Notatki z AiSD. Nr 6. 25 marca 2020

Programowanie dynamiczne

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

1 Wstęp

Zastosowanie metody Dziel i Zwyciężaj do problemów zdefiniowanych rekurencyjnie jest w zasadzie ograniczone do przypadków, gdy podproblemy, na które dzielimy problem, są niezależne. W przeciwnym razie metoda ta prowadzi do wielokrotnego obliczania rozwiązań tych samych podproblemów. Jednym ze sposobów zaradzenia temu zjawisku jest tzw. *spamiętywanie*, polegające na pamiętaniu rozwiązań podproblemów napotkanych w trakcie obliczeń. W przypadku, gdy przestrzeń wszystkich możliwych podproblemów jest nieduża, efektywniejsze od spamiętywania może być zastosowanie metody programowania dynamicznego. Polega ona na obliczaniu rozwiązań dla wszystkich podproblemów, począwszy od podproblemów najprostszych.

Przykład 1.
Problem:
Dane: Liczby naturalne n, k.
Wynik: $\binom{n}{k}$.

Naturalna metoda redukcji problemu obliczenia $\binom{n}{k}$ korzysta z zależności $\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k}$. Zastosowanie metody Dziel i Zwyciężaj byłoby jednak w tym przypadku nierozważne, ponieważ w trakcie liczenia $\binom{n-1}{k-1}$ jak i $\binom{n-1}{k}$ wywoływalibyśmy rekurencyjnie procedurę dla tych samych danych (tj. dla $n-2$ i $k-1$), co w konsekwencji prowadzi do tego, że niektóre podproblemy byłyby rozwiązywane wykładniczą liczbę razy. Poniższa procedura unika tego wykorzystując tablicę $tab[1..n, 1..k]$ do spamiętywania.

```
for i=1 to n do
    for j = 0 to k do tab_{i,j} ← "?"
        ..................
function nPOk(n, k)
    if k = n or k = 0 then tab_{k,n} ← 1; return 1;
    if tab_{n-1,k-1} = "?" then tab_{n-1,k-1} ← nPOk(n − 1, k − 1)
    if tab_{n-1,k} = "?" then tab_{n-1,k} ← nPOk(n − 1, k)
    tab_{n,k} = tab_{n-1,k-1} + tab_{n-1,k}
    return tab_{n,k}
```

Rekurencyjne obliczanie $\binom{n}{k}$ z użyciem spamiętywania

Za zastosowaniem w tym przypadku programowania dynamicznego przemawia fakt, iż liczba różnych podproblemów, jakie mogą pojawić się w trakcie obliczania $\binom{n}{k}$ jest niewielka, a mianowicie $O(n^2)$. Podobnie jak w metodzie spamiętywania, algorytm dynamiczny oblicza początkowy fragment trójkąta Pascala i umieszcza go w tablicy *tab*. W przeciwieństwie jednak do poprzedniej metody, która jest metodą "top-down" i jest implementowana rekurencyjnie, algorytm dynamiczny jest metodą "bottom-up" i jest implementowany iteracyjnie. To pozwala w szczególności na wyeliminowanie kosztów związanych z obsługą rekursji.

for $i = 1$ **to** n **do** $tab_{i,0} \leftarrow 1$
....
function nPOk(n, k)
  for $j = 1$ **to** k **do**
    $tab_{j,j} \leftarrow 1$
    for $i = j + 1$ **to** n **do** $tab_{i,j} \leftarrow tab_{i-1,j-1} + tab_{i-1,j}$
  return $tab_{n,k}$

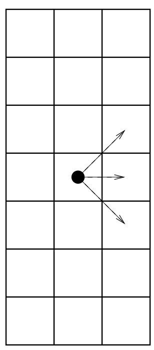
Obliczanie $\binom{n}{k}$ metodą programowania dynamicznego

Fakt, że metoda programowania dynamicznego oblicza w sposób systematyczny rozwiązania wszystkich podproblemów, pozwala często na poczynienie dodatkowych oszczędności w stosunku do metody spamiętywania. W tym przykładzie możemy znacznie zredukować koszty pamięciowe. Jak łatwo zauważyć, obliczenie kolejnej przekątnej trójkąta Pascala wymaga znajomości jedynie wartości z poprzedniej przekątnej. Tak więc zamiast tablicy $n \times k$ wystarcza tablica $n \times 2$, a nawet tablica $n \times 1$. □

Podobnie jak w przypadku metody dziel i zwyciężaj, kluczem do zastosowania programowania dynamicznego jest znalezienie takiego sposobu dzielenia problemu na podproblemy, by optymalne rozwiązanie problemu można było w prosty sposób otrzymać z optymalnych rozwiązań podproblemów. Wskazaniem na zastosowanie wówczas programowania dynamicznego a nie metody dziel i zwyciężaj jest sytuacja, gdy sumaryczny rozmiar podproblemów jest duży. Oczywiście, jak już wspominaliśmy, aby algorytm dynamiczny był efektywny, przestrzeń wszystkich możliwych podproblemów nie może być zbyt liczna.

