Pseudopolynomi

Den här texten är ett komplement till läromaterialet i ADK som förklarar i större detalj vad pseudopolynomi innebär och hur man undviker att konstruera pseudopolynomiska algoritmer på mästarprovet.

Se även presentationsbilder från ADK19 om pseudopolynomi.

Bit- och enhetskostnad

Låt oss först repetera begreppen *enhetskostnad* och *bitkostnad*. Dessa är två kostnadsmått som vi använder för att mäta tidskomplexiteten för en algoritm.

Med enhetskostnad antar vi att vissa grundläggande operationer på heltal tar konstant tid. Dessa operationer är oftast följande:

- Addition
- Subtraktion
- Multiplikation
- Division
- Tilldelning
- Jämförelse

Enhetskostnad är ett bra kostnadsmått då vi har att göra med "vanliga" tal som är begränsade i storlek, t.ex. int, long, float, double, o.s.v., då moderna processorer gör operationer på dessa på konstant tid. Använder man enhetskostnad som kostnadsmått måste man därför argumentera för varför talen i ens algoritm är begränsade eller hur man kan begränsa dem.

Med bitkostnad antar vi däremot inte detta, utan tar hänsyn till hur mycket tid som går åt beroende på hur stora talen är. Om talen har *n* bitar är operationernas komplexitet följande:[1]

Operation	Tidskomplexitet
Tilldelning / jämförelse	$O(n)$
Addition / subtraktion	$O(n)$
Multiplikation / division[2]	$O(n^2)$

Bitkostnad är ett bättre kostnadsmått då talen kan bli obegränsat stora beroende på indata, eftersom då tar operationerna längre tid beroende på talens storlek.

Pseudopolynomisk algoritm

Något som är lätt att förbise är vad komplexiteten för en algoritm är en funktion av, d.v.s. vad n är. Då vi räknar ut komplexiteten för en algoritm gör vi det normalt som en funktion av längden på indata. n är alltså antalet bitar som krävs för att representera indatan. Indata kan se ut på olika sätt för olika problem, men låt oss betänka fallet att indata enbart är ett heltal

[1] Den intresserade läsaren kan läsa mer om algoritmerna på Wikipediasidan.
[2] Den triviala algoritmen tar O(n^2) tid, men det finns även snabbare algoritmer.

k. Notera att med n bitar kan vi representera tal mellan 0 och $2^n - 1$, så i värsta fall har vi $k = 2^n - 1 \in O(2^n)$. Om vi nu bygger en algoritm som är polynomisk i k, till exempel $T(k) = O(k^3)$, så kommer den att vara exponentiell i n, eftersom $k^3 \in O((2^n)^3)$. En algoritm som är polynomisk i talen som ingår i indata sägs vara pseudopolynomisk. En pseudopolynomisk algoritm är en exponentiell algoritm när vi beräknar tidskomplexitet på vanligt sätt (som funktion av indatas längd) och är alltså inte effektiv för stora indata.

Exempel - Lapptäcke

Läs igenom uppgift 2 på Mästarprov 2 från ADK18. Vi ska titta på verifieringsalgoritmen, titta på hur den implementerats i studentlösningarna på B- och C-nivå för mästarprovet. Studentlösningen på B-nivå har en polynomisk verifieringsalgoritm, medan den på C-nivå är pseudopolynomisk och därför ej godkänd. Vi ska nu ge oss på att göra en egen verifieringsalgoritm.

Indata:

- En lista rektangulära lappar med bredd och höjd: $L = [(b_1, h_1), (b_2, h_2),\ldots]$
- En bredd b och en höjd h på det önskade lapptäcket
- En lösningslista P som anger varje lapps placering (x, y) samt ifall den är roterad (r): $P = [(x_1, y_1, r_1), (x_2, y_2, r_2),\ldots]$

Utdata: En boolean som säger ifall lösningen är korrekt eller ej.

Verifieringsalgoritmen är väldigt enkel - vi skapar lapptäcket och fyller det med bitarna. Täcker vi över ett område som vi redan täckt så överlappar lapparna och lösningen är felaktig. Finns det någon ruta på slutet som inte är täckt är lösningen felaktig.

```
VerifyPseudo(L, b, h, p):
    M ← (b × h)-booleanmatris
    for i ← 0 to len(L):
        if p[i].r:
            swap(L[i].x, L[i].y)
        for x ← p[i].x to p[i].x + L[i].b:
            for y ← p[i].y to p[i].y + L[i].h:
                if M[x][y]:
                    return FALSE
                M[x][y] ← TRUE
    for x ← 0 to b:
        for y ← 0 to h:
            if not M[x][y]:
                return FALSE
    return TRUE
```

Låt oss analysera tidskomplexiteten. Först skapar vi lapptäcket M med dimensioner $b \times h$, vilket tar $\Theta(b \cdot h)$ tid (theta eftersom vi alltid skapar lapptäcket). Att allokera minne tar alltid linjär tid i mängden minne, vilket ofta glöms bort. Därefter går vi igenom listan L och fyller i varje lapp i lapptäcket. Notera att vi fyller i varje ruta i lapptäcket högst en gång,

eftersom vi annars skulle se att vi redan fyllt i rutan och returnera. Alltså tar även denna slinga $O(b \cdot h)$ tid. Slutligen går vi igenom hela lapptäcket en gång till vilket även det tar $O(b \cdot h)$ tid. Alltså tar hela verifieringsalgoritmen $\Theta(b \cdot h)$ tid. Detta är dock inte acceptabelt, eftersom b och h är arbiträra tal, och som vi kommit fram till tidigare kan de vara exponentiella i längden på indata n. Vi måste därför konstruera en verifieringsalgoritm som löser problemet utan att hålla reda på lapptäcket explicit (eftersom det är för stort).

```
VerifyPoly(L, b, h, p):
     area ← 0
     for i ← 0 to len(L):
          area ← area + L.b * L.h
          for j ← i+1 to len(L):
               if intersects(i,j):
                    return FALSE
     return area = b * h
```

`intersects` är en subrutin som avgör om rektanglarna på index i, j skär varandra. Denna kan implementeras med enkla aritmetiska operationer och jämförelser som tar konstant tid med enhetskostnad, men implementationen utelämnas här.

Låt oss analysera tidskomplexiteten. Vi går igenom alla par av lappar, vilket med k lappar tar $O(k^2)$ tid. I övrigt utför vi enbart enkla aritmetiska operationer och jämförelser, vilket tar konstant tid med enhetskostnad. Antalet lappar måste vara proportionellt mot längden på indata, ty varje lapp tar ett antal bitar att representera i indata, eftersom vi måste lagra höjden och bredden för varje lapp. Alltså är algoritmen polynomisk sett till längden på indata.

Exempel - Täckning av en kvadrat med L

Läs igenom uppgift 1 på Mästarprov 1 från ADK19 samt lösningsförslaget. I uppgiften har vi en kvadrat med sidan m och en algoritm som tar tiden $O(m^2)$ för att täcka kvadraten med L-formade bitar. Notera att den enda indatan i problemet är talet m, som tar $n = log(m)$ bitar att representera. Algoritmen ska returnera en $(m \times m)$-matris, som alltså tar $2^n \cdot 2^n = 2^{2n} = 4^n$ bitar att representera. Bara att skapa en sådan matris tar alltså exponentiell tid (i n) - vi vet därför med säkerhet att problemet inte går att lösa på polynomisk tid. För att så skulle gå att göra måste vi formulera om problemet, t.ex. genom att istället för att konstruera matrisen istället skapa en funktion som svarar på vilken L-bit en given position hör till.