(index i, index j)
{
  if (i==j)
    cout << "A" << i;
  else {
    k = P[i][j];
    cout << "(";
    order(i,k);
    order(k+1,j);
    cout << ")";
  }
}
```

Optymalne drzewa wyszukiwania binarnego

Opracowujemy algorytm określania optymalnego sposobu zorganizowania zbioru elementów w postaci drzewa wyszukiwania binarnego.

Dla każdego wierzchołka w drzewie binarnym poddrzewo, którego korzeniem jest lewy (prawy) potomek tego wierzchołka, nosi nazwę ***lewego (prawego) poddrzewa*** wierzchołka.

Lewe (prawe) poddrzewo korzenia drzewa nazywamy lewym (prawym) poddrzewem drzewa.

Drzewo wyszukiwania binarnego.

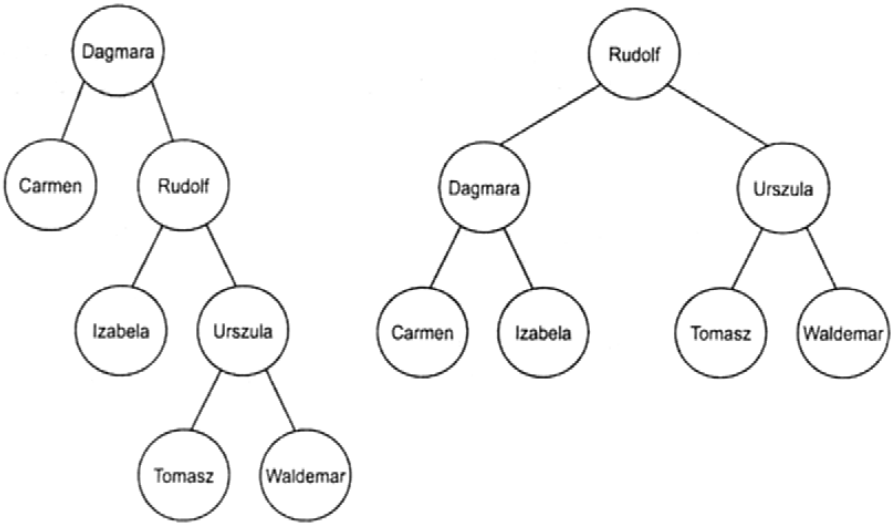
Drzewo wyszukiwania binarnego jest binarnym drzewem elementów (nazywanych kluczami) pochodzących ze zbioru uporządkowanego.
Musi spełniać warunki:

1. Każdy wierzchołek zawiera jeden klucz.

2. Każdy klucz w lewym poddrzewie danego wierzchołka jest mniejszy lub równy kluczowi tego wierzchołka.

3. Klucze znajdujące się w prawym poddrzewie danego wierzchołka są większe lub równe kluczowi tego wierzchołka.

Przykład.



Dwa drzewa o tych samych kluczach. W lewym drzewie prawe poddrzewo wierzchołka *Rudolf* zawiera klucze (imiona) *Tomasz*, *Urszula*, *Waldemar* wszystkie większe od *Rudolf* zgodnie z porządkiem alfabetycznym.
Zakładamy, że klucze są unikatowe.

Głębokość wierzchołka w drzewie jest liczbą krawędzi w unikatowej drodze, wiodącej od korzenia do tego wierzchołka, inaczej zwana ***poziomem wierzchołka*** w drzewie.

Głębokość drzewa to maksymalna głębokość wszystkich wierzchołków (w przykładzie - drzewo po lewej głębokość 3, po prawej głębokość 2)

Drzewo nazywane jest ***zrównoważonym***, jeżeli głębokość dwóch poddrzew każdego wierzchołka nigdy nie różni się o więcej niż 1 (w przykładzie – lewe drzewo nie jest zrównoważone, prawe jest zrównoważone).

Zwykle drzewo wyszukiwania binarnego zawiera pozycje, które są pobierane zgodnie z wartościami kluczy. Celem jest takie zorganizowanie kluczy w drzewie wyszukiwania binarnego, aby średni czas zlokalizowania klucza był minimalny. Drzewo zorganizowane w ten sposób jest nazywane optymalnym.

Jeżeli wszystkie klucze charakteryzuje to samo prawdopodobieństwo zostania kluczem wyszukiwania, to drzewo z przykładu (prawe) jest optymalne.

Weźmy przypadek, w którym wiadomo, że klucz wyszukiwania występuje w drzewie. Aby zminimalizować średni czas wyszukiwania musimy określić złożoność czasową operacji lokalizowania klucza.

Algorytm wyszukiwania klucza w drzewie wyszukiwania binarnego

Wykorzystujemy strukturę danych:

