Oskar Mothander
dit06omr@cs.umu.se

Interaktion & Design, ID06
18 mars 2009

Laboration 3

Komplexitetsanalys

Inlämning 2 augustikomplettering

Kurs: Datastrukturer & algoritmer VT07
Lärare: Lena Kallin Westin
Handledare: Lars Larsson
Tommy Löfstedt

Oskar Mothander
dit06omr@cs.umu.se

Innehållsförteckning

Del1 1
Algoritm 1 1
Algoritm 3 2
Del2 3
Swap 3
Beräkning av for-loopar 3
Naiv bubblesort 4
Optimerad bubblesort 6
In-place quickSort 7
RandomSort 8

Oskar Mothander
dit06omr@cs.umu.se

Interaktion & Design, ID06
18 mars 2009

Problemspecifikation

Labben är indelad i två delar. I första delen skulle det insamlas data genom testkörning av tre okända algoritmer och sedan göras grafer på dessa. Sedan skulle en gissning om Ordo göras.

I del två skulle Ordo, c och n_0 beräknas för två sorteringsfunktioner. Samt analys utav ”best-” och ”worst-case-scenario” för de två första sorteringsalgoritmerna. I de två sista algoritmerna räckte det med ett resonemang kring komplexiteten.

Labbspecifikation: http://www.cs.umu.se/kurser/TDBA36/VT07/ou/ou3/index.html[1]
Algoritmberäkning: http://www.cs.umu.se/kurser/TDBA36/forel_ny/OH/forel6_4.pdf

Analys

Del1

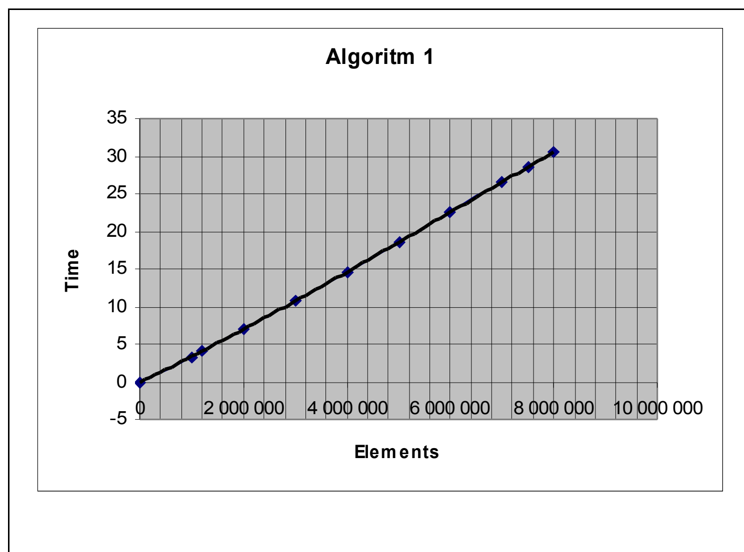
I den här delen skulle det göras experimentell komplexitetsanalys på några givna algoritmer. Genom testkörning fördes tabell över hur lång tid ett antal element tog i algoritmen. Tabellen ritades ut i en graf och sedan gjordes en uppskattning av tidskomplexiteten Ordo. Genom att beräkna tiden dividerat med Ordo ska en konstant bevittnas om Ordo är rätt. Konstanten blir dock inte helt exakt på grund av felkällor (såsom processoroperationer i datorn tar lite olika tid beroende på processortemperatur etc).

f(n) = tiden.
g(n) = Ordo = tidskomplexitet.
c = lim(n→∞) f(n) / g(n)

Algoritm 1

Algoritm 1 Elements	**tid**	**c för O(n*log(n))**	c för O(log(n))	c för O(n)
0	0			
1,0E+06	3,32	5,533E-07	5,53E-01	3,32E-06
1,2E+06	4,0391	5,537E-07	6,64E-01	3,37E-06
2,0E+06	6,9722	5,533E-07	1,11E+00	3,49E-06
3,0E+06	10,8184	5,567E-07	1,67E+00	3,61E-06
4,0E+06	14,613	5,534E-07	2,21E+00	3,65E-06
5,0E+06	18,5334	5,533E-07	2,77E+00	3,71E-06
6,0E+06	22,5503	5,545E-07	3,33E+00	3,76E-06
7,0E+06	26,5594	5,543E-07	3,88E+00	3,79E-06
7,5E+06	28,5295	5,533E-07	4,15E+00	3,80E-06
8,0E+06	30,5765	5,537E-07	4,43E+00	3,82E-06

g(n) = n * log(n)

