Några svar till exempeltentafrågor i TDDC91 Datastrukturer och algoritmer

15 oktober 2015

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.

1. (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 $O(\log n)$.

2. Notera att om v är en nod i T som är större än x så är alla nycklar lagrade i dess delträd också större än x. Alltså är det tillräckligt att söka med start i roten efter noder i T som är mindre än eller lika med x. Vi visar här ett rekursivt sätt att göra detta på. Genom att använda en kö eller en stack går det att formulera iterativa lösningar.

   Require: en heap T, en nod v i T, en frågenyckel x
   Ensure: hittar nycklarna i delträdet av T rotat i v som är mindre än eller lika med x

   ```
   procedure findSmaller(T, v, x)
       if v ≠ null then
           if (v.key()) ≤ x then
               reportKey(v.key())
               findSmaller(T, T.leftChild(v), x)
               findSmaller(T, T.rightChild(v), x)
   ```

   Exekveringstiden för algoritmen blir $O(k)$ av följande skäl. När vi flyttar oss nedåt i trädet tittar vi på varje nod med en nyckel mindre än eller lika med x exakt en gång. De enda övriga noder vi tittar på är deras barn. Givet k hittade nycklar har de k motsvarande noderna i heapen $2k$ barn. Trots det faktum att några av deras barn också är bland noderna vars nycklar är $\leq x$ ger detta oss fortfarande den sökta övre gränsen: vi tittar på $3k$ noder, vilket är $O(k)$ noder.

3. 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.)

4. (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.

5. Push: gör enqueue på Q_1. Tar $O(1)$ tid.

   Pop: dequeue:a alla element i Q_1 och enqueue:a dem i Q_2, förutom sista elementet som vi sparar i en temporär variabel. För tillbaka elementen till Q_1 genom att dequeue:a dem från Q_2 och enqueue:a dem i Q_1. Returnera temporärvariabelns element. Pop kommer att ta $\Theta(n)$ tid.

6. 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.

7. (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 5 and 5   // 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 $b = S$.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.)

8.
```
x <- S.pop()
if x < S.top() then
  x <- S.pop()
```
   Notera att om det största heltalet är det första eller andra elementet i S så lagras det i x. Alltså lagrar x det största elementet med sannolikhet 2/3.

9. 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 sökning skulle ta tid $O(\log n)$ och algoritmen totalt $O(n \log n)$ tid.

10. Den grundläggande idén är att om två prov är olika så kan de förkastas eftersom ett av dem inte tillhör majoriteten och majoriteten bevaras om vi gör detta.

    Algoritmen behandlar proverna ett och ett och håller reda på en kandidat c till att vara ett majoritetsprov och dess "multiplicitet" k. Invarianten vi använder är att om vi lägger till k kopior av prov c till de obehandlade proverna så är majoritetsprovet i ursprungsuppsättningen av prover och denna nya uppsättning prover samma.

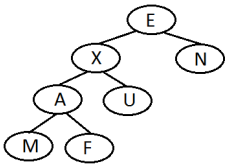
    Algorithm 0.1: FindMajority(S)

    $k \leftarrow 0$
    for $i \leftarrow 0$ **to** $n - 2$
    do
    - **if** $k = 0$
      - **then**
        - $k \leftarrow 1$
        - $c \leftarrow S[i + 1]$
      - **else**
        - **if** $S[i + 1] = c$
          - **then** $k \leftarrow k + 1$
          - **else** $k \leftarrow k - 1$

    return (c)

    Alternativa lösningar skulle kunna innebära att gå genom arrayen med prover och jämföra varje konsekutivt par. Om ett par innehåller distinkta prov, kasta bort dem. Om ett par innehåller två "likadana" prov, behåll ett av dem. Utför sedan samma procedur på resterande prover. I varje sådant svep över alla kvarvarande prover kastas åtminstone hälften av proverna bort. Låt oss, för enkelhets skull, anta att $n = 2^k$. Då är antalet jämförelser som behöver utföras inte fler än $n/2 + n/4 + n/8 + \ldots + 1 = 2^{k-1} + \ldots + 1 = 2^k - 1 = n - 1$.

