TEA

1. **Analýza časovej zložitosti algoritmov**. (Definícia časovej zložitosti. O-notácia. Odhad časovej zložitosti rekurzívnych algoritmov používajúcich metódu rozdeľ a panuj.)

1.9 Formálne značenie

Keď rozprávame o časovej zložitosti algoritmov, zväčša chceme zhora ohraničiť, ako rýchlo rastie nejaká funkcia (ktorej presné vyjadrenie často nepoznáme). Za týmto účelom si požičiame notáciu z matematickej analýzy. Najčastejšie používaným symbolom bude veľké písmeno O, ktorého význam je definovaný nasledovne:

Nech f a g sú rastúce funkcie na prirodzených číslach. Potom hovoríme, že f patrí do $O(g)$, ak existuje kladná konštanta c taká, že pre všetky dostatočne veľké n platí $f(n) \leq c \cdot g(n)$. Tento zápis čítame „funkcia f je veľké O od funkcie g“. Preložené z matematickej reči, tento zápis hovorí, že funkcia f rastie **nanajvýš rádovo** tak rýchlo ako funkcia g.

Formálne, veľké O predstavuje triedu (t.j. množinu) funkcií – teda $O(g)$ je trieda všetkých funkcií, ktoré rastú nanajvýš rádovo tak rýchlo ako funkcia g.

Navyše si toto značenie budeme často zjednodušovať: pod O(výraz obsahujúci n) budeme rozumieť „$O(f)$, kde f je funkcia definovaná predpisom $\forall n : f(n) =$ výraz obsahujúci n“. Teda napríklad „$O(n^2)$“ je to isté ako „$O(f)$, kde $\forall n : f(n) = n^2$“. Obe tieto značenia predstavujú *triedu všetkých funkcií, ktoré rastú nanajvýš rádovo tak rýchlo ako kvadratická funkcia.*

Ďalšie bežne používané symboly sú Ω a Θ.

Platí $f \in \Omega(g)$ práve vtedy, keď $g \in O(f)$. Omega teda predstavuje dolný odhad: $f \in \Omega(g)$ nám hovorí, že f rastie **aspoň rádovo** tak rýchlo ako g.

No a $f \in \Theta(g)$ platí vtedy, ak platí $f \in O(g)$ a zároveň $f \in \Omega(g)$ – inými slovami, keď funkcie f a g rastú rádovo rovnako rýchlo.

2. **Algoritmy pre triedenie**. (Efektívne algoritmy triedenia porovnávaním. Triedenie v lineárnom čase. Dolný odhad časovej zložitosti každého triedenia porovnávaním.)

3. **Dátové štruktúry v poli**. (Pole s dynamickou veľkosťou – vektor. Zásobník, fronta. Binárna halda a implementácia prioritnej fronty pomocou nej.)

4. **Usporiadané dátové štruktúry**. (Binárne vyhľadávacie stromy. Usporiadaná množina, usporiadané asociatívne pole – slovník. Vyvažovanie binárnych stromov.)

5. **Hešovanie**. (Kolízie a rôzne spôsoby ich riešenia. Narodeninový paradox. Množina, asociatívne pole.)

6. **Základné grafové algoritmy**. (Reprezentácie grafu v pamäti. Prehľadávanie do hĺbky a do šírky. Topologické triedenie.)

7. **Najkratšie cesty v grafe**. (Dijkstrov algoritmus, Floydov-Warshallov algoritmus.)

2.2 Najlacnejšie cesty v grafe

Definícia 2.5 *Nech $G = (V, E)$ je orientovaný graf s ohodnotenými hranami, tj. pre G je daná funkcia $h : E \to R$. Cesta v grafe G je postupnosť vrcholov $[v_0, v_1, \ldots, v_k]$, kde $(v_{i-1}, v_i) \in E$ pre $i = 1, 2, \ldots, k$ a $v_i \neq v_j$ pre $i \neq j$. Cena cesty $P = [v_0, v_1, \ldots, v_k]$ je $\sum_{i=1}^{k} h(v_{i-1}, v_i)$. Označme cenu cesty P symbolom $|P|$.*

V súvislosti s cenou cesty nás budú zaujímať tri problémy:

1. Nájsť pre danú dvojicu vrcholov u, v cenu najlacnejšej cesty z u do v.
2. Nájsť pre daný vrchol $v_0 \in V$ cenu najlacnejšej ciesty z v_0 do v pre všetky $v \in V$.
3. Nájsť cenu najlacnejšej cesty z u do v pre všetky $u, v \in V$.

2.2.1 Dijkstrov algoritmus

Ak sú ceny hrán grafu nezáporné reálne čísla, potom možno problém 2 riešiť Dijkstrovým algoritmom. Algoritmus dostane ako vstup orientovaný graf $G(V, E)$, vrchol v_0 a čiastočnú funkciu $h : V \times V \to R_0^+$. Predpokladáme, že $h(u, u) = 0$, ak $(u, v) \in E$ tak $h(u, v)$ je ohodnotenie hrany (u, v). Vrcholy grafu sú reprezentované celými číslami $1, 2, \ldots |V|$ a predpokladáme, že funkciu h možno vypočítať v čase $O(1)$. Po skončení algoritmu bude pre každý vrchol $v \in V$ v $D[v]$ uložená cena najlacnejšej cesty z v_0 do v.

