Några svar till exempeltentafrågor i TDDD86 Datastrukturer, algoritmer och programmeringsparadigm

5 december 2014

Följande är lösningsskisser och svar till exempeltentafrågorna. Lösningarna som ges här ska bara ses som vägledning och är oftast inte tillräckliga som svar på tentan.

Le 1 – Komplexitet, Algoritmanalys

1. (a) Kan vi hitta n_0 och $c > 0$ sådana att $(n+1)^2 \leq c \cdot n^3$ för alla $n \geq n_0$?

n	$(n+1)^2$	n^3
1	4	1
2	9	8
3	16	27

   Välj $c = 1$ och $n_0 = 3$. Då växer det kubiska polynomet snabbare än det kvadratiska.

   (b) Då ett kubiskt polynom växer snabbare än ett kvadratiskt kan vi inte hitta några $c > 0$ och n_0 sådana att

$$(n-1)^3 \leq c \cdot n^2 \text{ för alla } n \geq n_0.$$

   (c) Välj $c = 2$ och $n_0 = 1$:
   $2^{n+1} \leq 2 \cdot 2^n$ för varje $n \geq 1$.

   (d) Antag att påståendet är sant. Då finns det c och n_0 sådana att för varje $n > n_0$:

$$3^{n-1} \leq c \cdot 2^n$$

   Alltså

$$3^n \leq 3c \cdot 2^n$$

   och

$$(\frac{3}{2})^n \leq 3c$$

   för varje $n > n_0$, vilket inte kan vara sant eftersom c är en konstant. Alltså är påståendet falskt.

   (e) Antag att påståendet är sant. Då finns konstanter $c > 0$ och $n_0 \geq 0$ sådana att för alla $n > n_0$:

$$(n+1)^2 \geq c \cdot n^3$$

Alltså, för alla $n > n_0$:

$$\frac{(n+1)^2}{n^3} \geq c$$

vilket inte kan vara sant då c är en konstant. Alltså är påståendet falskt.

(f) Låt $c = 1$ och $n_0 = 4$:
$(n-1)^3 \geq n^2$ för alla $n \geq 3$

(g) Låt $c = 1$ och $n_0 = 0$:
$2^{n+1} \geq 2^n$ för alla $n \geq 0$

(h) Låt $c = 1$ och $n_0 = 3$:
$3^{n-1} \geq 2^n$ för alla $n \geq 3$

(i) Observera att $f(n) \in \Theta(g(n))$ om och endast om $f(n) \in \mathcal{O}(g(n))$ och $f(n) \in \Omega(g(n))$. Vi bevisade (se (e)) att $(n+1)^2 \notin \Omega(n^3)$. Alltså är påståendet falskt.

(j) falskt: följer från (b)

(k) sant: följer från (c) och (g)

(l) falskt: följer från (d)

2. Lemma: $f \in \mathcal{O}(g) \Rightarrow \mathcal{O}(f) \subseteq \mathcal{O}(g)$

(a) Visa först att $f(n) + g(n) \in \mathcal{O}(\max(f(n), g(n)))$:
För alla n, f, g har vi

$$f(n) + g(n) \leq 2 \cdot \max(f(n), g(n)).$$

Låt nu $c = 2$ och $n_0 = 1$ för att se att $f(n) + g(n) \in \mathcal{O}(\max(f(n), g(n)))$.

Visa sedan att $\max(f(n), g(n)) \in \mathcal{O}(f(n) + g(n))$:
Eftersom både $f(n)$ och $g(n)$ är positiva har vi: $f(n) \leq f(n) + g(n)$ och $g(n) \leq f(n) + g(n)$. Alltså gäller

$$\max(f(n), g(n)) \leq f(n) + g(n).$$

Tag t.ex. $c = n_0 = 1$ för att dra slutsatsen

$$\max(f(n), g(n)) \in \mathcal{O}(f(n) + g(n)).$$

Enligt lemmat ovan får vi nu $\mathcal{O}(\max(f(n), g(n))) = \mathcal{O}(f(n) + g(n))$.

(b) För alla h gäller $h \in \mathcal{O}(h)$, så vi får resultatet genom att sätta $h(n) = f(n) \cdot g(n)$.

3. Låt $f(n) = n^2$ och $g(n) = n$. Uppenbarligen gäller $\log f(n) = 2 \log n \in \mathcal{O}(\log g(n)) = \mathcal{O}(n)$, men $n^2 \notin \mathcal{O}(n)$. Alltså är påståendet falskt.