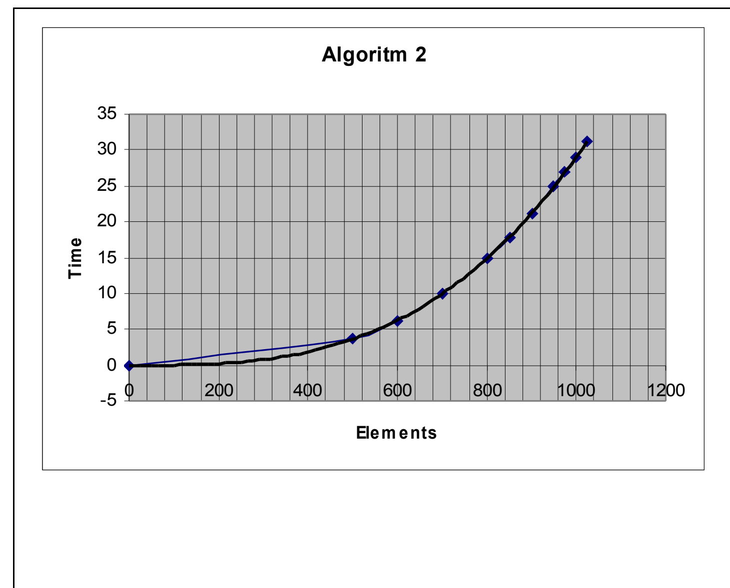


Efter testkörning var det tydligt att Ordo var nära n, d.v.s. en tillsynes rät graf (se bild ovan). För att bevisa Ordo undersöktes olika Ordo för vanliga algoritmer där Ordo(n*log(n)) bäst överensstämde (se tabell ovan) vid beräkning av konstant (se ovan).

Algoritm 2

Algoritm 2 Elements	tid	c för O(n^3)	c för O(n^2)	c för O(n^4)
0	0			
500	3,63	2,907E-08	1,454E-05	5,814E-11
600	6,27	2,905E-08	1,743E-05	4,841E-11
700	9,96	2,903E-08	2,032E-05	4,147E-11
800	14,86	2,903E-08	2,322E-05	3,629E-11
850	17,85	2,906E-08	2,470E-05	3,419E-11
900	21,15	2,902E-08	2,612E-05	3,224E-11
950	24,87	2,901E-08	2,756E-05	3,054E-11
975	26,95	2,907E-08	2,834E-05	2,982E-11
1000	29,01	2,901E-08	2,901E-05	2,901E-11
1025	31,24	2,901E-08	2,974E-05	2,830E-11

g(n) = n^3

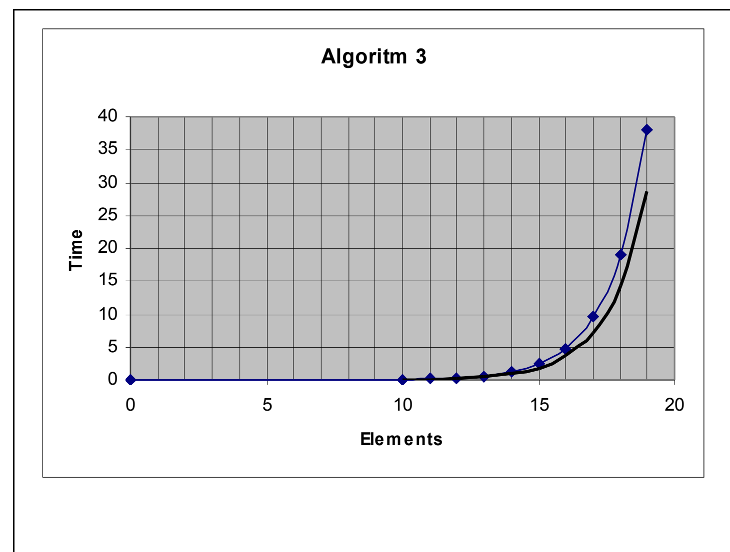


Efter testkörning var det tydligt att Ordo var exponentiell (se graf). För att bevisa Ordo undersöktes olika Ordo för vanliga algoritmer där Ordo(n^3) bäst överensstämde (se tabell ovan) vid beräkning av konstant (se ovan[2]).

Algoritm 3

Algoritm 3 Elements	tid	c för O(2^n)	c för O(1,5^n)	c för O(2,5^n)
0	0			
10	0,08	7,344E-05	1,304E-03	7,885E-06
11	0,15	7,324E-05	1,734E-03	6,291E-06
12	0,30	7,322E-05	2,311E-03	5,031E-06
13	0,60	7,312E-05	3,078E-03	4,020E-06
14	1,20	7,294E-05	4,093E-03	3,208E-06
15	2,38	7,250E-05	5,425E-03	2,551E-06
16	4,79	7,312E-05	7,296E-03	2,058E-06
17	9,58	7,308E-05	9,721E-03	1,646E-06
18	19,09	7,280E-05	1,291E-02	1,312E-06
19	37,99	7,246E-05	1,714E-02	1,044E-06

g(n) = 2^n



Efter testkörning var det tydligt att algoritm 3 blev extremt krävande redan vid små värden. Studering av ohanterliga algoritmer gav Ordo(2^n) som rimlig komplexitet (se tabell & graf) vid beräkning av konstant (se ovan[2]).