```
struct nodetype
{
   keytype key;
   nodetype* left;
   nodetype* right;
};
typedef nodetype* node_pointer;
```

Zmienna typu *node_pointer* jest wskaźnikiem do rekordu typu *nodetype*.

Problem: określić wierzchołek zawierający klucz w drzewie wyszukiwania binarnego, zakładając że taki występuje w drzewie.

Dane: wskaźnik *tree* do drzewa wyszukiwania binarnego oraz klucz *keyin*.

Wynik: wskaźnik *p* do wierzchołka zawierającego klucz.

```
void search(node_pointer tree,
            keytype keyin,
            node_pointer& p)
{
  bool found;

  p = tree;
  found = false;
  while (!found)
     if (p->key == keyin)
         found = true;
     else if (keyin < p->key)
          p = p->left;
          else
          p = p->right;
}
```

Liczbę porównań wykonywanych przez procedurę *search* w celu zlokalizowania klucza możemy nazwać **czasem wyszukiwania**. Chcemy znaleźć drzewo, dla którego średni czas wyszukiwania jest najmniejszy.

Zakładając, że w każdym przebiegu pętli *while* wykonywane jest tylko jedno porównanie możemy napisać :

czas wyszukiwania = *głębokość*(*key*) + 1

Przykładowo (lewe poddrzewo):

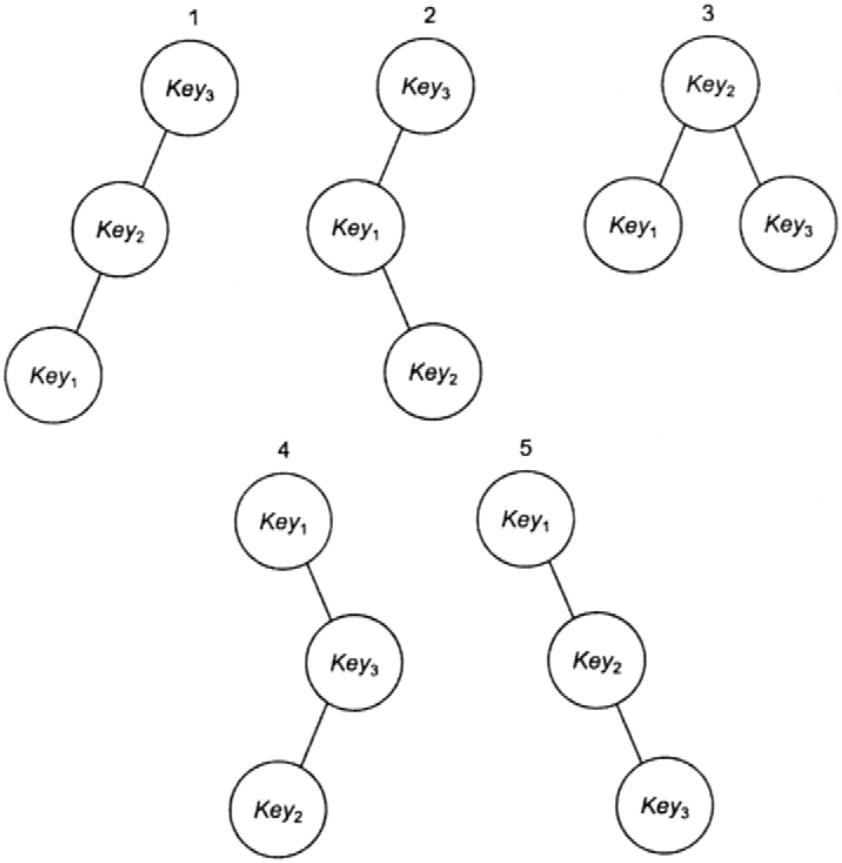
czas wyszukiwania = *głębokość*(*Urszula*) + 1 = 2+1 = 3

Niech Key_1, Key_2, …, Key_n będą n uporządkowanymi kluczami oraz p_i będzie prawdopodobieństwem tego, że Key_i jest kluczem wyszukiwania. Jeżeli c_i oznacza liczbę porównań koniecznych do znalezienia klucza Key_i w danym drzewie, to:

$$\text{średni czas wyszukiwania} = \sum_{i=1}^{n} c_i \, p_i$$

Jest to wartość która trzeba zminimalizować.

Przykład.



Mamy 5 różnych drzew dla $n = 3$. Wartości kluczy nie są istotne.

Jeżeli:

$$p_1 = 0.7 \quad , \quad p_2 = 0.2 \quad \text{oraz} \quad p_3 = 0.1$$

to średnie czasy wyszukiwania dla drzew wynoszą :

1. $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
2. $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
3. $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
4. $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
5. $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

Piąte drzewo jest optymalne.

Oczywiście znalezienie optymalnego drzewa wyszukiwania binarnego poprzez rozpatrzenie wszystkich drzew wiąże się z ilością drzew co najmniej wykładniczą w stosunku do n.
W drzewie o głębokości n-1 wierzchołek na każdym z n-1 poziomów (oprócz korzenia) może się znajdować na prawo lub lewo. Zatem liczba różnych drzew o głębokości n-1 wynosi 2^{n-1}

Załóżmy, że klucze od Key_i do Key_j są ułożone w drzewie, które minimalizuje wielkość:

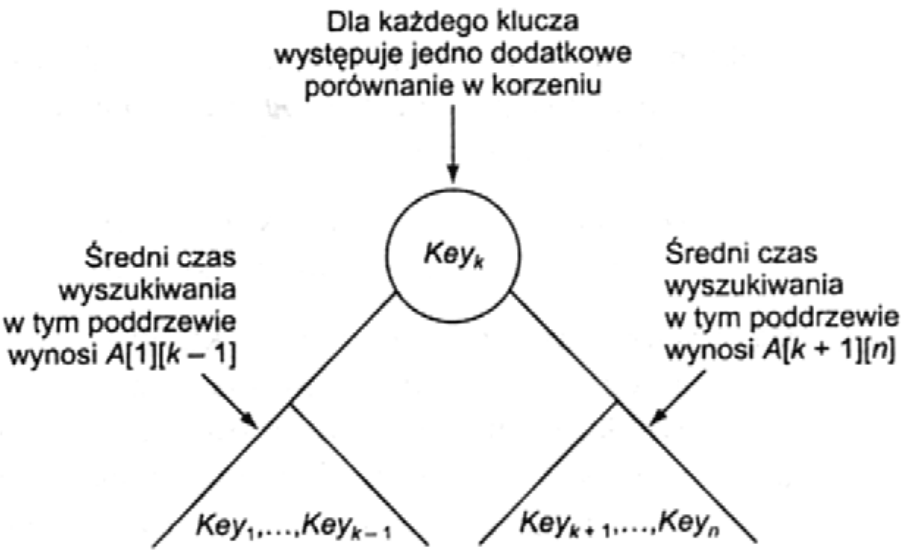
$$\sum_{m=i}^{j} c_m p_m$$