4. Eftersom $f \in \mathcal{O}(g)$, finns det konstanter $c_f > 0$ och $n_{0_f} \geq 0$ sådana att för alla $n \geq n_{0_f}$ gäller $f(n) \leq c_f g(n)$. På samma sätt finns konstanter c_g och n_{0_g} sådana att för alla $n \geq n_{0_g}$ gäller $g(n) \leq c_g h(n)$. Låt $n_0 = max(n_{0_f}, n_{0_g})$ och $n \geq n_0$. Då gäller $f(n) \leq c_f g(n) \leq c_f c_g h(n)$. Alltså gäller $f \in \mathcal{O}(h)$ och resp. konstanter är $c = c_f c_g$ och n_0 definierade som ovan.

5. Antag $f \in \Theta(g)$.
Då gäller $f \in \mathcal{O}(g)$ och $f \in \Omega(g)$.
Alltså har vi $g \in \Omega(f)$ och $g \in \mathcal{O}(f)$.
Det följer att $g \in \Theta(f)$.

6. - Mätningen gjordes på data med för liten storlek ($< n_0$).
   - Datat kan ha valts för att vara nära bästa fallet för A och nära värsta fallet för B.

- Ordonotationen ger bara en övre gräns för tidskomplexiteten. Om A has tidskomplexitet $\mathcal{O}(n^2)$ kan den också ha tidskomplexitet $\mathcal{O}(n)$ till exempel.

7. (a) Ja. Följer av (b).

   (b) Ja. Vi vet att värstafallskomplexiteten hos algoritmen är minst $\Theta(n)$. Den kan alltså mycket väl vara värre, som t.ex. $\Theta(n^2)$. Vi har ingen information om exekveringstiden i bästa fallet, men för vissa algoritmer kan bästa- och värstafallskomplexiteten sammanfalla.

   (c) Ja. För något indata kan algoritmen ha högre komplexitet än för det bästa indatat.

   (d) Nej. För det bästa indatat karakteriseras exekveringstiden av en funktion $f(n) \in \Theta(n)$ och $f(n) \notin \Omega(n^2)$.

8. (a) Numrera for-looparna från 1 till 3.

   Förenklad analys av varje loop: 1: $\Theta(n)$, 2: $\mathcal{O}(n)$, 3: $\mathcal{O}(n)$, sammantaget: $\Theta(n) \cdot \mathcal{O}(n) \cdot \mathcal{O}(n) \cdot \mathcal{O}(1) = \mathcal{O}(n^3)$

   Mer precis analys: Byt första $\mathcal{O}(1)$ till en konstant c.

   loop 3:
   $\sum_{k=1}^{j} c = cj$
   loop 2:
   $\sum_{j=i+1}^{n} cj = c(n-(i+1)+1)\frac{i+1+n}{2} = \frac{c}{2}(n-i)(1+i+n)$
   loop 1:
   $\sum_{i=1}^{n-1} \frac{c}{2}(n-i)(1+i+n) =$
   [lång uträkning utelämnad] =
   $\frac{c}{3}n(n-1)(n+1) \in \Theta(n^3)$

   (b) **Förenklad analys:** Båda inre looparna använder tid $\Theta(n)$. Den inre loopen exekveras $\Theta(n)$ gånger, kroppen i if-satsen exekveras varannan gång av dessa. Detta ger:
   $\Theta(n) \cdot \frac{1}{2}\Theta(n) = \Theta(n^2)$

   (c) $T(n) = \sum_{i=1}^{n} \sum_{j=3}^{n-1} 5 \cdot c \log n = 5c \sum_{i=1}^{n} (n-4) \log n = 5cn(n-4) \log n =$
   $= 5cn^2 \log n - 20cn \log n \in \Theta(n^2 \log n)$

   (d) Ytterloopen körs n gånger. Innerloopen körs: en gång för $i = 1$, två gånger för $i = 2$,..., $1 \cdot 2 \cdot 3 ... \cdot k$, d.v.s $k!$ gånger för $i = k$. Detta ger tidskomplexiteten $\Theta(n!)$.