Del2

I den här delen skulle vi göra en asymptotisk komplexitetsanalys på given kod (se labbspec[1])

Swap

```
function swap(a, b)
begin
   temp    :=     a
           1      1     2
     a     :=     b     0
           1      1     2
     b     :=    temp   0
           1      1     2
end                     =
         Subtot:        6
```

Det som kostar 1 är:

- Tilldelning :=
- Referenser
- Metodanrop()
- Indexering i en array[]
- Jämförelse <=, >=, =
- Operationer +,-,/,*

Ordo(1)

c=lim(n→∞) 6 / 1 + 1 = 7 (c=f(n)/g(n))
n_0=1

Swap-funktionen kostar alltid lika mycket därför finns inget best- eller worst-case.

Beräkning av for-loopar

```
for i := 0 to (numElements – 1)
```

Utveckling av pseudokod till mer java liknande kod för for-loopar ger:

```
for (i := 0;i <= numElements – 1; i++)
```

För att avgöra kostnaden för hela processen är det dock bäst att tänka sig det i form av en while-loop:

```
i := 0;
while (i <= numElements - 1)
{
            i := i + 1;
}
```

När denna omskrivning har gjorts är det lättare att se att deklareringen av i är en engångskostnad, villkoret och additionen av 1 till i blir en kostnad för varje varv som körs d.v.s. n. I slutändan, d.v.s. efter ”sista varvet”, kommer dock jämförelsen göras men den kommer returnera falsk. Därför måste även en engångskostnad för detta läggas till.

for ([x];[y];[z])
Detta leder till:
f(n) = [x] + [y] + ([y]+[z])n[]

Oskar Mothander
dit06omr@cs.umu.se

Naiv bubblesort

Enklaste varianten av bubblestort. Till skillnad från den optimerade varianten undersöker den aldrig om listan är färdigsorterad i förtid. Utöver det loopar den över ett onödigt stort intervall.

```
function bubbleSort(numElements, list[])
begin
        for (i := 0;  i    <=      numElements    -   1;i  := i   +  1)
                1     1    1           1          1       1    1   1      5 (+7n+n[])
        begin
          for (j   := numElements             -         1;  j  >= 0; j  := j    +   1)
                 1   1                         1              1    1         1  1    1
          begin                                                                5 (+5n+n[])
               if  list[     j]             <           list[   j   -  1] then
                     1        1             1             1      1   1              6
               begin
                    swap(    list[          j],         list[   j   -  1])
                     1  6      1            1             1      1   1             12
               end
          end                                                                     18
        end
end
```

Engångskostnad	Kostnad för varje loop	Kostnad för metoden
Eventuell kostnad	Kostnad för metodanrop	
Worst case 1+7n+n(3+5n+n(18))		
Best case 1+7n+n(3+5n+n(6))		

Worst-case-scenario:

Det värsta fallet skulle vara ifall listan från början är helt osorterad. Då blir inte if-satsen falsk förren n antal försök vid körning av sorteringen.

```
5+7n+n(5+5n+n(18))
= 5+7n+n(5+5n+18n)
= 5+7n+n(5+23n)
= 5+7n+5n+23n^2
= 5+12n+23n^2
c = lim(n→∞) (5+12n+23^2) / n^2 +1 = 24

f(n) <= c*g(n) ger
5+12n+23n^2 <= 24n^2
5+12n <= n^2
0 = n^2-12n-5
```

$n_0 = 6 \pm \sqrt{(6^2)+5}$

$n_0 \approx 12.5 \approx$ 13 element

Kurs:	Datastrukturer & algoritmer	VT07
Lärare:	Lena Kallin Westin	
Handledare:	Lars Larsson Tommy Löfstedt	

Best-case-scenario:

För att få best-case för naiv bubblesort så måste if-satsen bli falsk, vilket inträffar enbart då listan redan är sorterad.

```
5+7n+n(5+5n+n(6))
= 5+7n+n(5+5n+6n)
= 5+7n+n(5+11n)
= 5+7n+5n+11n^2
= 5+12n+11n^2
c = lim(n→∞) (5+12n+11n^2) / (n^2) +1 = 12

f(n) <= c*g(n) ger
5+12n+11n^2 <= 12n^2
5+12n <= n^2
0 = n^2-12n-5
```