gdzie c_m jest liczbą porównań wymaganych do zlokalizowania klucza Key_m w drzewie. Drzewo to nazywamy optymalnym.

Wartość optymalną oznaczymy jako A[i][j] oraz A[i][i]=p_i (jeden klucz wymaga jednego porównania).

Korzystając z przykładu można pokazać, że w problemie tym zachowana jest zasada optymalności.

Możemy sobie wyobrazić n różnych drzew optymalnych: drzewo 1 w którym Key_1 jest w korzeniu, drzewo 2 w którym Key_2 jest w korzeniu, … , drzewo n w którym Key_n jest w korzeniu. Dla $1 \le k \le n$ poddrzewa drzewa k muszą być optymalne, więc czasy wyszukiwania w tych poddrzewach można opisać:



Dla każdego $m \neq k$ wymagana jest o jeden większa liczba porównań w celu zlokalizowania klucza Key_m w drzewie k niż w celu zlokalizowania tego klucza w poddrzewie w którym się znajduje. Dodatkowe porównanie jest związane z korzeniem i daje 1 x p_m do średniego czasu wyszukiwania.

Średni czas wyszukiwania dla drzewa k wynosi

$$\underbrace{A[1][k-1]}_{\text{Średni czas w lewym poddrzewie}} + \underbrace{p_1 + \ldots p_{k-1}}_{\text{Dodatkowy czas związany z porównaniem w korzeniu}} + \underbrace{p_k}_{\text{Średni czas wyszukania korzenia}} + \underbrace{A[k+1][n]}_{\text{Średni czas w prawym poddrzewie}} + \underbrace{p_{k+1} + \ldots p_n}_{\text{Dodatkowy czas związany z porównaniem w korzeniu}}$$

lub inaczej

$$A[1][k\text{-}1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$

Jedno z k drzew musi być optymalne więc średni czas wyszukiwania optymalnego drzewa określa zależność:

$$A[1][n] = minimum(A[1][k\text{-}1] + A[k+1][n]) + \sum_{m=1}^{n} p_m$$

gdzie $A[1][0]$ i $A[n+1][n]$ są z definicji równe 0.

Uogólniamy definicje na klucze od Key_i do Key_j , gdzie $i < j$ i otrzymujemy:

$$A[i][j] = \underset{i \le k \le j}{minimum}(A[i][k\text{-}1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \quad i < j$$

$$A[i][i] = p_i$$

$A[i][i\text{-}1]$ oraz $A[j+1][j]$ są z definicji równe 0.

Wyliczenia prowadzimy podobnie jak w algorytmie łańcuchowego mnożenia macierzy.

Algorytm znajdowania optymalnego drzewa przeszukiwania binarnego.

Problem: określenie optymalnego drzewa wyszukiwania binarnego dla zbioru kluczy, z których każdy posiada przypisane prawdopodobieństwo zostania kluczem wyszukiwania.

Dane: *n*-liczba kluczy oraz tablica liczb rzeczywistych *p* indeksowana od 1 do *n*, gdzie *p*[*i*] jest prawdopodobieństwem wyszukiwania *i*-tego klucza

Wyniki: zmienna *minavg*, której wartością jest średni czas wyszukiwania optymalnego drzewa wyszukiwania binarnego oraz tablica R, z której można skonstruować drzewo optymalne. *R*[*i*][*j*] jest indeksem klucza znajdującego się w korzeniu drzewa optymalnego, zawierającego klucze od *i*-tego do *j*-tego.