9. Analys av tidskomplexiteten för foo:

$$\sum_{i=1}^{4} \Theta(1) = 4\Theta(1) = \Theta(1)$$

   - $\mathcal{O}(0)$ inkluderar endast funktioner som uppfyller $\to 0$ då $n \to \infty$
   - $\mathcal{O}(7) = \mathcal{O}(1) \supseteq \Theta(1)$
   - $\mathcal{O}(n) \supseteq \Theta(1)$
   - $\Omega(7) = \Omega(1) \supseteq \Theta(1)$

   Följdaktligen: $T(n) \in \mathcal{O}(7),\ T(n) \in \mathcal{O}(n),\ T(n) \in \Omega(7)$

10. Det här är en variant av binärsökning. Låt k vara det minsta naturliga tal sådant att $n \leq 2^k$. I värsta fallet exkeveras då innerloopen $k+1$ gånger. Alltså är värstafallskomplexiteten $\Theta(\log n)$

11. $T(0) = 1$
$T(n+1) = T(n) + 1$

Alltså gäller $T(n) = n + 1 \in \Theta(n)$.

12. Den första for-loopen initialiserar arrayen så att alla element är 0. Det tar $\mathcal{O}(n)$ tid.

Nu följer en analys av `loop1`. Den första for-loopen skriver ut innehållet i arrayen `A`. Det tar $\mathcal{O}(n)$ tid. Den kluriga biten är `loop2`: Först ökar loopen värdet i position 1 i `A` med 1. Efter första iterationen av `loop1`, innehåller alltså `A` en etta i position 1 och nollor i resterande positioner. I nästa iteration ökas position 1 igen och värdet blir 2. I det här läget återställs position 1 till noll igen och nästa position ökas med 1. Denna position kollas igen för att se om dess värde blivit 2. Om så är fallet återsälls värdet och nästkommande position undersöks.

I värsta fallet måste `loop2` undersöka alla positioner i arrayen vilket betyder att tidskomplexiteten för `loop2` är $\mathcal{O}(n)$. Notera att den här loopen använder `A` som en binär räknare där den första positionen är den minst signifikanta: Vid inträde i loopen ökas räknaren med ett. Innehållet i räknaren ändras inte i `loop1` som bara skriver ut räknaren och går in i `loop2` med $i = 1$ igen.

Den stora frågan är nu när blir $i = n$, alltså när uppfylls villkoret för att avbryta `loop1`? Detta händer när alla positioner i `A`, efter att ha blivit ökade, utom den sista har värdet 1. Eftersom algoritmen räknar, med binära tal, från 0 till talet med n 1;or, måste `loop1` iterera precis så många gånger, d.v.s. $\mathcal{O}(2^n)$.

Tidskomplexiteten sammantaget: $T(n) \in \mathcal{O}(n) + \mathcal{O}(2^n) \cdot (\mathcal{O}(n) + \mathcal{O}(n)) = \mathcal{O}(n2^n)$

Att analysera rumskomplexiteten är enkelt. Arrayen använder $\Theta(n)$ minne, vilket utgör algoritmens hela rumskomplexitet.

13. Vi använder följande notation för operationerna:

- *send* nästkommande indatatecken skickas direkt till utskrift.
- *push* nästkommande indatatecken pushas på stacken.
- *pop* stacken poppas och tecknet som resulterar skickas till utskrift.
- *enqueue* nästkommande indatatecken placeras i kön.
- *dequeue* ett tecken tas bort från kön ich skickas till utskrift.

(a) Permutationen X U W V Y kan produceras med följande operationer:

send, *push*, *send*, *push*, *push*, *pop*, *pop*, *pop*

Permutationen går inte att åstadkomma genom att använda en kö. Efter att ha skickat X direkt till utskrift måste vi läsa Y. För att få till den önskade utsekvensen kan vi inte skicka tecknet till utskrift. Vi kan bara sätta in tecknet i kön. Då måste U skickas direkt till utskrift eftersom U ska komma före Y i den önskade permutationen. Nästa tecken är V. Vi kan inte skicka V till utskrift och om vi köar tecknet får vi fortfarande inte till permutationen eftersom det kommer att föregås av det redan köade Y.

(b) Permutationen Y U W X V går inte att få till med hjälp av en stack. Den längsta möjliga sekvensen av operationer som inte bryter mot kravet är *push*, *send*, *send*. Nästa indatatecken är V. Vi kan inte skicka det till utskrift. Om vi pushar tecknet på stacken kommer V att hamna för X som för tillfället befinner sig på stacken.