PRZYKŁAD 2.
PROBLEM:
Dane: Tablica $\{a_{i,j}\}$ liczb nieujemnych $(i = 1, \ldots, n;\ j = 1, \ldots, m)$
Wynik: Ciąg indeksów $i_1, \ldots, i_m$ taki, że $\forall_{j=1,\ldots,m-1} |i_j - i_{j+1}| \leq 1$, minimalizujący sumę $\sum_{j=1}^{m} a_{i_j,j}$

INTERPRETACJA: Ciąg $i_1, \ldots, i_m$ wyznacza trasę wiodącą od pierwszej do ostatniej kolumny tablicy a. Startujemy z dowolnego pola pierwszej kolumny i kończymy na dowolnym polu ostatniej kolumny. W każdym ruchu przesuwamy się o jedno pole: albo w prawo na wprost albo w prawo na ukos (jak pokazano na rysunku 1). Chcemy znaleźć trasę o minimalnej długości rozumianej jako suma liczb z pól znajdujących się na trasie.



Rysunek 1: *Możliwe kierunki ruchu w tablicy a.*

Jak łatwo sprawdzić liczba wszystkich prawidłowych tras jest wykładnicza, więc rozwiązanie siłowe nie wchodzi w rachubę.

Rozważmy najpierw nieco prostsze zadanie, polegające na znalezieniu długości optymalnej trasy. Potem pokażemy w jaki sposób zorganizować obliczenia, by wyznaczenie samej trasy było proste.

Niech $d_{i,k}$ oznacza minimalną długość trasy wiodącej od dowolnego pola pierwszej kolumny do pola $a_{i,k}$, a $P(i,k)$ - problem wyznaczenia $d_{i,k}$. Rozwiązanie $P(i,k)$ (dla $k > 1$) można łatwo otrzymać z rozwiązań trzech prostszych podproblemów, a mianowicie $P(i-1,k-1)$, $P(i,k-1)$ i $P(i+1,k-1)$ (w przypadku $P(1,k)$ i $P(n,k)$ - dwóch podproblemów). Problem spełnia więc wymagane kryterium optymalności.

Jeśli za rozmiar $P(i,k)$ przyjmiemy wartość k, to problem rozmiaru k redukujemy do trzech podproblemów rozmiaru $k-1$. To zbyt skromna redukcja, by stosować metodę dziel i zwyciężaj. Z drugiej strony przestrzeń wszystkich podproblemów jest stosunkowo niewielka - składa się z nm elementów (zawiera wszystkie $P(i,j)$ dla $i = 1,\ldots,n,\ j = 1,\ldots,m$), możemy więc zastosować programowanie dynamiczne.

for $j = 1$ **to** m **do** $d_{0,j} \leftarrow d_{n+1,j} \leftarrow \infty$
for $i = 1$ **to** n **do** $d_{i,1} \leftarrow a_{i,1}$
for $j = 2$ **to** m **do**
    for $i = 1$ **to** n **do** $d_{i,j} \leftarrow a_{i,j} + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i+1,j-1}\}$
return $\min\{d_{i,m} \mid i = 1,\ldots,n\}$

Pozostaje wyjaśnić, w jaki sposób można odtworzyć optymalną trasę. Niech i_0 będzie wartością i, dla której osiągane jest $\min\{d_{i,m} \mid i = 1,\ldots,n\}$, a więc $a_{i_0,m}$ jest ostatnim polem optymalnej trasy. Aby wyznaczyć przedostatnie pole wystarczy sprawdzić, która z trzech wartości $d_{j,m-1}$ (dla $j \in \{i_0-1, i_0, i_0+1\}$) jest minimalna. Postępując dalej rekurencyjnie wyznaczymy całą trasę.