Poznámka: Pre jednoduchosť čiastočnú funkciu h zúplníme tak, že v prípade $(u, v) \notin E$ položíme $h(u, v) = \infty$. V tomto zmysle potom pre ľubovoľnú postupnosť vrcholov $[v_0, v_1, \ldots, v_k]$ vieme vypočítať cenu zodpovedajúcej cesty, pričom ak pre nejaké i $(0 \leq i < k)$ platí $(v_i, vi + 1) \notin E$, cena uvedenej cesty bude ∞.

Algoritmus 8 (Dijkstra)

begin

$S \leftarrow \{v_0\}$ *(1)*
$D[v_0] \leftarrow 0$ *(2)*
pre každý vrchol $v \in V \setminus \{v_0\}$: $D[v] \leftarrow h(v_0, v)$ *(3)*
while $S \neq V$ **do begin** *(4)*
  vyber $w \in V \setminus S$ *taký, že hodnota* $D[w]$ *je minimálna* *(5)*
  $S \leftarrow S \cup \{w\}$ *(6)*
  pre každý $v \in V \setminus S$: *(7)*
    $D[v] \leftarrow min\{D[v], D[w] + h(w, v)\}$ *(8)*
end

end

Veta 2.6 *Algoritmus 8 vypočíta cenu najlacnejšej cesty z v_0 do každého vrcholu grafu G v najhoršom prípade v čase $O(|V|^2)$.*

Dôkaz:

Časová zložitosť: Riadok (5) a tiež aj cyklus v riadkoch (7) a (8) potrebujú čas $O(|V|)$. Riadok (6) potrebuje čas $O(1)$. Keďže riadky (5) až (8) sú vykonané $(|V|-1)$ krát a riadky (1) až (3) potrebujú čas $O(|V|)$, na vykonanie algoritmu stačí čas $O(|V|^2)$.

Korektnosť: Korektnosť algoritmu ukážeme metódou invariantov. Stanovíme invariant, ktorý bude platný pred začatím každej iterácie cyklu while (riadky (4) až (8)) resp. po jeho skončení. Pre každé $v \in V$:

1. Ak $v \in S$, potom $D[v]$ je cena najlacnejšej cesty z v_0 do v, pričom existuje cesta z v_0 do v celá ležiaca v S s cenou $D[v]$.

2. Ak $v \in V \setminus S$, potom $D[v]$ je cena najlacnejšej cesty z v_0 do v spomedzi ciest, ktoré celé s výnimkou vrcholu v ležia v S.

Platnosť invariantu dokážeme indukciou vzhľadom na $|S|$.

Pre $|S| = 1$ (tj. pri prvom prechode) má najlacnejšia cesta z v_0 do v_0 cenu 0 a cesta z v_0 do v celá s výnimkou vrcholu v ležiaca v množine S pozostáva z hrany (v_0, v).

Nech ďalej invariant platí pre $|S| = k$. Nech w je vrchol, ktorý vyberieme na základe podmienky v riadku (5). Najprv sporom ukážeme, že $D[w]$ je cena najlacnejšej cesty z v_0 do w.

Nech teda existuje cesta P z v_0 do w, kde $|P| < D[w]$. Podľa indukčného predpokladu je $D[w]$ cena najlacnejšej cesty z v_0 do w spomedzi takých ciest, ktoré celé okrem vrcholu w ležia v S. Preto musí na ceste P existovať vrchol (rôzny od w), ktorý nepatrí do S. Nech v je prvý takýto vrchol. Označme Q úsek cesty P od v_0 po v. S výnimkou vrcholu v ležia všetky vrcholy cesty Q v množine S. Potom ale podľa indukčného predpokladu musí platiť $D[v] \leq |Q|$. Súčasne, keďže ceny hrán sú nezáporné, platí $|Q| \leq |P| < D[w]$ a teda $D[v] < D[w]$, čo je v spore s podmienkou výberu vrcholu w v riadku (5). Preto $D[w]$ musí byť cena najlacnejšej cesty z v_0 do w. Zvyšné časti tvrdenia invariantu sú zrejmé.

V každom kroku cyklu pridáme do množiny S práve jeden vrchol a teda po konečnom počte krokov cyklus skončí. Z platnosti invariantu po skončení cyklu priamo vychádza správnosť algoritmu 8. □

Na začiatku je potrebné položiť pre každé v $P[v] = v_0$. Ak modifikujeme $D[v]$ v riadku (8) algoritmu 8, je potrebné modifikovať príslušným spôsobom aj pole P, tj. riadok (8) nahradíme takto:

if $D[w] + h(w, v) < D[v]$ **then begin**
  $D[v] \leftarrow D[w] + h(w, v)$
  $P[v] \leftarrow w$
end

2.2.2 Floyd–Warshallow algoritmus

V tejto časti ukážeme algoritmus, ktorý rieši problém 3 v čase $O(|V|^3)$.

Je daný orientovaný graf $G = (V, E)$ s cenami hrán z R, pričom sa v ňom nenachádza cyklus zápornej ceny. Nech je graf G reprezentovaný incidenčnou maticou $W = (w_{ij})$, pričom ak $(i, j) \in E$, potom w_{ij} je cena hrany (i, j), ak $(i, j) \notin E$ tak $w_{ij} = \infty$, ďalej pre každé i $w_{ii} = 0$.