Permutationen kan genereras med en kö:

enqueue, *send*, *send*, *enqueue*, *send*, *dequeue*, *dequeue*

14. Beteckna stackarna $S1$ and $S2$. Elementen som pushas på $S1$ lagras i cellerna $T[0], T[1], ..., T[k_1]$ där k_1 är index för top när $S1$ innehåller $k_1 + 1$ element. På liknande sätt lagras elementen som pushas på $S2$ i $T[n], T[n-1], ..., T[n-k_2]$ där $n - k_2$ är index för top när $S2$ innehåller $k_2 + 1$ element. Overflow-fel returneras vid en push-operation om $k_1 + k_2 = n + 1$.

15. I vad som följer antar vi att *input* läser och returnerar ett tecken från indataströmmen.

```
function palindrome(n): boolean
   makeEmptyStack(S);
  for i = 1 to ⌊n/2⌋ do
     Push(S, input)
  if 2 * ⌊n/2⌋ ≠ n then input
  for i = 1 to ⌊n/2⌋ do
    if  Pop(S) ≠ input then return false
  return true
```

Körtiden i värsta fallet är $\mathcal{O}(n)$ (motivering utelämnad).

16. (a)
    - *MakeEmptyQueue*: skapa två tomma stackar: E and D.
    - För att göra *enqueue* på dataelement d pusha det på E.
    - För att göra *dequeue*:
      Om D är tom och E är tom rapportera ett fel.
      Annars:
      i. Om D är tom, pappa ett efter ett alla element på E och pusha dem på E. Efter detta finns all köad data på D från topp till botten i köordning. *Front*-elementet finns i toppen av D och vi poppar det för att slutföra dequeue-operationen.
      ii. Annars poppar vi från D.
    - *Front*-operation är som *dequeue* men dataelementet poppas inte från D.
    - *IsEmptyQueue* implementeras med två kontroller: *IsEmptyStack(E)* och *IsEmptyStack(D)*; returnera *true* omm båda kontrollerna returnerar *true*.

    (b) Pseudokod inte inkluderad

    (c) Vi antar att alla stackoperatioener för i tid $\Theta(1)$.
    i. *Enqueue* är alltid en push på E, så den kör i konstant tid $\Theta(1)$.
    ii.
    - En enstaka *enqueue*-operation implementeras som en push på E, exekverar därför **alltid** i $\Theta(1)$ tid; det finns inget särskilt värsta fall.
    - Värsta fallet för *dequeue* är när operationen behöver utföras efter att n dataelement pushats på E. I det fallet behöver vi:
      (1) tomhetskontroll av båda stackarna i tid $\Theta(1)$
      (2) n pop-operationer och n tomhetskontroller på E i $\Theta(n)$ tid
      (3) n push-operationer på D i $\Theta(n)$ tid
      (4) en pop-operation på D i $\Theta(1)$ tid
      Alltså har värsta fallet för dequeue tidskomplexitet $\Theta(n)$.

    (d) Vare dataelement i sekvensen pushas bara en gång på E, poppas från E, pushas på D och poppas från D. Alltså är den totala kostnaden får att göra enqueue och dequeue på alla n dataelement $\Theta(n)$, och den amorterade kostnaden per dataelement $\Theta(1)$.

17. Lösningen är en enkel modifiering av en *ring buffer* eller *cirkulär array*. Vi lagrar alltså elementen i en dequeue D i en vektor $T[0..N-1]$. Två variable F och L används för att

hålla reda på index för front respektive längden av den faktiskt lagrade sekvensen. En tom dequeue skapas fenom att sätta F och L till 0 och allokera vektorn T. Om D är icke-tom lagras dess elements $d_0, ..., d_{L-1}$, i positionerna $T(F), T(F+1) \mod N, \ldots, T(F+L) \mod N$. Köoperationerna utförs som följer:

- *addEnd*(E, D): Om $L = N$ rapportera ett fel. Öka annars L med ett och placera E i $T((F + L - 1) \mod N)$.
- *addFront*(E, D): Om $L = N$ rapportera ett fel. Oka annars L med ett och placera E i $T(N - 1)$ om $F = 0$ och i $T(F - 1)$ om $F \neq 0$. Uppdatera F.
- *deleteEnd*(D): Om $L = 0$ rapportera ett fel. Returnera annars $T((F + L - 1) \mod N)$ och minska L med ett.
- *deleteFront*(D): Om $L = 0$ rapportera ett fel. Returnera annars $T(F)$, minska L med ett och öka F med ett modulo N.