procedure trasa(i,j)
{ **if** $j = 1$ **then return** i
    if $d_{i-1,j-1} < d_{i,j-1}$ **then** $k \leftarrow i-1$ **else** $k \leftarrow i$
    if $d_{k,j-1} < d_{i+1,j-1}$ **then** $k \leftarrow i+1$
    return *concat*(trasa$(k, j-1), i)$
}
....
write(trasa(i_0, m))

□

Programowanie dynamiczne jest częstą metodą rozwiązywania problemów optymalizacyjnych. Przykład 2 stanowi ilustrację klasycznego sposobu rozwiązania takiego problemu: najpierw znajdujemy wartość optymalnego rozwiązania a dopiero potem, na podstawie wyliczeń tej wartości, konstruujemy optymalne rozwiązanie.

2 Dalsze przykłady

2.1 Najdłuższy wspólny podciąg.

2.1.1 Definicja problemu

Definicja 1 *Ciąg* $Z = \langle z_1, z_2, \ldots, z_k\rangle$ *jest* podciągiem *ciągu* $X = \langle x_1, x_2, \ldots, x_n\rangle$*, jeśli istnieje ściśle rosnący ciąg indeksów* $\langle i_1, i_2, \ldots, i_k\rangle$ $(1 \le i_j \le n)$ *taki, że*

$$\forall_{j=1,2,\ldots k}\ \ x_{i_j} = z_j.$$

Jeśli Z jest podciągiem zarówno ciągu X jak i ciągu Y, to mówimy, że Z jest wspólnym podciągiem *ciągów X i Y.*

KONWENCJA: Dla wygody, w dalszej części ciągi będziemy traktować jako napisy nad ustalonym alfabetem.

PRZYKŁAD:
'BABA' jest wspólnym podciągiem ciągów 'ABRACADABRA' i 'RABARBAR', ale nie jest ich najdłuższym wspólnym podciągiem (dłuższym jest np. 'RAAAR').

□

OZNACZENIA:

- $LCS(X, Y) = \{Z \mid Z$ jest wspólnym podciągiem X i Y o maksymalnej długości$\}$
- przez X_i oznaczamy i-literowy prefiks ciągu $X = \langle x_1, x_2, \ldots, x_n \rangle$, tj. podciąg $\langle x_1, x_2, \ldots, x_i \rangle$; w szczególności przez X_0 oznaczamy ciąg pusty.

PROBLEM:
Dane: ciągi $X = \langle x_1, x_2, \ldots, x_m \rangle$ i $Y = \langle y_1, y_2, \ldots, y_n \rangle$
Wynik: dowolny ciąg Z z $LCS(X, Y)$

2.1.2 Redukcja problemu

Problem znalezienia ciągu Z z $LCS(X, Y)$ możemy zredukować do prostszych problemów na podstawie następującej obserwacji:

- jeśli ostatnia litera X i ostatnia litera Y są takie same, to litera ta musi być ostatnim elementem każdego ciągu z $LCS(X, Y)$.
- jeśli X i Y różnią się na ostatniej pozycji (tj, $x_m \neq y_n$), to istnieje ciąg w $LCS(X, Y)$, który na ostatniej pozycji ma literę różną od x_m lub istnieje ciąg w $LCS(X, Y)$, który na ostatniej pozycji ma literę różną od y_n.

W pierwszym przypadku problem znalezienia ciągu z $LCS(X_m, Y_n)$ redukujemy do podproblemu znalezienia ciągu z $LCS(X_{m-1}, Y_{n-1})$. Rozwiązaniem będzie konkatenacja znalezionego ciągu i ostatniej litery X-a. W drugim przypadku problem redukujemy do dwóch podproblemów: znalezienie ciągu z $LCS(X_{m-1}, Y_n)$ i znalezienie ciągu z $LCS(X_m, Y_{n-1})$. W tym przypadku rozwiązaniem będzie dłuższy ze znalezionych ciągów.

2.1.3 Algorytm

Najpierw koncentrujemy się na obliczeniu wartości rozwiązania optymalnego, którą w tym przypadku jest długość elementów z $LCS(X, Y)$. Sposobu na obliczenie tej wartości dostarcza nam obserwacja poczyniona w poprzednim paragrafie.