Výstupom algoritmu bude matica $C^{(n)} = (c_{ij}^{(n)})$, kde $c_{ij}^{(n)}$ je cena najlacnejšej cesty z vrcholu i do vrcholu j.

Algoritmus 9 (Floyd–Warshall)

begin
  $n \leftarrow |V|$
  $C^{(0)} \leftarrow W$
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
        $c_{ij}^{(k)} \leftarrow MIN(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)})$
  return $C^{(n)}$
end

Veta 2.7 *Algoritmus 9 vypočíta cenu najlacnejšej cesty z každého vrcholu i do každého vrcholu j v najhoršom prípade v čase $O(|V|^3)$.*

Dôkaz:

Časová zložitosť: Zrejmá.

Korektnosť algoritmu: Indukciou vzhľadom na k možno dokázať, že $c_{ij}^{(k)}$ je cena najlacnejšej cesty z vrcholu i do vrcholu j, ktorej všetky vnútorné vrcholy sú z množiny $\{1, 2, \ldots, k\}$ (vnútorné vrcholy cesty sú všetky vrcholy cesty okrem jej prvého a posledného vrcholu). □

8. **Najlacnejšia kostra grafu**. (Algoritmus Union-FindSet. Kruskalov algoritmus.)

2.1 Najlacnejšia kostra grafu

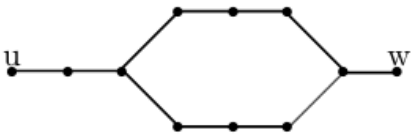
Definícia 2.1 *Nech $G = (V, E)$ je neorientovaný súvislý graf s ohodnotenými hranami, (tj. pre G je daná cenová funkcia $h : E \to R$; R je množina reálnych čísel).*

1. *Kostra grafu G je ľubovoľný neorientovaný strom (V, T), $T \subseteq E$, spájajúci všetky vrcholy z V (t.j. ľubovoľné dva vrcholy z V sú spojené cestou v strome (V, T)).*
2. *Cena kostry je $\sum_{e \in T} h(e)$.*
3. *Kostrový les pre graf G je ľubovoľná množina stromov $\{(V_1, T_1), \ldots, (V_k, T_k)\}$, $k \geq 1$ taká, že $V = \cup_{i=1}^{k} V_i$, $V_i \cap V_j = \emptyset$ pre $i \neq j$, v každom strome (V_i, T_i) sú spojené všetky vrcholy z V_i a $T_i \subseteq E \cap (V_i \times V_i)$ pre každé i (každý strom (V_i, T_i) je kostra grafu $(V_i, E \cap (V_i \times V_i))$).*

Lema 2.2 *Nech $G = (V, E)$ je súvislý neorientovaný graf a nech $S = (V, T)$ je kostra grafu G. Potom:*

1. *Pre všetky $u, w \in V$ je cesta medzi u a w v S jediná.*
2. *Po pridaní ľubovoľnej hrany z $E - T$ do S vznikne jediná kružnica.*

Dôkaz: Časť 1 vyplýva z toho, že keby boli v S dve cesty medzi u a w, potom by bola v S kružnica.



Obrázok 7: Ak sú medzi u a w dve cesty, existuje v grafe kružnica

Časť 2. Keďže S je kostra (t.j strom spájajúci všetky vrcholy), existuje medzi ľubovoľnými vrcholmi jediná cesta (pozri časť 1) a preto po pridaní ľubovoľnej hrany z $E - T$ musí vzniknúť jediná kružnica. □

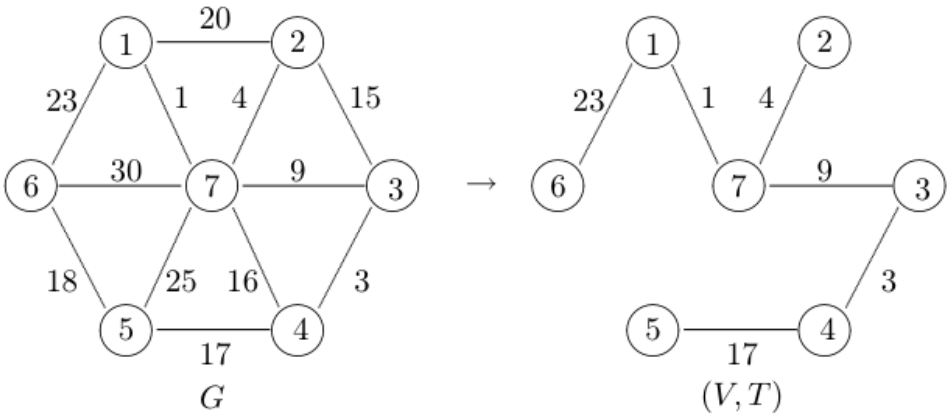
Nasledujúci greedy algoritmus nájde najlacnejšiu kostru grafu. Vstupom je neorientovaný súvislý graf $G = (V, E)$ s ohodnotenými hranami. Vrcholy sú reprezentované prirodzenými číslami $1, 2, \ldots, |V|$, hrany spolu s cenami sú dané v zozname.