18. (a) Den här algoritmen beräknar a^n. Körtiden för algoritmen är $O(n)$ eftersom

    - de initiala tilldelningarna tar konstant tid
    - varje iteration av **while**-looopen tar konstant tid
    - det blir exakt n iterationer

    (b) Den här algoritmen beräknar också a^n. Körtiden för algoritmen är $O(\log n)$ av följande skäl:

    Initialiseringen och **if**-satsen med sitt innehåll tar konstant tid, så vi behöver lista ut hur många gånger **while**-loopen nås. Eftersom k minskar (antingen halveras eller minskas med ett) i varje steg och är lika med n initialt kommer loopen som värst att exekveras n gånger. Men vi kan göra en bättre analys.

    Notera att k halveras om k är jämnt och att k minskas med ett och halveras i nästa iteration om k är udda. Så åtminstone varannan iteration av **while**-loopen halverar k. Det går att halvera ett tal n som mest $\lceil \log n \rceil$ gånger innan det blir ≤ 1. (Varje gång vi halverar ett tal skiftar vi det åt höger med en bit, och ett tal n har $\lceil \log n \rceil$ bitar.) Om vi minskar talet med ett mellan att vi halverar det kan vi fortfarande inte halvera det fler än $\lceil \log n \rceil$ gånger. Eftersom vi bara kan minska k med ett mellan två iterationer som halverar k (om inte n är udda eller det är sista iterationen) får vi göra en interation som minskar k med ett högst $\lceil \log n \rceil + 2$ gånger. Så vi har som mest $2\lceil \log n \rceil + 2$ iterationer. Vilket ger oss tidskomplexiteten $O(\log n)$.

19. Den här algoritmen kommer att använda en sorteringsalgoritm – vilken sorteringsalgoritm som helst med $O(n \log n)$ tidskomplexitet i värsta fallet duger (t.ex. merge-sort).

    Vi ger en grov beskrivning av algoritmen följt av pseudokod.

    - Först beräknar vi en ny sekvens C, med storlek n, sådan att för varje i mellan 0 och $n - 1$ gäller $C[i] = m - A[i]$. Detta tar $O(n)$ tid.
    - Vi sorterar C, vilket tar $O(n \log n)$ tid.
    - Vi sorterar B, vilket tar $O(n \log n)$ tid.
    - Vi söker genom C och B från början och letar efter element som är lika. Eftersom C och B är sorterade behöver vi som värst göra $2n$ jämförelser. Därför tar det här steget $O(n)$ tid.
    - Om C och B har ett gemensamt element (säg b) finns det ett element a i A sådant att $b = m - a$. Notera att vi inte behöver hitta något sådant element enligt problembeskrivningen, det räcker att bekräfta att ett sådant element finns.

Alltså blir exekveringstiden för den här algoritmen $O(n + n\log n + n\log n + n) = O(n\log n)$

Require: array A, array B, storlek n på A och B, heltal m
Ensure: avgör om det finns $a \in A$ och $b \in B$ sådana att $a + b = m$
function isThereASum(A, B, n, m)
låt C vara en ny array med storlek n
for $i \leftarrow 0$ **to** $n - 1$ **do**
$C[i] \leftarrow m - A[i]$
sort(C)
sort(B)
$cind \leftarrow 0$
$bind \leftarrow 0$
while $cind \neq n$ och $bind \neq n$ **do**
if $C[cind] = B[bind]$ **then**
return *true*
if $C[cind] \leq B[bind]$ **then**
$cind \leftarrow cind + 1$
else
$bind \leftarrow bind + 1$
return $false$

20. Det här kan åstadkommas genom att modifiera Dijkstras algoritm. I stället för att representera den kortaste vägen från a till u låter vi märkningen $D[u]$ representera den maximala bandbredden över alla vägar från a till u. Den maximala bandbredden för en väg från a via u till en nod z som är granne till u är $\min\{D[u], w((u,z))\}$ så att relaxeringssteget uppdaterar $D[z]$ till $\max\{D[z], \min\{D[u], w((u,z))\}\}$.

