Informationsteknologi
Tom Smedsaas 18 februari 2005

Quicksort

Detta papper beskriver kortfattat quicksort, gör en genomsnittsanalys, kommenterar en del och presenterar avslutningsvis ett C-program. Analysen bygger på resultat som erhölls i vårt papper om analys av binära sökträd.

Principen för quicksort

Grundidén i quicksort är att man delar upp ursprungsmängden i två mängder – den ena innehållande alla små tal och den andra alla stora. Dessa två mängder sorteras sedan var för sig varefter de sammanläggs till en resultatmängd. Om man hela tiden kan dela mängderna i två lika stora delmängder blir analysen identisk med analysen för mergesort dvs man erhåller en algoritm som tar $O(n \log n)$ tid. För att kunna dela mängden i två lika stora delar krävs dock att man känner till det till storleken mittersta elementet (*medianen*) vilket sällan är fallet. Lösningen är att inte bekymra sig utan välja ett element på måfå och använda detta som delningselement! Det betyder att man kan ha tur eller otur men, precis som i fallet med binära sökträd, så har man slumpen på sin sida och det blir bra i genomsnitt.

Grundideén är realiserbar på alla möjliga datastrukturer men det är just på arrayer som quicksort blir bäst.

Quicksort i array

Grunden i quicksort är den så kallade *partitioneringen* (quicksort kallas ibland också *partition sort*). Partitioneringen skall ordna om elementen i arrayen så att "små" element kommer på ena sidan och "stora" på andra. Problemet är att man från början inte vet vad som menas med "små" respektive "stora" element. Man gör då så att man väljer något av elementen och låter det utgöra gräns mellan "små" och "stora". Det valda elementet kallas *pivotelement*. Man går igenom alla element och ordnar om de så arrayen får följande uppdelning.

1 … k	$k+1$ … n
$\leq p$	$\geq p$

Förutom att ordna om elementen returnerar partitioneringen indexet k dvs gränsen mellan små och stora. Man har tur om k blir nära $n/2$ och otur om k blir nära 1 eller n. Därefter kan quicksort rekursivt appliceras på de små respektive de stora talen.

Nedan visas en enkel partitioneringsalgoritm. Den använder elementet på första plats ("längst till vänster") som pivotelement. I iterationen letas från höger efter små tal och från vänster efter stora. När två sådana påträffats (som båda alltså ligger i fel del) byter man plats på dessa och fortsätter. (Att tal som är lika med pivotelementet alltid betraktas som liggande på fel plats förenklar algoritmen – pivotelementet fungerar som *vaktpost* (eng. *sentinel*) i `repeat`-satserna, dvs man behöver inte kolla att i och j håller sig inom sina gränser.)

Partition(A, *left*, *right*)
    $p \leftarrow A[\mathit{left}]$
    $i \leftarrow \mathit{left} - 1$
    $j \leftarrow \mathit{right} + 1$
    while `True` **do**
        repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$
        repeat $i \leftarrow i + 1$ **until** $A[i] \leq p$
        if $i < j$
            then swap(A, i, j)
            else return j

Själva quicksort uttrycks sedan enkelt:

Quicksort(A, *left*, *right*)
    if $\mathit{left} < \mathit{right}$ **then**
        $k \leftarrow$ Partition(A, *left*, *right*)
        Quicksort(A, *left*, $k - 1$)
        Quicksort(A, $k + 1$, *right*)

Analys av quicksort

För att göra analysen aningen enklare modifierar vi partitioneringen något. Algoritmen skall se till att pivotelementet hamnar på plats k. Vi gör det genom att, när vi valt ett element på en slumpmässig plats som pivotelement p så kopierar vi det första elementet till den plats vi fann p på. Därmed finns en plats "ledig" bland de små elementen. Vi söker från höger bland de stora tills ett element mindre än p påträffas. Detta kopieras till den lediga platsen. Nu finns en ledig plats bland de stora. Då söker man efter ett stort element bland de små osv. Avslutningsvis läggs p på den lediga platsen k. Implementationen är enkel och lämnas som övning.

Efter partitioneringen skall således arrayen ha följande struktur:

$1 \quad\quad k-1$	k	$k+1 \quad\quad n$
$\leq p$	p	$\geq p$

Själva quicksortfunktionen anropas därefter med de $k-1$ första respektive de $n-k$ sista platserna.