Algoritmus 7 (Kruskal)

begin

$T \leftarrow \emptyset$ *(1)*
pre každý vrchol $v \in V$ *vytvor množinu* $\{v\}$ *(2)*
utrieď hrany v E *podľa cien v neklesajúcom poradí* *(3)*
pre každú hranu v $(u, w) \in E$ *v poradí podľa neklesajúcich cien:* *(4)*
begin
  if *FIND-SET*$(u) \neq$ *FIND-SET*(w) **then begin** *(5)*
    $T \leftarrow T \cup \{(u, w)\}$ *(6)*
    UNION(FIND-SET(u)*, FIND-SET*(w)*)* *(7)*
  end *(8)*
end
return T

end



Obrázok 8: Príklad Kruskalovho algoritmu

Veta 2.4 *Algoritmus 7 nájde najlacnejšiu kostru grafu* $G = (V, E)$ *s časovou zložitosťou v najhoršom prípade* $O(|E| \log |E|)$.

9. **Násobenie matíc**. (Naivný algoritmus. Strassenov algoritmus. Efektívne umocňovanie matice. Tranzitívny uzáver grafu pomocou umocňovania matíc.)

3 Algoritmy na maticiach

V tejto kapitole sa budeme zaoberať výpočtovou zložitosťou násobenia matíc. Uvidíme, že časová zložitosť $O(n^3)$ priamočiareho algoritmu na násobenie matíc nie je optimálna. Ukážeme si algoritmus s časovou zložitosťou $O(n^{2.81})$. Najlepší algoritmus známy do roku 1990 má časovú zložitosť $O(n^{2.376})$. Množstvo iných problémov je možné zredukovať na násobenie matíc. V týchto prípadoch použitím lepšieho násobenia matíc dostaneme efektívne riešenia, ktoré majú nižšiu časovú zložitosť ako by sa na prvý pohľad dalo predpokladať (napríklad riešenie sústav lineárnych rovníc).

3.1 Strassenov algoritmus násobenia matíc

Zlepšenie priamočiareho algoritmu násobenia matíc spočíva v použití techniky "rozdeľuj a panuj". Priamočiare použitie tejto techniky však neprinesie očakávaný výsledok.

Pokúsme sa vynásobiť matice rozmeru $2n \times 2n$ pomocou operácií násobenia a sčítania matíc rozmeru $n \times n$ (každú maticu rozdelíme na štyri podmatice rozmeru $n \times n$):

$$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Takto sme pôvodný problém násobenia dvoch matíc rozmerov $2n \times 2n$ previedli na 8 násobení a 4 sčítania matíc rozmerov $n \times n$. Nech $T(n)$ je počet operácií potrebných na násobenie násobenie dvoch matíc rozmerov $n \times n$. Pri použití metódy rozdeľuj a panuj dostávame rekurentý vzťah

$$T(n) = 8T(n/2) + \Theta(n^2).$$

Riešením tohto rekurentného vzťahu dostávame $T(n) = \Theta(n^3)$. Použitím tejto metódy sme teda nedosiahli žiadne zlepšenie.

Kľúčom k riešeniu je nájdenie takého spôsobu násobenia matíc rozmerov 2×2, pri ktorom sa použije menší počet násobení.

Lema 3.1 *Súčin dvoch matíc typu 2×2 možno vypočítať pomocou 7 násobení a 18 sčítaní/odčítaní.*

Dôkaz: Pre súčin matíc

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

platí

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7, \end{aligned}$$

kde

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \end{aligned}$$

$$\begin{aligned} m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 &= (a_{11} + a_{12})b_{22} \\ m_5 &= a_{11}(b_{12} - b_{22}) \\ m_6 &= a_{22}(b_{21} - b_{11}) \\ m_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

□

Veta 3.2 (Strassen) *Na vynásobenie dvoch matíc typu $n \times n$ stačí $O(n^{\log_2 7})$ aritmetických operácií.*

Dôkaz: Nech A a B sú dve matice typu $n \times n$, nech $n = 2^k$. Rozdeľme každú z matíc A a B na štyri podmatice typu $\frac{n}{2} \times \frac{n}{2}$. Teda

$$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Podľa lemy 3.1 možno všetky podmatice C_{ij} vypočítať pomocou 7 súčinov a 18 súčtov matíc typu $\frac{n}{2} \times \frac{n}{2}$. Rekurzívnym aplikovaním tohoto algoritmu možno vypočítať súčin dvoch matíc typu $n \times n$ s použitím $T(n)$ jednoduchých aritmetických operácií, kde

$$T(n) \leq 7T(\frac{n}{2}) + 18(\frac{n}{2})^2,$$

pre $n \geq 2$. Preto $T(n) = O(7^{\log_2 n}) = O(n^{log_2 7})$.

Ak n nie je mocnina čísla 2, potom rozšírime obe matice A a B tak, aby boli typu $2^k \times 2^k$, kde $n < 2^k \leq 2n$. Celkový počet operácií postačujúci na vykonanie algoritmu popísaného vyššie na takto rozšírených maticiach bude $O((2^k)^{\log_2 7}) = O((2n)^{\log_2 7}) = O(n^{\log_2 7})$. □

10. **Dynamické programovanie** (Konkrétne príklady použitia. Charakterizácia problémov riešiteľných dynamickým programovaním. Porovnanie iteratívneho prístupu a rekurzie s memoizáciou.)