Require: Viktad sammanhängande enkel graf G och två skiljda noder a och b
Ensure: Returnerar den maximala bandbredden över alla vägar från a till b
function maxBandwidth(G, a, b)
initialisera $D[a] \leftarrow \infty$ och $D[u] \leftarrow 0$ för varje nod $u \neq a$ i G
låt en prio-kö Q innehålla alla noder i G med D-värdena som nycklar
while Q är icketom **do**
$u \leftarrow Q$.removeMax()
if $u = b$ **then** ▷ hittade slutnoden — kan stanna här
return $D[u]$
else
for each granne z till u som är i Q **do**
$d \leftarrow \min\{D[u], w((u,z))\}$
if $d > D[z]$ **then**
$D[z] \leftarrow d$
ändra z:s nyckelvärde i Q till $D[z]$

En avslutande **return**-sats är onödig eftersom $u = b$ vid något tillfälle i huvudloopen varför algoritmen terminerar och returnerar resultatet då.

Genom att representera grafen med en grannlista blir exekveringstiden densamma som för Dijkstras algoritm – $O((n + m)\log n)$, där n är antalet noder i G och m antalet bågar i G, om prioritetskön implementeras m.h.a. en heap och $O(n^2)$ om prioritetskön implementeras med en osorterad sekvens. (Det går att komma ner till $O(n\log n + m)$ genom att använda en ännu tjusigare datastruktur för prioritetskön.)

21. (a)
```
res = 0;
for (i = 0; i <= n; i++) {
  ai = a[i];
  xi = 1;
```

```
  for (j = 0; j < i; j++) {
    xi = xi * x;
  }
  res = res + ai * xi;
}
return res;
```

Tidskomplexitet $\Theta(n^2)$.

(b)
```
res = a[n] * x;
for (i = n-1; i >= 1; i--) {
  res = res + a[i];
  res = res * x;
}
res = res + a[0] ;
return res;
```

Detta är ett exempel på dynamisk programmering.

22. 240 finns efter 911, så vi letar i vänster delträd till 911. Där är alla element mindre än eller lika med 911 så alla nycklar i sekvensen måste vara mindre än 911. Men 912 besöks efter 240 vilket är en motsägelse.

23. (a)
```
foo(S,C,5,0,7)                          // S is (C,H,B,E,J,G,D,A)
  partition(S,C,0,7)
    swap elements at ranks 0 and 7      // S is now (A,H,B,E,J,G,D,C)
    return 0
  foo(S,C,5,1,7)                        // c is 0; choose c < k case since 0 < 5
    partition(S,C,1,7)
      swap elements at ranks 1 and 2    // S is now (A,B,H,E,J,G,D,C)
      swap elements at ranks 2 and 7    // S is now (A,B,C,E,J,G,D,H)
      return 2
    foo(S,C,5,3,7)                      // c is 2; choose c < k case since 2 < 5
      partition(S,C,3,7)
        swap elements at ranks 4 and 6 // S is now (A,B,C,E,D,G,J,H)
        swap elements at ranks 6 and 7 // S is now (A,B,C,E,D,G,H,J)
        return 6
      foo(S,C,5,3,5)                    // c is 6; choose c > k case since 6 > 5
        partition(S,C,3,5)
          swap elements at ranks 7 and 7   // S is now (A,B,C,E,D,G,H,J)
          return 5
      return G                          // c is 5; choose c = k case since 5 = 5
    return G
  return G
return G
```

Resultatet av `foo(`S`,`C`,5,0,7)` är G.

(b) Om $a = 0$ och $b = S.\text{size}() - 1$ så returnerar `foo(`S`,`C`,`k`,`a`,`b`)` elementet vid rank k i den sorterade sekvensen. (Annars finns två möjligheter. Om $k < a$ eller $k > b$ får vi en krasch. Om $a \leq k \leq b$ returneras elementet vid rank $k - a$ i den sorterade sekvensen bestående av element med rank mellan a och b i indatat S.)

24. Lösningen använder en teknik som påminner om merge sort.

Find-2-Numbers(S, x, n)
S is a set of n numbers.
x is the sum we are aiming for.

(a) $A \leftarrow MergeSort(S)$

(b) $left \leftarrow 0$

(c) $right \leftarrow (n-1)$

(d) while $(left < right)$

(e) if $((A[left] + A[right]) = x)$

(f) then return TRUE.