Fakt 1 *Niech $d_{i,j}$ oznacza długość elementów z $LCS(X_i, Y_j)$. Wówczas:*

$$d_{i,j} = \begin{cases} 0 & \textit{jeśli } i = 0 \textit{ lub } j = 0, \\ 1 + d_{i-1,j-1} & \textit{jeśli } i, j > 0 \textit{ i } x_i = y_j, \\ \max(d_{i,j-1}, d_{i-1,j}) & \textit{jeśli } i, j > 0 \textit{ i } x_i \neq y_j \end{cases}$$

□

Tablicę d możemy obliczać kolejno wierszami (lub kolumnami), a wynik odczytamy z $d_{m,n}$.

```
Procedure LCS(X_m, Y_n)
    for i ← 1 to m do d_{i,0} ← 0
    for j ← 0 to n do d_{0,j} ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if x_i = y_j then d_{i,j} ← 1 + d_{i-1,j-1}
            else d_{i,j} ← max{d_{i-1,j}, d_{i,j-1}}
```

Aby wypisać jakiś element z LCS musimy przejść tablicę d jeszcze raz, począwszy od elementu $d_{n,m}$, w podobny sposób jak to robiliśmy w Przykładzie 2.

Jeśli zależy nam na szybkości algorytmu, możemy nieco przyspieszyć tę jego fazę. W tym celu, w trakcie obliczania tablicy d, możemy w dodatkowej tablicy zapamiętywać "drogę dojścia" do poszczególnych elementów tablicy d. Elementy dodatkowej tablicy przyjmowałyby jedną z trzech różnych wartości, w zależności od tego czy $d_{i,j}$ powstał przez dodanie 1 do $d_{i-1,j-1}$, czy przez przepisanie $d_{i-1,j}$, czy też wreszcie przez przepisanie $d_{i,j-1}$.

2.1.4 Koszt algorytmu

Obliczenie każdego elementu tablicy d odbywa się w czasie stałym. Tak więc całkowity koszt wypełnienia tablicy d jest równy $\Theta(n \cdot m)$. Koszt skonstruowania najdłuższego podciągu na podstawie tablicy d jest liniowy.

2.2 Wyznaczanie optymalnej kolejności mnożenia macierzy.

2.2.1 Definicja problemu

Mamy obliczyć wartość wyrażenia $\mathcal{M}$ postaci $M_1 \times M_2 \times \cdots \times M_n$, gdzie M_i są macierzami. Zakładamy, że wyrażenie jest poprawne, tj. liczba kolumn macierzy M_i jest równa liczbie wierszy macierzy M_{i+1} (dla $i = 1, \ldots, n-1$).

Ponieważ mnożenie macierzy jest działaniem łącznym, wartość $\mathcal{M}$ możemy liczyć na wiele sposobów. Wybór sposobu może w istotny sposób wpłynąć na liczbę operacji skalarnych jakie wykonamy podczas obliczeń.

Przykład Niech macierze M_1, M_2, M_3 mają wymiary odpowiednio $d \times 1$, $1 \times d$ i $d \times 1$. Rozważmy dwa sposoby obliczenia ich iloczynu:

- $(M_1 \times M_2) \times M_3$
  W wyniku pierwszego mnożenia otrzymujemy macierz $d \times d$, więc jego koszt (niezależnie od przyjętej metody mnożenia macierzy) wynosi co najmniej d^2. W drugim mnożeniu także musimy wykonać $\Theta(d^2)$ operacji.
- $M_1 \times (M_2 \times M_3)$
  Koszt obliczenia $M_2 \times M_3$ wynosi $O(d)$. W jego wyniku otrzymujemy macierz 1×1, więc koszt następnego mnożenia wynosi także $O(d)$.

□

Dalsze rozważania będziemy przeprowadzać przy następującym założeniu[1]:

Koszt pomnożenia macierzy o wymiarach $a \times b$ i $b \times c$ wynosi abc.

[1] Jest to koszt mnożenia wykonanego metodą tradycyjną; później poznamy inne, szybsze metody.

PROBLEM:

Dane: $d_0, d_1, \ldots, d_n$ - liczby naturalne
INTERPRETACJA: $d_{i-1} \times d_i$ - wymiar macierzy M_i.

Zadanie: Wyznaczyć kolejność mnożenia macierzy $M_1 \times M_2 \times \cdots \times M_n$, przy której koszt obliczenia tego iloczynu jest minimalny.