4.5 Dynamické programovanie

Dynamické programovanie, podobne ako metóda "Rozdeľuj a panuj", zostrojí riešenie problému pomocou riešení podproblémov. Na rozdiel od "Rozdeľuj a panuj"metóda dynamického programovania rieši problém "zdola-nahor"(bottom-up) a to tak, že postupuje od menších podproblémov k väčším. Medzivýsledky zapisujeme do tabuľky, čím možno zabezpečiť, že každý podproblém je riešený práve raz. V niektorých aplikáciách iných metód ("Rozdeľuj a panujälebo backtracking) môže totiž dochádzať k viacnásobnému riešeniu niektorých podproblémov, čo zapríčiní spravidla horšiu časovú zložitosť.

Poznámka: Dynamické programovanie nemusí viesť k efektívnemu algoritmu, ak nepotrebujeme pre výpočet celkového problému poznať riešenia všetkých podproblémov.

Príklad: Floyd-Warshallow algoritmus (pozri stranu 22), riešenie problému násobenia reťazca matíc, 0-1 knapsack problému (viď. nižšie).

4.5.1 Problém násobenia reťazca matíc

Príkladom problému, ktorý sa dá efektívne riešiť použitím metódy dynamického programovania je problém násobenia reťazca matíc.

Daných je n matíc $M_1, \ldots, M_n$, kde M_i je matica typu $r_{i-1} \times r_i$. Úlohou je vypočítať súčin týchto matíc s minimálnym celkovým počtom skalárnych násobení, pričom predpokladáme, že súčin matice typu $p \times q$ s maticou $q \times r$ potrebuje pqr skalárnych násobení (tj. matice násobíme klasickým spôsobom)[11].

Počet rôznych spôsobov, ako vypočítať súčin n matíc, tj. počet rôznych uzátvorkovaní, je $P(n)$, kde

$$P(n) = \begin{cases} 1 & \text{ak } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{inak} \end{cases}$$

Možno ukázať, že

$$P(n) = \frac{1}{n}\binom{2n-2}{n-1} = \Omega(4^n/n^{3/2})$$

Preskúmaním všetkách možností výpočtu súčinu n matíc vedie k neefektívnemu algoritmu s exponenciálnou zložitosťou. Pomocou dynamického programovania zostrojíme algoritmus so zložitosťou $O(n^3)$.

Nech sú dané čísla $r_0, \ldots, r_n$, kde $r_{i-1} \times r_i$ je typ matice M_i. Zavedieme pole $m[1 \ldots n, 1 \ldots n]$, kde $m[i, j]$ bude minimálny počet skalárnych násobení potrebných na výpočet súčinu matíc $M_i, \ldots, M_j$. V pomocnom poli $s[1 \ldots n, 1 \ldots n]$ budeme

ukladať číslo $s[i,j] = k$, ktoré určuje, ako treba pri optimálnom násobení matíc reťazec uzátvorkovať (tj. určuje takéto uzátvorkovanie: $(M_i \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_j)$, pričom reťazec $M_i, \ldots, M_k$ násobíme podľa hodnôt $m[i,k]$ a $s[i,k]$ a reťazec $M_{k+1}, \ldots, M_j$ podľa hodnôt $m[k+1,j]$ a $s[k+1,j]$).

Algoritmus 11

```
begin
    for i ← 1 to n do m[i,i] ← 0
    for l ← 1 to n − 1 do
        for i ← 1 to n − l do begin
            j ← i + l
            min ← i; minh ← m[i,i] + m[i+1,j] + r_{i−1} r_i r_j
            for k ← i + 1 to j − 1 do begin
                h ← m[i,k] + m[k+1,j] + r_{i−1} r_k r_j
                if h < minh then begin
                    minh ← h
                    min ← k
                end
            end
            m[i,j] ← minh
            s[i,j] ← min
        end
end
```

Hodnoty tabuľky $m[1 \ldots n, 1 \ldots n]$ je možné vypočítať aj použitím metódy rozdeľuj a panuj. Procedúra RP(i, j) vypočíta hodnoty tabuľky $m[k,l]$ pre všetky $i \leq k \leq l \leq j$, tj. pre všetky podreťazce reťazca matíc $M_i M_{i+1} \ldots M_j$. Celú tabuľku teda vypočítame pomocou RP(i,j).

Algoritmus 12

```
procedure RP(i,j)
begin
    if i = j then m[i,j] ← 0
    else m[i,j] ← min_{i≤k<j}{RP(i,k) + RP(k+1,j) + r_{i−1} r_k r_j}
    return m[i,j]
end
```

Lema 4.3 *Nech $T(m)$ je čas výpočtu procedúry RP(i,j) pre $m = j - i + 1$. Platí $T(m) \geq 2^{m-1}$.*

Dôkaz: Matematickou indukciou. Pre $m = 1$ platí $T(1) \geq 1$, ďalej

$$\begin{aligned} T(m) &= T(j-i+1) \geq \sum_{k=i}^{j-1} (T(k-i+1) + T(j-k)) = |l := k-i+1| = \\ &= \sum_{l=1}^{m-1} (T(l) + T(m-l)) = \\ &= 2 \sum_{l=1}^{m-1} T(l) \geq 2 \sum_{l=1}^{m-1} 2^{l-1} = 2(2^{m-1} - 1) \geq 2^{m-1} \end{aligned}$$

Z neefektívnej procedúry RP ($\Omega(2^n)$) možno ľahko vyrobiť efektívnu procedúru tak, že si budeme pamätať, či už boli jednotlivé podproblémy vyriešené alebo nie (aby nedochádzalo k ich viacnásobnému riešeniu).