(g) else if $((A[left] + A[right]) < x)$

(h) $left \leftarrow (left + 1)$

(i) else

(j) $right \leftarrow (right - 1)$

(k) return FALSE.

Steg (a) tar $O(n \log n)$ tid och resten av stegen tar $O(n)$ tid. Algoritmen evekverar i tid $O(n \log n)$.

Ett alternativt sätta att göra detta skulle kunna vara att sortera arrayen och att sedan, för varje y i arrayen, söka efter $(x - y)$. Varje sökningen skulle ta tid $O(\log n)$ och algoritmen totalt $O(n \log n)$ tid.

25. (a) **Möjligt**. Använd insertion sort. 1000 är en konstant, så detta är "nästan sorterat data".

    (b) **Omöjligt**. Måste titta på allt data $\Rightarrow \Omega(n)$ tid.

    (c) **Omöjligt**. Jämförelsebaserad sortering gör $\Omega(n \log n)$ jämförelser i värsta fallet.

26. $O(mn)$ resp. $O(m + n)$.

27. (a)
```
tl (root r)
  if (r == null)
    return 0
  else
    return tl(r, 0)
}

tl (node r, len x) {
  s1 = 0
  s2 = 0
  if (r.hasLeft)
    s1 = tl(r.left, x+1)
  if (r.hasRight)
    s2 = tl(r.right, x+1)
  return x + s1 + s2
}
```

    Algoritmen har linjär tidskomplexitet.

    (b) Om a eller b finns är svaret ja. Annars? Att söka efter $a + 1, a + 2, \ldots, b - 1$ blir väldigt långsamt. (Beror på värdena, inte *antalet* noder i trädet.) Om sökning efter a och b terminerar i a_0 och b_0 $(a_0 \neq b_0)$, så finns specifik vikt mellan a och b: förfadern till a_0 och b_0 är en av dem. Om båda misslyckade sökningarna terminerade i samma löv är svaret nej. Tidskomplexiteten blir $O(\log n)$.

28. Algoritmen liknar binärsökning.

```
public static int max (int[] a, int lo, int hi) {
   if (hi == lo) return a[hi];
   int mid = lo + (hi - lo) / 2;
   if      (a[mid] < a[mid + 1]) return max(a, mid + 1, hi);
   else if (a[mid] > a[mid + 1]) return max(a, lo, mid);
   else                          return a[mid];
}
```

29. (a) Algoritm 1: Iteration i: $O(i)$ tid för att kolla om tal ej redan finns. Förväntat antal slumptal som behöver genereras för `a[i]` är $N/(N-i)$ eftersom i av talen är dubletter (och sannolikheten att få icke-dublett är $(N-1)/N$). Vi får

$$\sum_{i=0}^{N-1} \frac{Ni}{N-i} \leq \sum_{i=0}^{N-1} \frac{N^2}{N-i} \leq N^2 \sum \frac{1}{N-i} \leq N^2 \sum_{j=1}^{N} \frac{1}{j} \in O(N^2 \log N).$$

$\sum_{j=1}^{N} \frac{1}{j} = H_N$, det N:te harmoniska talet. Antingen känner man till att $\lim_{n\to\infty}(H_n - \ln n) = \gamma \approx 0.5772\ldots$ eller så kan man använda att $\int_1^N \frac{1}{x}\mathrm{d}x = \ln n$.

Algoritm 2: Sparar faktor i för varje position $\Rightarrow O(n \log n)$ förväntad tid.

Algoritm 3 är alltid linjär.

(b) Det finns ingen begränsning på värstafallstiden för algoritm 1 och 2 eftersom det alltid finns en nollskiljd sannolikhet att programmet inte terminerat vid en given tid T.

30. (a)

0	1	2	3	4	5	6
G	D	B	F	E	A	C

(b) I är resultatet av insättning i ordning F, D, B, G, E, C, A. II är omöjlig, då första nyckeln som sätts in hamnar på en plats i tabellen som motsvarar dess hashvärde, men ingen nyckel har denna egenskap. III är omöjlig — B och G är i rätt position, så vi kan anta att de sattes in först. Men då skulle den tredje nyckeln också vara i rätt position.

31. (E)

32. Sortera `a[]` med heapsort (genom att använda någon naturlig ordning för punkterna). För varje punkt `b[j]` använd binärsökning för att leta efter punkten i den sorterade arrayen `a[]`. Körtiden är $M \log M$ för sorteringen och $N \log M$ för de N binärsökningarna. Både heapsort och binärsökning kan göras in-place.