11. (a)



(b)

Djupet av en nod v är lika med djupet av föräldern plus ett. Därför kommer vår algoritm att härma en preordertraversering (varje förälder måste behandlas före sina barn). För att beräkna djupet av varje nod i T anropas följande algoritm med T och T.root() som indata.

Algorithm 0.2: ComputeDepth(T, v)

```
if T.isRoot(v)
  then setDepth(v, 0)
  else setDepth(v, 1 + getDepth(T.parent(v)))
children ← T.children(v)
while children.hasNext()
  do { child ← children.next()
     { ComputeDepth(T, child)
```

Vi antar att children(v) går i tid $O(c_v)$ i värsta fallet, där c_v är antalet barn v har. Då tar varje rad i **if** -satsen tid $O(1)$. Tilldelningen till *children* tar $O(c_v)$ tid och **while** -loopen exekveras också c_v gånger för att rekursivt beräkna djupen av alla noder i delträdet rotat i v. Om vi exkluderar de rekursiva anropen tar ComputeDepth $O(c_v)$ tid.

Hur lång tid tar ComputeDepth$(T, T.\text{root}())$? Notera att för varje nod v i T anropar vi ComputeDepth(T, v) exakt en gång (eftersom varje nod har som mest en förälder och vi börjar med roten i T). Därför är exekveringstiden för ComputeDepth$(T, T.\text{root}())$ lika med den sammanlagda tiden det tar att exekvera den icke-rekursiva delen av ComputeDepth för varje nod i T. Det blir totalt $\sum_{v \in T} O(c_v) = O(\sum_{v \in T} c_v)$, vilket är $O(n)$. En alternativ lösning till det här problemet behöver en extra datastruktur som kan göra insättningar och borttagningar i $O(1)$ tid (t.ex. stackar och köer). Följande är en algoritm som använder en stack för att beräkna djupet. För att få till en algoritm som använder en kö får vi starta med en tom kö och ersätta alla push- och pop-operationer med anrop till enqueue respektive dequeue.

Algorithm 0.3: ComputeDepth(T)

```
current ← T.root()
S ← en tom Stack
S.push(current)
while !S.isEmpty()
  do { current ← S.pop()
     { if T.isRoot(current))
     {   then setDepth(current, 0)
     {   else setDepth(v, 1 + getDepth(T.parent(v)))
     { children ← T.children(v)
     { while children.hasNext())
     {   do S.push(children.next())
```

Anledningen till att vi kan använda både stack och kö här är att det inte spelar någon roll i vilken ordning vi behandlar barnen till varje nod, så länge vi behandlar noden själv före alla dess barn. Vi säkerställer detta genom att ta ut en nod och *sedan* sätta in dess barn i vilken extra datastruktur det nu är vi använder. Observera att varje nod sätts in och tas ut exakt en gång. Antalet insättningar är en övre gräns för hur många gånger den yttre **while** -loopen exekveras och antalet uttagningar begränsar antalet iterationer i den inre **while** -loopen. Allting annat är enkla operationer och metoder som tar $O(1)$ tid, så den här algoritmen går också i $O(n)$ tid.

12. (a) För att illustrera lösningsprincipen ger vi först en algoritm baserad på antagandet att elementen har en av två färger: röd eller blå. Vi håller reda på två index i arrayen $front$ och *end*. $front$ och *end* rör sig mot varandra till de möts och då terminerar algoritmen. Medan de traverserar sekvensen gör vi följande: så fort $front$ ser ett rött element och *end* ser ett blått element byter vi plats på dessa element. Det betyder att alla platser i

arrayen $front$ traverserar kommer att innehålla blåa element, medan alla element *end* traverserar kommer att innehålla röda element.