Algoritmus 13

procedure *RPM(i,j)*
begin
  if *procedúra RPM ešte nebola volaná s parametrami* i *a* j **then begin**
    if $i = j$ **then** $m[i,j] \leftarrow 0$
    else $m[i,j] \leftarrow \min_{i \leq k < j}\{RPM(i,k) + RPM(k+1,j) + r_{i-1}r_k r_j\}$
  end
  return $m[i,j]$
end

Lema 4.4 *Procedúra RPM(1,n) vypočíta hodnoty tabuľky* $m[1 \ldots n, 1 \ldots n]$ *v čase* $O(n^3)$.

4.5.2 0-1 knapsack problém

Daných n objektov s váhami $w_1, w_2, \ldots, w_n$ (kde w_i sú prirodzené čísla), cenami $v_1, v_2, \ldots, v_n$ a ďalej je dané prirodzené číslo w. Úlohou je vybrať niektoré z objektov tak, aby celková cena vybraných objektov bola najväčšia a zároveň aby ich celková váha neprekročila hodnotu w, tj. nájsť číslo

$$\max_{S \subseteq \{1,2,\ldots,n\}} \{\sum_{i \in S} v_i | \sum_{i \in S} w_i \leq w\}$$

.

Nech $V(w,j) = \max_{S \subseteq \{1,2,\ldots,j\}}\{\sum_{i \in S} v_i | \sum_{i \in S} w_i \leq w\}$, pre $j = 1, 2, \ldots, n$. Ak optimálny výber objektov pre $V(w, j+1)$ obsahoval objekt s indexom $j+1$, po odobratí tohto objektu dostaneme výber s celkovou váhou o w_{j+1} menšou, pričom tento výber je zrejme optimálny pre $V(w - w_{j+1}, j)$. Ak optimálny výber pre

$V(w, j+1)$ nebsahuje objekt s indexom $j+1$, potom $V(w, j+1) = V(w,j)$. Teda pre každé j platí

$$V(w, j+1) = \max\{V(w,j), v_{j+1} + V(w - w_{j+1}, j)\}.$$

Dynamickým programovaním možno riešiť 0-1 knapsack problém v čase $O(nW)$ a to postupným vypĺňaním tabuľky $V[0 \ldots W, 0 \ldots n]$ použijúc vzťah

$$V[w,j] = \begin{cases} \max\{V[w, j-1], v_j + V[w - w_j, j-1]\} & \text{ak } 0 < j \leq n \\ 0 & \text{ak } j = 0 \end{cases}$$

Výslednou hodnotou je číslo $V[w,n]$.

11. **Ďalšie princípy tvorby efektívnych algoritmov**. (Rozdeľuj a panuj, pažravé algoritmy, princíp vyváženosti, voľba vhodnej dátovej štruktúry. Konkrétne príklady použitia.)

4 Metódy tvorby efektívnych algoritmov

Táto kapitola sa bude zaoberať niektorými technikami, ktoré sa používajú pri tvorbe efektívnych algoritmov. Vo všeobecnosti neexistuje univerzálna metóda konštrukcie efektívnych algoritmov. Napriek tomu však často možno použiť niektorú z nasledujúcich metód:

Princíp neustáleho zlepšovania. Tento, kto navrhuje algoritmus riešiaci danú úlohu, mal by pokračovať v skúmaní problému z rôznych pohľadov, až kým si nie je istý, že získal najlepší algoritmus pre jeho potreby.

Voľba vhodnej štruktúry údajov. Spôsob organizácie dát pri výpočte algoritmu často výrazne vplýva na jeho efektívnosť. Preto voľbou vhodnej reprezentácie získavame obvykle efektívnejší algoritmus.

Princíp vyváženosti (Balancing). Navrhované algoritmy často používajú rekurzívne schémy výpočtu alebo rekurzívne dátové štruktúry. V týchto prípadoch sa ukazuje, že je výhodné z hľadiska efektívnosti, aby jednotlivé podštruktúry (prípadne podvýpočty) mali približne rovnakú veľkosť.

Metóda "Rozdeľuj a panuj"(Divide and conquer). Rozdelíme úlohu na niekoľko menších podúloh, ktoré riešime samostatne. Potom z ich výsledkov vypočítame celkový výsledok.

Dynamické programovanie. Táto metóda je podobná ako "rozdeľuj a panuj– riešenie problému takisto dostávame z riešení podproblémov. V dynamickom programovaní použijeme ten istý princíp vo väčšom rozsahu: ak nevieme presne určiť, ktoré menšie problémy riešiť, jednoducho ich vyriešime všetky a uložíme si výsledky, aby mohli byť použité pri výpočte väčších problémov.

Greedy algoritmy. Pri hľadaní otimálneho riešenia daného problému si pri výpočte zvolíme vždy lokálne najlepšiu možnosť. Tento prístup vedie k rýchlemu algoritmu, musíme však dávať pozor na to, aby riešenie bolo korektné.

4.1 Princíp neustáleho zlepšovania

Pri skúmaní určitého problému je zvykom postupovať z dvoch strán: na jednej strane skúmame problém z rôznych pohľadov a snažíme sa nájsť stále efektívnejší algoritmus a takto nájdený algoritmus nám poskytuje horný odhad zložitosti problému. Na druhej strane sa snažíme nájsť čo najlepší dolný odhad. Pokiaľ sa tieto dva odhady nestretnú, je obvykle možné zlepšiť buď dolný odhad, alebo nájsť efektívnejší algoritmus.