33. - `sample()`: Välj ett slumpmässigt arrayindex `r` (mellan 1 och N) och returnera nyckeln `a[r]`.

    ```
    public Key sample() {
       int r = 1 + Random.uniform(N);
       return a[r];
    }
    ```

    - `removeRandom()`:
      - Välj ett slumpmässigt arrayindex `r` (mellan 1 och N) och spara undan nyckeln `a[r]` som ska returneras.
      - Byt plats på `a[r]` och `a[N]` och minska `N` med ett.
      - Återställ heapordningen genom att utföra anropen `downHeap(r)` och `upHeap(R)` för att fixa till att heapordningen ev. inte är uppfylld kring `r`. Notera att `a[N]` i ursprungsheapen inte behöver ha varit den största nyckeln i heapen, så anropet till `upHeap(R)` är nödvändigt.

```
public Key removeRandom() {
   int r = 1 + Random.uniform(N);
   Key key = a[r];
   swap(r, N--);
   downHeap(r);    // om a[N] var för stor
   upHeap(r);      // om a[N] var för liten
   a[N+1] = null;  // städa upp
   return key;
}
```

34. 1:a, 2:d, 3:e, 4:c, 5:f, 6:b

35. Idén är att om det finns *någon* trippel $x < y < z$ som i uppgiftslydelsen så är det också sant att $\min_{T_B} < y < \max_{T_B}$. Alltså letar vi först upp det minsta och det största elementet i T_B och söker sedan efter dessa i T_A. Beroende på resultatet av dessa sökningar kan vi avgöra om vi ska returnera `true` eller `false`. Det blir en del specialfall att hålla reda på, men om vi använder t.ex. AVL-träd klarar vi oss med $O(\log n)$ tid för att leta reda på min, max och de två modifierade sökningarna.

36. (a) `j` kan bli så stort som $\mathtt{i}^2$, vilket kan bli så stort som $\mathtt{n}^2$. `k` kan bli så stort som `j`, vilket är $\mathtt{n}^2$. Exekveringstiden är därför proportionell mot $\mathtt{n} \cdot \mathtt{n}^2 \cdot \mathtt{n}^2$, vilket är $O(\mathtt{n}^5)$.

    (b) if-satsen exekveras som mest $\mathtt{n}^3$ gånger, enligt argumentet ovan, men är endast sann $O(\mathtt{n}^2)$ gånger (eftersom den är sann exakt `i` gånger för varje `i`). Alltså exekveras den innersta loopen bara $O(\mathtt{n}^2)$ gånger. Varje gång tar den $O(\mathtt{j}^2)$, eller $O(\mathtt{n}^2)$ tid, vilket ger totalt $O(\mathtt{n}^4)$ tid. Det fungerar inte alltid att multiplicera looparnas storlek.

37. Lösningsförslag: Välj en godtycklig nod v och färga den red. Färga alla dess grannar blue och kolla om någon konflikt uppstått. Om inte, färga alla noder som gränsar till de blue-färgade red och så vidare. Antingen lyckas man färga hela grafen eller så uppstår en konflikt någonstans. Det kan vara lite för mycket begärt att man ska prestera ett helt formellt korrekthetsbevis men en vettig diskussion kan man kräva.

38. Använd en skiplista för att implementera ADT Dictionary. Varje nod på bottennivån pekar på en länkad lista med element med samma nyckel. På så sätt fungerar alla uppdateringsoperationer i $O(\log r)$ tid, medan `getAll` exekverar i tid $O(\log r + s)$.

39. (a) Om inexakta sökningar behövs. Om vi, till exempel, vill hitta det element med nyckel mindre än eller lika med 5 som är närmast 5. Hashtabeller kan bara utföra exakta sökningar.

    (b) Om varje enskild operation måste ha körtid $O(\log n)$. Eller om de flesta operationerna är `find()` och accessmönstret är likformigt fördelat.

    (c) Om minnesanvändning är absolut högst prioriterat.

40. En möjlig lösning: Använd DFS för att identifiera bågarna som behöver tas bort för att göra grafen acyklisk. Det totala antalet "back"-bågar är det minsta antalet bågar som behöver tas bort för att göra grafen acyklisk. Algoritmen använder $O(|V| + |E|)$ tid.