Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Tommy Färnqvist

26 oktober 2015

TDDC91 Datastrukturer och algoritmer
Datortentamen (DAT1)
2015-10-26, 14–18

Examinator: Tommy Färnqvist
Jour: Tommy Färnqvist (telefon 013-282479).
Max poäng: 113 poäng (betyg 5 = 111p, 4 = 108p, 3 = 100p)
Hjälpmedel: Inga hjälpmedel tillåtna, förutom OpenDSA!

VÄNLIGEN IAKTTAG FÖLJANDE

- Lösningar till olika problem skall placeras enkelsidigt på separata blad. Skriv inte två lösningar på samma papper.
- Sortera lösningarna innan de lämnas in.
- MOTIVERA DINA SVAR ORDENTLIGT: avsaknad av, eller otillräckliga, förklaringar resulterar i poängavdrag. Även felaktiga svar kan ge poäng om de är korrekt motiverade.
- Om ett problem medger flera olika lösningar, t.ex. algoritmer med olika tidskomplexitet, ger endast optimala lösningar maximalt antal poäng.
- SE TILL ATT DINA LÖSNINGAR/SVAR ÄR LÄSBARA.
- Lämna plats för kommentarer.
- En student behöver inte göra något för att eventuella bonuspoäng ska tas med i poängberäkningen, detta sköts automatiskt vid rättning av tentamen.

Lycka till!

1. Efter inloggning i datortentasystemet finns i startmenyn för Linux Mint: (100 p)

   - "OpenDSA" (öppnar URL för tentamensversion av OpenDSA i Chromium) och
   - "Inloggningsuppgifter OpenDSA" (öppnar sida med inloggningsuppgifter till tentamensversion av OpenDSA i Chromium).

   Starta tentamensversionen av OpenDSA, logga in med inloggningsuppgifterna enligt ovan och lös de anvisade uppgifterna. Genom att klicka på ditt inloggningsnamn kan du se i betygsboken hur många av de poänggivande uppgifterna du löst hittills. När det står 10.00/10.00 i betygsboken är du färdig med den här uppgiften och kan logga ut. Du behöver inte skicka in något via tentamenssystemet. Om du inte löser samtliga poänggivande uppgifter får du noll poäng på den här tentamensuppgiften.

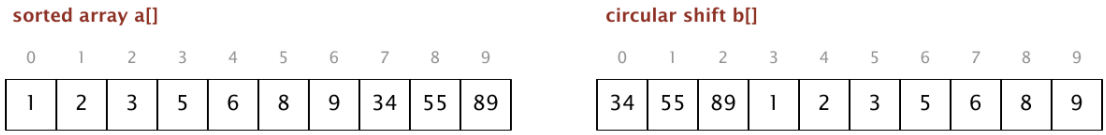
2. För var och en av kodsnuttarna nedan, ange den komplexitet (A–H) som bäst matchar dess exekveringstid. (5 p)

```
1. int x = 1, i;
   for (i = 0; i < N; i++)
     x++;

2. public static int f2(int N) {
      int x = 1;
      while (x < N)
         x = x * 2;
      return x;
   }

3. int x = 0, i;
   for (i = 0; i < N; i++)
      x += f2(N);

4. int x = 1, i, j;
   for (i = 0; i < N; i++)
      for (j = 1; j < R; j++)
         x = x * j;

5. int x = 0, i, j;
   for (i = 1; i <= N; i++)
      for (j = 1; j <= N+R; j += i)
         x += j;
```

(A) N (C) $N \log N$ (E) RN (G) $(N + R) \log N$

(B) $\log N$ (D) $R + N$ (F) $N + R^2$ (H) $N(N + R)$

3. Låt $a = a_0, a_1, \ldots, a_{N-1}$ vara en array av längd N. En array b kallas en *rotation* av a om den innehåller delarrayen $a_k, a_{k+1}, \ldots, a_{N-1}$ följd av delarrayen $a_0, a_1, \ldots, a_{k-1}$ för något heltal k. I exemplet nedan är b en rotation av a (med $k = 7$ och $N = 10$). (4 p)



Antag att du får en array b som är en rotation av någon *sorterad* array (men att du varken har tillgång till k eller den sorterade arrayen). Antag att arrayen b innehåller N unika, jämförbara, nycklar. Konstruera en effektiv algoritm för att avgöra om en given nyckel finns i arrayen b. Körtiden för din algoritm ska tillhöra $\mathcal{O}(\log N)$ i värsta fallet.

4. Redogör för en effektiv datastruktur för att lagra en samling genfragment över DNA-alfabetet $\{A, C, T, G\}$, enligt följande gränssnitt: (4 p)

`FragmentCollection()`	skapa en tom samling DNA-fragment
`void add(String fragment)`	lägg till DNA-fragmentet till samlingen
`int prefixCount(String p)`	antalet DNA-fragment som börjar med prefixet p

Exempel:

```
FragmentCollection fc = new FragmentCollection();
fc.add("AC");
fc.add("TACG");
fc.add("TCGAA");
fc.add("CGA");
fc.add("AGCT");
fc.add("TCGG");
fc.add("TCGG");       // added twice, will be counted twice
fc.prefixCount("");   // returns 7 (number of adds)
fc.prefixCount("T");  // returns 4 (TACG, TCGAA, TCGG, TCGG)
fc.prefixCount("TC"); // returns 3 (TCGAA, TCGG, TCGG)
fc.prefixCount("G");  // returns 0
```

Beskriv dina instansvariabler och, gärna med pseudokod, hur man kan implementera metoderna `add(String)` och `prefixCount(String)`. Vilken tidskomplexitet har din implementation av `prefixCount(String)` som funktion av antalet insatta fragment N, prefixets längd W, alfabetets storlek R och antalet fragment M som matchar det givna prefixet?