```
void optsearch(int n, const float p[],
               float& minavg, index R[][])
{
   index i, j, k, diagonal;
   float A[1..n+1][0..n];

   for (i=1; i <= n; i++) {
       A[i][i-1] = 0;
       A[i][i] = p[i];
       R[i][i] = i;
       R[i][i-1] = 0;
   }
   A[n+1][n] = 0;
   for(diagonal = 1; diagonal <= n-1; diagonal++)
    for(i = 1; i <= n - diagonal; i++)  //Przekatna 1
      {                                 //tuz nad glowna przek
       j = i + diagonal;

       A[i][j]=minimum(A[i][k-1]+A[k+1][j] + Σ_{m=i}^{j} p_m ;
               i ≤ k ≤ j

       R[i][j]= wartość k, która dała minimum;
   }
  minavg = A[1][n];
}
```

Złożoność czasową można określić podobnie jak dla mnożenia łańcuchowego macierzy:

$$T(n) = n(n-1)(n+1)/6 \in \Theta(n^3)$$

Algorytm budowania optymalnego drzewa przeszukiwania binarnego.

Problem: zbudować optymalne drzewo wyszukiwania binarnego.

Dane: *n* – liczba kluczy, tablica *Key* zawierająca *n* uporządkowanych kluczy oraz tablica *R*, utworzona w poprzednim algorytmie. *R*[*i*][*j*] jest indeksem klucza w korzeniu drzewa optymalnego, zawierającego klucze od *i*-tego do *j*-tego

Wynik: wskaźnik *tree* do optymalnego drzewa wyszukiwania binarnego, zawierającego *n* kluczy.