Z tohto hľadiska o každom algoritme, o ktorom sa zatiaľ nepodarilo dokázať, že ho nemožno zlepšiť, treba predpokladať, že sa zlepšiť dá.

Príklad: Algoritmus QUICKSORT má v priemernom prípade zložitosť $O(n \log n)$. Ukázali sme, že zložitosť triedenia v priemernom prípade je $\Omega(n \log n)$. V tomto prípade sme teda dosiahli zhodu dolného a horného odhadu.

Príklad: Násobenie matíc: klasická metóda $O(n^3)$, Strassenova metóda $O(n^{2.81})$, najlepší výsledok do roku 1990 $O(n^{2.376})$. Známy dolný odhad $\Omega(n^2)$.

Príklad: Násobenie n-bitových čísel: klasická metóda $O(n^2)$, metóda "Rozdeľuj a panuj"$O(n^{1.59})$, Shönhage–Strassenova metóda pomocou Fourierovej transformácie $O(n \log n \log \log n)$

4.2 Voľba vhodnej štruktúry údajov

Abstraktný dátový typ je abstrakcia nad dátovými štruktúrami, kde neuvažujeme skutočné uloženie dát, ale iba to, aké operácie sa budú na štruktúre vykonávať.

Príklad: Slovník je abstraktný dátový typ, u ktorého považujeme operácie MEMBER, INSERT, DELETE.

Pri tvorbe algoritmu musíme rozhodnúť, akými dátovými štruktúrami budeme realizovať jednotlivé abstraktné dátové typy potrebné v algoritme. Pri tom musíme brať ohľad na to, aby najpoužívanejšie operácie boli realizované čo najefektívnejšie.

Príklad: Realizácia slovníka:

Implementácia	MEMBER	INSERT	DELETE
Pole	$O(n)$	$O(1)$	$O(n)$
Utriedené pole	$O(\log n)$	$O(n)$	$O(n)$
2-3 stromy	$O(\log n)$	$O(\log n)$	$O(\log n)$

Príklad: Prioritná fronta je abstraktný dátový typ, od ktorého požadujeme operácie INSERT, MIN, DELETE_MIN.

Implementácia	INSERT	MIN	DELETE_MIN
Pole	$O(1)$	$O(n)$	$O(n)$
Utriedené pole	$O(n)$	$O(1)$	$O(1)$
Halda	$O(\log n)$	$O(1)$	$O(\log n)$
2-3 stromy	$O(\log n)$	$O(\log n)$	$O(\log n)$

4.3 Princíp vyváženosti

Pri návrhu algoritmov sa často stretneme s prípadmi, keď sa výpočet rozdeľuje na niekoľko podúloh, alebo sa nejaká dátová štruktúra rozdeľuje na menšie podštruktúry. Efektívnosť takýchto algoritmov možno často zvýšiť tým, že sa snažíme, aby medzi jednotlivými podobjektami (či už podvýpočtami alebo podštruktúrami) bola istá vyváženosť.

Príklad: Algoritmus QUICKSORT, ktorý vyberá pivotný prvok náhodne, má v priemernom prípade časovú zložitosť $O(n \log n)$. V najhoršom prípade (keď za pivotný prvok vyberieme vždy minimum) však tento algoritmu má časovú zložitosť $O(n^2)$. Ak však za pivotný prvok volíme napríklad medián (viď. strana 4), čím zaistíme, že pole sa rozdelí na dve rovnaké časti (s rozdielom nanajvýš jeden prvok), dostávame aj v najhoršom prípade časovú zložitosť $O(n \log n)$.

Príklad: Výška binárneho prehľadávacieho stromu je v priemernom prípade $\log n$ a vyhľadávanie v takomto strome má teda časovú zložitosť $O(\log n)$. V najhoršom prípade však výška takéhoto stromu môžu byť až n a teda časová zložitosť vyhľadanie prvku v najhoršom prípade je $O(n)$. Ak však zabezpečíme, aby podstromy pod ľubovoľným prvkom mali približne rovnakú výšku (tzv. vyvážené stromy), získame vždy strom s výškou približne $\log n$ a teda vyhľadávanie v takomto strome má časovú zložitosť $O(\log n)$ aj v najhoršom prípade.

Príklad: 2-3 stromy (viď. strana 9), AVL stromy, červeno-čierne (RB) stromy, stromy pre UNION/FIND-SET problém (viď. strana 14), binárne prehľadávacie stromy, algoritmus SELECT pre hľadanie k-teho najmenšieho prvku (viď. strana 4).

4.4 Metóda "Rozdeľuj a panuj"

Metóda je založená na tom, že problém rozdelíme na niekoľko podproblémov, podobných ako pôvodný problém, ale menšieho rozsahu. Potom rekurzívne vyriešime tieto podproblémy a nakoniec zostrojíme riešenie celého problému pomocou riešení podproblémov.

Ak sa podarí rozdeliť problém rozsahu n na d problémov rozsahu n/c, pričom celkový čas potrebný na rozklad na podproblémy a konštrukciu riešenia problému pomocou riešení podproblémov nepresiahne čas bn, potom možno časovú zložitosť odhadnúť pomocou nasledujúcej vety.