2.2.2 Rozwiązanie siłowe

Rozwiązanie siłowe, polegające na sprawdzeniu wszystkich możliwych sposobów wykonania obliczeń, jest nieakceptowalne. Liczba tych sposobów dana jest wzorem

$$S(n) = \begin{cases} 1 & \text{jeśli } n = 1 \\ \sum_{i=1}^{n-1} S(i)S(n-i) & \text{jeśli } n > 1 \end{cases}$$

UZASADNIENIE WZORU: Każde z $n-1$ mnożeń występujących w ciągu $M_1 \times \ldots \times M_n$, może być ostatnim, jakie wykonamy obliczając ten iloczyn. Liczba sposobów mnożenia macierzy, w których i-te mnożenie jest ostatnim, jest równa iloczynowi $S(i) \cdot S(n-i)$ (tj. liczby sposobów, na które można pomnożyć i pierwszych macierzy oraz liczby sposobów, na które można pomnożyć $n-i$ ostatnich macierzy).

Rozwiązaniem powyższego równania jest $S(n)$ ="$n-$ta liczba Catalana"$= \frac{1}{n}\binom{2n-2}{n-1} = \Omega(\frac{4^n}{n^2})$. Tak więc koszt sprawdzania wszystkich możliwych iloczynów jest wykładniczy.

2.2.3 Rozwiązanie dynamiczne

Zauważamy, że problem spełnia kryterium optymalności. Jeśli bowiem k-te mnożenie jest ostatnim jakie wykonamy w optymalnym sposobie obliczeń, to iloczyny $M_1 \times \ldots \times M_k$ oraz $M_{k+1} \times \ldots \times M_n$ też musiały być obliczone w optymalny sposób.

Na podstawie tej własności możemy ułożyć następujący algorytm rekurencyjny wyznaczający optymalny koszt obliczeń.

```
function matmult(i, j)
    if i = j then return 0
    opt ← ∞
    for k ← i to j − 1 do
        opt ← min(opt, d_{j−1} d_k d_j + matmult(i, k) + matmult(k + 1, j))
    return opt
```

Algorytm ten, jakkolwiek szybszy od metody siłowej, nadal działa w czasie wykładniczym ($\Theta(3^n)$). Przyczyna tkwi w wielokrotnym wykonywaniu obliczeń dla tych samych wartości parametrów (i, j). Unikniemy tego mankamentu stosując programowanie dynamiczne. Niech

$$m_{i,j} = \text{"minimalny koszt obliczenia } M_i \times M_{i+1} \times \cdots \times M_j\text{"}$$

Dla wygody przyjmujemy, że $m_{i,j} = 0$ (dla $i \geq j$). Wówczas

$$m_{i,j} = \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + d_{i-1}d_kd_j).$$

Składnik $m_{i,k}$ jest kosztem obliczenia $M_i \times M_{i+1} \times \cdots \times M_k$, składnik $m_{k+1,j}$ - kosztem obliczenia $M_{k+1} \times M_{k+2} \times \cdots \times M_j$, natomiast $d_{i-1}d_kd_j$ to koszt obliczenia iloczynu dwóch powstałych macierzy.

```
procedure dyn − matmult(d[0..n]);
    int m[1..n, 1..n],  p[1..n, 1..n]
     for  i ← 1 to n do m_ii ← 0;
     for  s ← 1 to n − 1 do
         for  i ← 1 to n − s do
             j ← i + s
             m_ij ← min_{i≤k<j}(m_{i,k} + m_{k+1,j} + d_{i−1} d_k d_j)
             p_ij ← "to k, przy którym osiągane było minimum dla m_ij"
     return  p[1..n, 1..n]
```

Algorytm oblicza wartości $m_{i,i+s}$ (na podstawie powyższego wzoru) oraz wartości $p_{i,i+s}$, które umożliwiają późniejsze skonstruowanie rozwiązania.

Koszt algorytmu. Tablicę $m_{i,j}$ liczymy przekątna za przekątną począwszy od głównej przekątnej. Koszt policzenia jednego elementu $m_{i,i+l}$ na s-tej przekątnej wynosi $\Theta(s)$. Ponieważ na s-tej przekątnej znajduje się $n-s$ elementów, koszt algorytmu wynosi

$$T(n) = \sum_{s=0}^{n-1} \Theta(s) \cdot (n-s) = \Theta(n^3).$$

Odtworzenie rozwiązania Odtworzenia rozwiązania dokonujemy w standardowy sposób na podstawie tablicy p. Zwróć uwagę, że znalezienie rozwiązania na podstawie samych tylko wartości m_{ij} wymagałoby czasu $\Theta(n^2)$.