I partitioneringen kommer varje element att jämföras med pivotelementet, dvs den kräver exakt $n-1$ jämförelser. Om pivotelementet p råkar bli nummer k i storleksordning ($1 \leq k \leq n$) så kan antalet jämförelser skrivas

$$T(n) = (n-1) + T(k-1) + T(n-k) \tag{1}$$

(partitionera, sortera de $k-1$ element som är mindre än eller lika med och de $n-k$ som är större än eller lika med p). Om alla element (dvs alla k) är lika sannolika så blir arbetet i genomsnitt

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{i=1}^{n}\left(T(i-1) + T(n-i) + n - 1\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(T(i-1) + T(n-i)\right) + \frac{1}{n}\sum_{i=0}^{n-1}(n-1) \\ &= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + (n-1). \end{aligned} \tag{2}$$

Genom att multiplicera båda sidor med n kan ovanstående skrivas

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n(n-1). \tag{3}$$

Standardknepet för att bli av med summan är att skriva formeln med $n-1$ i stället för n:

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)(n-2) \tag{4}$$

och sedan subtrahera denna från den föregående, vilket ger

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + n(n-1) - (n-1)(n-2) \\ &= 2T(n-1) + 2(n-1). \end{aligned} \tag{5}$$

vilket kan skrivas som

$$T(n) = \frac{n+1}{n}T(n-1) + 2 - 2/n < \frac{n+1}{n}T(n-1) + 2. \tag{6}$$

Vi har redan visat i samband med analysen av binära sökträd att lösningen till denna relation är $O(n \log n)$

Val av pivotelement

Första elementet är inte sämre som pivotelement än vilket annat element som helst under förutsättning att det inte finns någon systematik i indata. Om däremot mängden redan är sorterad blir denna strategi katastrofal – delningen blir den sämsta möjliga hela tiden. Det vanliga sättet att skydda sig mot detta är att slumpa fram ett index och använda elementet på motsvarande plats som pivotelement.

Man kan försöka minska risken för olyckliga val genom att ta ett stickprov på tre, fem eller sju element och använda medianen av dessa som pivotelement. I praktiken lönar sig sällan dessa strategier utom möjligen vid mycket stora sorteringsmängder och då endast vi de inledande partitioneringarna.

Små mängder

Som alla avancerade sorteringsmetoder är quicksort inte lönsam på små mängder. Var gränsen går är svår att säga men 20 kan vara ett riktvärde. Ett vanligt sätt är att låta quicksort lämna avsnitt med färre än t.ex. 20 element osorterade. Därefter appliceras vanlig instickssortering på *hela* arrayen. Instickssortering drar automatiskt fördel av att arrayen är grovsorterad. Praktiska försök visar emellertid att effekten av denna förbättring är ganska liten.

Arbetsutrymme

Även om inte arrayen behöver kopieras så kräver rekursionen stackutrymme som, med en klumpig implementation, i värsta fall kan uppgå till $\Theta(n)$ minnesceller. Om man själv hanterar stacken så kan man garantera att utrymmet som krävs är $O(\log n)$. I praktiken kan man dock förlita sig på att slumpen är "god" eftersom värsta fallet är mycket osannolikt.

Beräkning av fraktiler (median, kvartil...)

Partitioneringsalgoritmen kan användas för att beräkna fraktiler (median, kvartil etc.) i linjär tid. Man gör en "halv quicksort" dvs väljer ett pivotelement och gör partitioneringen. Därefter vet man om det sökta värdet ligger bland de små eller stora. Upprepa sedan på den del som innehåller det sökta värdet. Se boken för vidare detaljer.

En C-implementation

Kernighan och Ritchie har i sin klassiska C-bok (*The C Programming Language*) givit en enkel och snärtig implementation av quicksort:

```
void qsort( int v[], int left, int right ) {
  int i, last;
  void swap( int v[], int i, int j );

  if ( left >= right )
     return;                        /* Färre än två element */

      /* Partitionera */
  swap( v, left, (left+right)/2 );  /* Flytta pivot till första pos */
  last = left;
  for ( i = left+1; i <= right; i++ )
      if ( v[i] < v[left] )
         swap( v, ++last, i );
  swap( v, left, last );            /* Lägg tillbaka pivot */

      /* Sortera delarna för sig */
  qsort( v, left, last );
  qsort( v, last+1, right );
}
```

Med denna implementation så tar en sortering av en miljon heltal någon sekund på en normal PC (år 2003).

Slutligen bör man notera att quicksort finns som en standardfunktion i C under namnet `qsort` (inkludera `stdlib.h`).