$n_0 = 6 \pm \sqrt{((6^2)+5)}$

$n_0 \approx 12.5 \approx$ 13 element

Optimerad bubblesort

Till skillnad från den naiva versionen loopar den inte över element som redan är i rätt position. Den avslutar dessutom listan om den blir färdigsorterad innan alla element gåtts igenom.

```
function bubbleSort(numElements, list[])
begin
  done        :=            false
              1
  i           :=            0
              1
     while (i < numElements) and (done = false)
               1 1       1        1    1    1   = 2 + 6n+n[] +3
     begin
              done := true
                    1
              for (j := numElements - 1; j >= i; j := j - 1)
                      1      1      1    1  1 1     1 1 1 = 1+3+3+(6n/2)+n[]
              begin
                   if list[j] < list[j - 1] then
                         1  1  1   1  1 1 = 6
                        begin
                             swap(list[j], list[j - 1])
                             1+6    1  1     1  1 1 = 12
                             done := false
                                      1                   = 6 + 13 = 19
                        end
              end
              i := i + 1
                1  1 1 = 3
      end
end
```

Worst-case-scenario:

Värsta fallet är ifall listan är sorterad i fel ordning så att första värdet är störst och sista värdet är minst.

```
  2 + 6n+n(1+3+3+((6n)/2)+n(19))+3
= 5 + 6n+n(7+((6n)/2)+19n)

= 5 + 6n+7n+((6n^2)/2)+19n^2
= 5 + 13n+3n^2+19n^2
= 5 + 13n+22n^2
c = lim(n→∞) (5 + 13n + 22n^2) / (n^2) +1 = 23

f(n) <= c*g(n) ger
5+13n+22n^2 <= 23n^2
5+13n <= n^2
0 = n^2-13n-5
```

$n_0 = 6.5 \pm \sqrt{(6.5^2) + 5}$

$n_0 \approx$ 14 element

Best-case-scenario:

I detta fall har 6n+n[] ersatts med 6*2+2[] eftersom villkoret i loopen bara kommer att kontrolleras två gånger om listan redan är sorterad. If-satsen kommer heller aldrig bli sann alltså kan koden inuti räknas bort.

```
  2 + 6*2+2(1+3+3+((6n)/2)+n(6))+3
= 5 + 12+2(7+3n+6n)
= 5 + 12+14+6n+12n
= 31+ 18n
c = lim (n → ∞) ((31+18n) / (n)) +1 = 19

f(n) <= c*g(n) ger
31+ 18n <= 19n
31 <= n
n≈  31 element
```

Motivering till varför man bör dela n med två för hela loop-kroppen:

Då i ökar och j minskar för varje gång loop-kroppen körs vill man få ut ett medelvärde på summan av det antal gånger loop-kroppen körs.
För att få ut det medelvärdet delar man n med två för loop-kroppen, vilket fås genom att den totala summan delas med två.

Totala summan:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Totala summan genom antalet gånger loopen körs:

$$\frac{\left(\frac{n(n+1)}{2}\right)}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2} \approx \frac{n}{2},\ n >> 1$$

In-place quickSort

Använder sig utav ett pivot-tal, dvs. ett referenstal. Detta tal tas från platsen längst till höger i listan och läggs i mitten på listan. De tal som är större än pivot-talet läggs sedan på platserna till höger i listan och lägre tal på platserna till vänster i listan. På det här sättet ”sorteras” varje del av listan tills hela listan är fullt sorterad.

I värsta fall är listan sorterad på så vis att det lägsta talet ligger på platsen längst till vänster och sorterad uppåt mot höger där det högsta värdet ligger på platsen längst till höger.
För att listan då ska bli rätt sorterad måste det bytas plats på alla värden. Ordo blir således n^2.

I bästa fall är pivot-talet listans medeltal för varje del som sorteras. Ordo blir då n*log(n).

RandomSort

Går ut på att den slumpar hela listan och sedan hoppas på att listan blir sorterad efter att algoritmen använts.

I bästa fall blir listans alla element sorterade vid första försöket vilket ger att Ordo blir n. Detta eftersom vi bara behöver titta på varje värde och konstatera att det ligger rätt.

Det värsta fallet kan visas genom följande exempel:
Man har n antal värden som man vill sortera efter en viss ordning, låt säga efter storleksordning. Det ger att det finns n antal platser som varje värde kan hamna i vid slumpningen. I värsta fall måste slumpningen ske n! antal gånger för varje n värde så att den hamnar på rätt plats. I slutändan skulle detta ge Ordo n*n!

Problem och reflektioner

Inga kommentarer.