Veta 4.1 *Nech b, c, d sú nezáporné konštanty. Riešenie rekurentnej rovnice*

$$T(n) = \begin{cases} bn & \text{pre } n = 1, \\ dT(n/c) + bn & \text{pre } n > 1, \end{cases}$$

je pre $n = c^k$

$$T(n) = \begin{cases} O(n) & \text{ak } d < c, \\ O(n \log n) & \text{ak } d = c, \\ O(n^{\log_c d}) & \text{inak.} \end{cases}$$

Dôkaz: Ak $n = c^k$, potom platí

$$\begin{aligned} T(n) &\leq bn + dT(n/c) \\ &\leq bn + bn\frac{d}{c} + bn\frac{d^2}{c^2} + \cdots + bn\left(\frac{d}{c}\right)^{\log_c n} \\ &= bn \sum_{i=0}^{\log_c n} r^i, \end{aligned}$$

kde $r = d/c$. Ak $r < 1$, potom rad $\sum_{i=0}^{\log_c n} r^i$ konverguje, čiže

$$T(n) \leq bn \sum_{i=0}^{\log_c n} r^i \leq \sum_{i=0}^{\infty} r^i \leq hn,$$

pre nejakú vhodnú kladnú konštantu h. Ak $r = 1$, potom

$$T(n) \leq bn(\log_c n + 1).$$

Ak $r > 1$, potom

$$\begin{aligned} T(n) &\leq bn \sum_{i=0}^{\log_c n} r^i = bn\frac{r^{1+\log_c n} - 1}{r - 1} = bc^{\log_c n}\frac{\left(\frac{d}{c}\right)^{1+\log_c n} - 1}{\frac{d}{c} - 1} = \\ &= O(d^{\log_c n}) = O(n^{\log_c d}). \end{aligned}$$

□

Príklad: Sú dané dve n-bitové čísla x a y. Pokúsme sa nájsť efektívny algoritmus, ktorý tieto dve čísla vynásobí. Nech $x = a2^{n/2} + b$ a $y = c2^{n/2} + d$, kde a, b, c, d sú $n/2$ bitové čísla. Potom súčin $z = xy$ možno vypočítať takto:

$u \leftarrow (a + b)(c + d)$
$v \leftarrow ac$
$w \leftarrow bd$
$z \leftarrow v2^n + (u - v - w)2^{n/2} + w$

Čitateľ ľahko ukáže na základe predchádzajúcej úvahy, že dve n-bitové čísla možno vynásobiť v čase $T(n)$, kde $T(n) = 3T(n/2) + tn$ pre nejaké $t \geq 0$ a každé n, ktoré je mocninou dvojky. Podľa vety 4.1 teda $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

Príklad: Metódu Rozdeľuj a panuj využívame aj pri konštrukcii týchto algoritmov: Strassenovo násobenie matíc (pozri strana 26), hľadanie k-teho najmenšieho prvku (pozri strana 4), quicksort, binárne vyhľadávanie.

Poznámka: Bez dôkazu uveďme ešte jedno tvrdenie, ktoré je o niečo všeobecnejšie, ako tvrdenie 4.1.

Veta 4.2 *Nech $a \geq 1$, $b > 1$ sú konštanty, nech f je funkcia a nech $T(n)$ je definovaná rekurentne ako nezáporná funkcia*

$$T(n) = aT(n/b) + f(n).$$

Potom $T(n)$ môže byť asymptoticky ohraničená takto:

- *ak $f(n) = O(n^{\log_b a - \varepsilon})$ pre nejaké $\varepsilon > 0$, potom $T(n) = \Theta(n^{\log_b a})$,*
- *ak $f(n) = \Theta(n^{\log_b a})$, potom $T(n) = \Theta(n^{\log_b a} \log n)$,*
- *ak $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pre nejaké $\varepsilon > 0$ a ak $af(n/b) \leq cf(n)$ pre nejaké $0 < c < 1$ a pre všetky dostačne veľké n, potom $T(n) = \Theta(f(n))$.*

4.6 Greedy algoritmy

Greedy algoritmy sa používajú na riešenie optimalizačných problémov. Globálne optimálne riešenia hľadajú (alebo vytvárajú) pomocou postupnosti lokálne optimálnych rozhodnutí, ktoré už ďalej nie sú revidované. Obvykle sú to iteratívne algoritmy, v ktorých lokálne optimálne rozhodnutia redukujú problémy na podproblémy menšieho rozsahu, v dôsledku čoho sú mnohé z týchto algoritmov rýchle.

Príklad: Dijkstrov algoritmus (pozri stranu 20), Kruskalov algoritmus (pozri stranu 18)

Príklad: Daných je n objektov s váhami $w_1, w_2, \ldots, w_n$ (kde w_i sú prirodzené čísla), cenami $v_1, v_2, \ldots, v_n$ a ďalej je dané prirodzené číslo w. Úlohou je vybrať časti niektorých objektov tak, aby celková cena vybraných častí bola najväčšia a zároveň aby ich celková váha neprekročila hodnotu w, tj. nájsť číslo

$$\max_{\alpha_1, \alpha_2, \ldots, \alpha_n \in <0,1>} \{ \sum_{i=1}^{n} \alpha_i v_i | \sum_{i=1}^{n} \alpha_i w_i \leq w \}$$