```
node_pointer tree(index i,j)
{
   index k;
   node_pointer p;

   k = R[i][j];
   if(k == 0)
      return NULL;
   else
   {
   p = new nodetype;
   p->key = Key[k];
   p->left = tree(i,k-1);
   p->right = tree(k+1,j);
   return p;
   }
}
```

Przykład.
Załóżmy, że mamy następujące wartości w tablicy *Key:*

Damian	Izabela	Rudolf	Waldemar
Key[1]	*Key[2]*	*Key[3]*	*Key[4]*

oraz

$p_1 = 3/8$ $p_2 = 3/8$ $p_3 = 1/8$ $p_4 = 1/8$

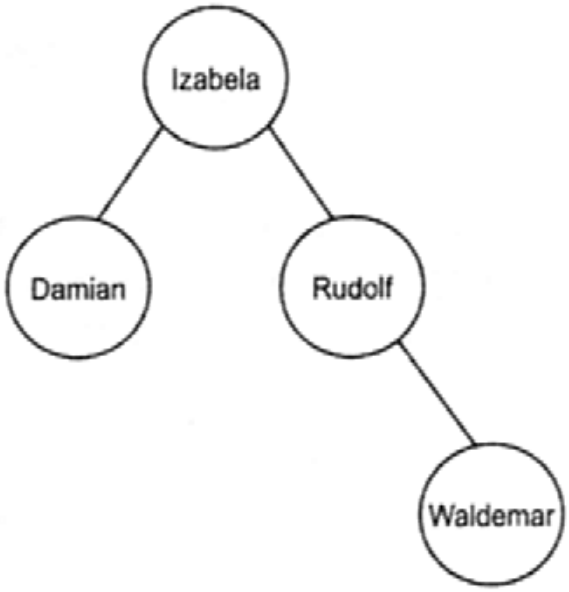
Tablice *A* i *R* będą wówczas wyglądać:

	0	1	2	3	4
1	0	3/8	9/8	11/8	7/4
2		0	3/8	5/8	1
3			0	1/8	3/8
4				0	1/8
5					0

A

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

R


Izabela
Damian
Rudolf
Waldemar

Problem komiwojażera.

Komiwojażer planuje podróż, która uwzględnia odwiedzenie 20 miast. Każde miasto jest połączone z niektórymi innymi miastami. Chcemy zminimalizować czas czyli musimy określić najkrótszą trasę, która rozpoczyna się w mieście początkowym, przebiega przez wszystkie miasta i kończy w punkcie startu.

Problem określania najkrótszej trasy nosi nazwę problemu komiwojażera.

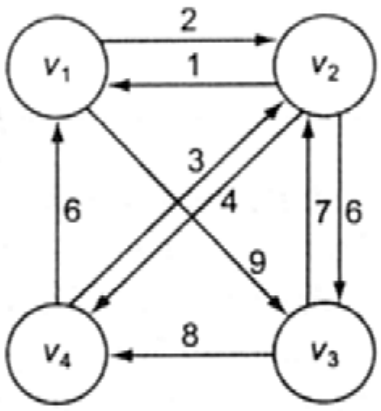
Problem może być reprezentowany przez graf ważony, z wierzchołkami-miastami.

Trasa (droga Hamiltona) w grafie skierowanym jest drogą wiodącą z wierzchołka do niego samego, przechodzącą przez wszystkie wierzcholki dokładnie raz.

Optymalna trasa w ważonym grafie skierownym jest taką drogą, która posiada najmniejszą długość.

Problem polega na na znalezieniu optymalnej trasy w ważonym grafie skierowanym, kiedy instnieje przynajmniej jedna trasa.

Wierzchołek początkowy to v_1.



Możemy przykładowo opisać trzy trasy:

$$length[v_1, v_2, v_3, v_4, v_1] = 22$$
$$length[v_1, v_3, v_2, v_4, v_1] = 26$$
$$length[v_1, v_3, v_4, v_2, v_1] = 21$$

Ostatnia trasa jest optymalna.

Najprostsza realizacja polega na rozważeniu wszystkich tras.
W ogólnym przypadku może istnieć krawędź łącząca każdy wierzchołek z każdym innym wierzchołkiem. Drugi wierzchołek na trasie może być jednym z *n-1* wierzchołków, trzeci wierzchołek – jednym spośród *n-2* wierzchołków, *n*-ty wierzchołek – ostatnim wierzchołkiem.

Zatem całkowita liczba tras wynosi - $(n-1)(n-2)\ldots 1 = (n-1)!$
co oznacza wartość gorszą od wykładniczej.

Czy można zastosować programowanie dynamiczne ?

Jeżeli v_k jest pierwszym wierzchołkiem po v_1 na trasie optymalnej, to droga podrzędna tej trasy z v_k do v_1 musi być drogą najkrótszą, przechodzącą przez wszystkie pozostałe wierzchołki dokładnie raz. Zatem zasada optymalności działa i można stosować programowanie dynamiczne.

Graf reprezentuje macierz przyległości W:

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

W rozwiązaniu:
V = zbiór wszystkich wierzchołków
A = podzbiór zbioru V
$D[\nu_i][A]$ = długość najkrótszej drogi z ν_i do ν_1 przechodzącej przez każdy wierzchołek A dokładnie raz

Zatem w przykładzie: $V = \{\nu_1, \nu_2, \nu_3, \nu_4\}$ – reprezentuje zbiór,