Algorithm 0.4: RedBlueSort(S)

$n \leftarrow S$.size
$front \leftarrow 0$
$end \leftarrow n - 1$
while $front < end$
do
- **if** $S[front]$.color() $= red$ **and** $S[end].color() = blue$
  - **then**
    - **comment:** swap the two elements
    - $tmp \leftarrow S[front]$
    - $S[front] \leftarrow S[end]$
    - $S[end] \leftarrow tmp$
- **comment:** find the leftmost red element if such exists
- **while** $S[front]$.color() $= blue$ **and** $front < end$
  - **do** $front \leftarrow front + 1$
- **comment:** find the rightmost blue element if such exists
- **while** $S[front]$.color() $= red$ **and** $front < end$
  - **do** $end \leftarrow end - 1$

Algoritmen är in-place och eftersom $front$ och *end* rör sig mot varandra besöks varje element i sekvensen av någon av dem exakt en gång. Varje gång vi inkrementerar $front$ eller dekrementerar *end* utförs som mest en swap-operation och som mest tre kontroller (färgkontroll och möjligtvis två kontroller av $front < end$). Alla dessa tar konstant tid att utföra. Därför är körtiden för RedBlueSort $O(n)$.

Algoritmen för fallet med k färger är baserad på ovanstående algoritm. k gånger gör vi följande:

- Vi håller reda på $right\_boundary$ för S (initialiserad till n).
- Vi väljer en av k färger, $color_i$ som vi inte valt förut.
- Vi sorterar arrayen S precis som ovan förutom att vi använder $color_i$ som den röda färgen och alla färger som inte är $color_i$ som den blåa färgen. Dessutom sorterar vi endast element i S som finns mellan 0 och (inte inklusive) $right\_boundary$.
- Efter att vi är klara sätter vi $right\_boundary$ till det värde indexen $front$ och *end* konvergerat till.

Observera att en iteration av den här algoritmen har exekveringstid $O(n)$ eftersom det, i princip, är samma algoritm som för fallet med två färger och den körs k gånger. Därför blir den totala exekveringstiden $O(nk)$.

Algorithm 0.5: SINGLECOLORSORT($S, right\_boundary, currcolor$)

$front \leftarrow 0$
$end \leftarrow right\_boundary - 1$
while $front < end$
do
- **if** $S[front]$.color() $=$ $currcolor$ **and** $S[end].color() \neq currcolor$
  - **then**
    - **comment:** swap the two elements
    - $tmp \leftarrow S[front]$
    - $S[front] \leftarrow S[end]$
    - $S[end] \leftarrow tmp$
- **comment:** find the leftmost $currcolor$ element if such exists
- **while** $S[front]$.color() $\neq$ $currcolor$ **and** $front < end$
  - **do** $front \leftarrow front + 1$
- **comment:** find the rightmost non-$currcolor$ element if such exists
- **while** $S[front]$.color() $=$ red **and** $front < end$
  - **do** $end \leftarrow end - 1$

if $S[front]$.color() $=$ $currcolor$
then return $(front)$
else return (end)

Algorithm 0.6: COLORSORT(S, C)

$right\_boundary \leftarrow S$.size()
for $colorind \leftarrow 0$ **to** C.size() $- 1$
do
- $currcolor \leftarrow C[colorind]$
- $right\_boundary \leftarrow$ SINGLECOLORSORT($S, right\_boundary, currcolor$)

(b) Låt k vara ett heltal i intervallet $[0, n^2 - 1]$. Om vi representerar talet i bas n skulle det bestå av två tal (l, f), sådana att $k = l \cdot n + f$, där l och f båda är från intervallet $[0, n-1]$. Lämpligt nog är den lexikografiska ordningen på bas n-representationen precis samma som ordningen för heltalen vi vill sortera. Därför kan vi använda Radix-sort för att sortera sekvensen i linjär tid.

Algorithm 0.7: LIMITEDSORT(S, n)

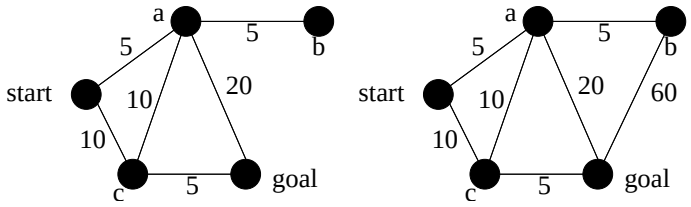
comment: change the representation
for $i \leftarrow 0$ **to** $n - 1$
do $A[i] \leftarrow ((S[i] - S[i] \bmod n)/n, S[i] \bmod n)$
RADIXSORT(A)
comment: restore the representation
for $i \leftarrow 0$ **to** $n - 1$
do $S[i] \leftarrow A[i][0] \cdot n + A[i][1]$

13. Det här kan misslyckas av två skäl. För det första är det inte säkert att strategin hittar någon stig alls, eftersom den inte har någon mekanism för att backa om den hamnar i situationen att alla utgående bågar från noden den befinner sig i leder till noder som redan besökts. I figuren nedan till vänster besöks noderna i ordningen *start*, a och b. Väl i b har alla grannar till b redan besökts, så algoritmen fallerar vid steg iv.