[ ] – reprezentuje drogę

Jeżeli $A = \{\nu_3\}$, to

$$D[\nu_2][A] = length[\nu_2, \nu_3, \nu_1] = \infty$$

Jeżeli $A = \{\nu_3, \nu_4\}$, to

$$D[\nu_2][A] = minimum(length[\nu_2, \nu_3, \nu_4, \nu_1], length[\nu_2, \nu_4, \nu_3, \nu_1])$$
$$= minimum(20, \infty) = 20$$

Zbiór $V - \{\nu_1, \nu_j\}$ zawiera wszystkie wierzchołki oprócz ν_1 oraz ν_j i ma zastosowanie zasada optymalności, możemy stwierdzić:

$$Długość\ trasy\ minimalnej = \min_{2 \le j \le n}(W[1][j] + D[\nu_j][V - \{\nu_1, \nu_j\}])$$

i ogólnie dla $i \neq 1$ oraz ν_i nie należącego do A

$$D[\nu_i][A] = \min_{j:\ \nu j \in A}(W[i][j] + D[\nu_j][A - \{\nu_j\}]) \quad \text{jeżeli } A \neq \varnothing$$

$$D[\nu_i][\varnothing] = W[i][1]$$

Określmy optymalną trasę dla grafu z przykładu.
Dla zbioru pustego:

$$D[\nu_2][\varnothing] = 1$$
$$D[\nu_3][\varnothing] = \infty$$
$$D[\nu_4][\varnothing] = 6$$

Teraz rozważamy wszystkie zbiory zawierające jeden element:

$$D[\nu_3][\{\nu_2\}] = minimum(W[3][j] + D[\nu_j][\{\nu_2\}-\{\nu_j\}])$$
$$= W[3][2] + D[\nu_2][\varnothing] = 7 + 1 = 8$$

Podobnie:

$$D[\nu_4][\{\nu_2\}] = 3 + 1 = 4$$
$$D[\nu_2][\{\nu_3\}] = 6 + \infty = \infty$$
$$D[\nu_4][\{\nu_3\}] = \infty + \infty = \infty$$
$$D[\nu_2][\{\nu_4\}] = 6 + 4 = 10$$
$$D[\nu_3][\{\nu_4\}] = 8 + 6 = 14$$

Teraz rozważamy wszystkie zbiory zawierające dwa elementy:

$$D[\nu_4][\{\nu_2, \nu_3\}] = \underset{j:\nu j \in \{\nu 2,\, \nu 3\}}{minimum}(W[4][j] + D[\nu_j][\{\nu_2,\nu_3\} - \{\nu_j\}])$$

$$= minimum(W[4][2]+D[\nu_2][\{\nu_3\}],\ W[4][3]+D[\nu_3][\{\nu_2\}])$$
$$= minimum(3+\infty,\ \infty+8) = \infty$$

Podobnie:

$$D[\nu_3][\{\nu_2, \nu_4\}] = minimum(7+10,\ 8+4) = 12$$
$$D[\nu_2][\{\nu_3, \nu_4\}] = minimum(6+14,\ 4+\infty) = 20$$

Na końcu liczymy długość optymalnej trasy:

$$D[\nu_1][\{\nu_2, \nu_3, \nu_4\}] = minimum(W[1][j] + D[\nu_j][\{\nu_2,\nu_3, \nu_4\} - \{\nu_j\}])$$

$$= minimum(W[1][2] + D[\nu_2][\{\nu_3,\nu_4\}],$$
$$W[1][3] + D[\nu_3][\{\nu_2,\nu_4\}],$$
$$W[1][4] + D[\nu_4][\{\nu_2,\nu_3\}])$$
$$= minimum(2+20,\ 9+12,\ 26/\infty) = 21$$

Algorytm programowania dynamicznego dla problemu komiwojażera

Problem: określić optymalną trasę w ważonym grafie skierowanym. Wagi są liczbami nieujemnymi.

Dane wejściowe: ważony graf skierowany oraz *n*, liczba wierzchołków w grafie. Graf reprezentujemy macierzą przyległości W *W*[*i*][*j*] reprezentuje wagę krawędzi od wierzchołka *i*-tego do *j*-tego.

Wynik: zmienna *minlength*, której wartością jest długość optymalnej trasy oraz macierz P, na podstawie której konstruujemy optymalną trasę. Wiersze tablicy P są indeksowane od 1 do *n*, zaś jej kolumny są indeksowane przez wszystkie podzbiory zbioru V-{ν_1}. Element P[i][A] jest indeksem pierwszego wierzchołka, znajdującego się po {ν_i} na najkrótszej drodze z ν_i do ν_1 , która przechodzi przez wszystkie wierzchołki A dokładnie raz.