    För det andra kan strategin misslyckas om det finns tre noder v_i, v_j, v_k där bågvikterna mellan dem uppfyller följande egenskaper:

- $w((v_i, v_j)) < w((v_i, v_k))$ (så stigen algoritmen väljer mellan v_i och v_k går genom v_j)
- $w((v_i, v_j)) + w((v_j, v_k)) \geq w((v_i, v_k))$ (triangelolikheten)

Figuren nedan till höger visar ett exempel på detta — noderna besöks i ordningen *start*, *a*, *b*, *goal* med 70 som total väglängd. Den kortaste vägen, däremot, är *start* → *c* → *goal*, med längd 15. Att bågvikterna uppfyller triangelolikheten är rätt så vanligt, särskilt om de representerar ett faktiskt fysiskt avstånd mellan noderna.



14. (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.

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

16. (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 $O(\log n)$.

17. 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];
}
```

18. Om G är en oriktad enkel graf med $|V(G)| \geq 2$ så finns det två noder med samma antal grannar i grafen.

    Antag att G ej har isolerad nod. Detta medför att varje nod har $[1, n-1]$ grannar. Men det finns n noder, så då måste två noder ha samma antal grannar enligt duvslagsprincipen. Om G har en isolerad nod finns det $n-1$ noder med $[1, n-2]$ grannar...

19. (a) **Sant**. Använd räknesortering.

    (b) **Falskt**. Antag motsatsen. Då växer $n^{\varepsilon/\sqrt{\log n}}$ långsammare än $\log n$. Logaritmera $\Rightarrow$ $\log n \cdot \varepsilon/\sqrt{\log n}$ växer långsammare än $\log \log n$. Men $\log n \cdot \varepsilon/\sqrt{\log n} = \varepsilon \cdot \sqrt{\log n}$. Om vi sätter $L = \log n$ får vi att $\varepsilon\sqrt{L}$ växer långsammare än $\log L$ eller, ekvivalent, att εL växer långsammare än $\log^2 L$, men $\log^2 L$ växer strikt långsammare än L, så vårt ursprungliga antagande måste vara falskt.

20. (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.

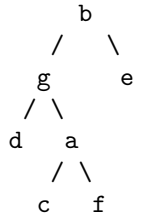
21. (a)

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.

22. (a) Algoritmen beräknar antalet interna noder i trädet.

    (b)



    (c) (E)

23. 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.

24. 1:a, 2:d, 3:e, 4:c, 5:f, 6:b

25. K M R

26.
- `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;
}
```

27. 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.

28. Endast I och II. Anropsstacken innehåller en sekvens av noder på en riktad stig från s till nuvarande nod (med s på botten och nuvarande nod på toppen).

29. (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.

30. (a) Vi använder två stackar. En stack S används för att hålla reda på de `push` och `pop` som görs. Den andra stacken M håller reda på det minsta elementet. För att implementera `push` gör vi `push` på S. Om det nya elementet är mindre än översta element på M gör vi också `push` på M. För att implementera `pop` gör vi `pop` på S. Om elementet som `pop`:as är samma som det översta elementet på M `pop`:ar vi M också. `findMin` görs genom att undersöka översta elementet på M. All dessa operationer går uppenbarligen att utföra i tid $O(1)$.

(b) Med dessa fyra operationer går det att sortera en indatasekvens med $O(n)$ operationer. Vi vet att (jämförelsebaserad) sortering tar tid $\Omega(n \log n)$ i värsta fallet vilket ger att minst en av operationerna måste använda $\Omega(\log n)$ tid.

31. Sant.

32. 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.