```
void komiwojazer(int n,
                 const number W[][],
                 index P[][],
                 number& minlength)
{
 index i,j,k;
 number D[1..n][podzbior zbioru V-{ν1}];

 for(i=2;i<=n;i++)
     D[i][∅] = W[i][1];

 for(k=1;k<=n-2;k++)
  for(wszystkie podzbiory A∈V-{ν1} z k wierzch)
   for(i,takie ze i≠1 oraz νi nie należy do A){
    D[i][A]=minimum(W[i][j]+D[j][A-{νj}]);
                j: νj ∈ A
    P[i][A] = wartość j, która daje minimum;
   }
```

```
D[1][V-{ν1}] = minimum(W[1][j]+D[j][V-{ν1,νj});
                2 ≤ j ≤ n
P[1][V-{ν1}] = wartość j, która daje minimum;
minlength = D[1][V-{ν1}];
}
```

Elementy tablicy *P*, wymagane do określenia optymalnej trasy dla grafu z przykładu to:

$$P[1,\{\nu_2, \nu_3, \nu_4\}] \quad P[3,\{\nu_2, \nu_4\}] \quad P[4,\{\nu_2\}]$$

Optymalną trasę można uzyskać:

Indeks pierwszego wierzchołka = $P[1][\{\nu_2, \nu_3, \nu_4\}] = 3$

Indeks drugiego wierzchołka = $P[3][\{\nu_2, \nu_4\}] = 4$

Indeks trzeciego wierzchołka = $P[4][\{\nu_2\}] = 2$

Optymalna trasa ma postać:

$$\{\nu_1, \nu_3, \nu_4, \nu_2, \nu_1\}$$

Dotychczas nie opracowano algorytmu dla problemu komiwojażera, którego złożoność w najgorszym przypadku byłaby lepsza niż wykładnicza.