TENTAMEN – Lösningsförslag

Algoritmer och datastrukturer

Uppgift 1, grundläggande: Komplexitet & korrekthet

(A) ”Ge en så snäv värstafallskomplexitet för `match1()` som möjligt. Använd s för längden på arrayen som söks i och p för längden på arrayen som söks efter. Motivera ditt svar.”

Lösningsförslag:

Den yttre loopen löper som mest s gånger och i för varje steg i den yttre loopen löper den inre loopen som mest p gånger.

Vi kan använda oss av det faktum att den yttre antal steg kan beskrivas snävast i form av $\max(0, s - p)$.

Snävast blir då $O(\max(1, p(s - p))$ men $O(sp)$ är också tillräckligt snävt för att vara godkänt.

(B) ”Visa på ett fall där `match2()` inte ger samma svar som `match1()`.”

```
match1(”112”, ”12”) == 1
match2(”112”, ”12”) == -1 // Erronously skips to ’2’ after not matching on ’11’.
```

Uppgift 2, grundläggande: Hashtabeller & funktioner

Exempel för en tabell med m = 10, k = 3 och följande nycklar:

```
H:1    B:9    G:4    A:2    C:3    E:2    D:3      F:2
```

```
0:
1: H-1
2: A-1 E-1 F-1
3: C-1 D-1
4: G-1
5: E-2 F-2
6: D-2
7:
8: F-3
9: B-1
```

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 3, grundläggande: Prioritetsköer

(A) Siffrorna du fick för `add()`-operationerna.

(B) De tre sekvenserna a), b) och c)

(C) För varje sekvens, a), b) och c):
Om sekvensen kan skrivas ut, ge en följd av av `add()` och `removeMin()`-operationer som ger utskriften.
Om sekvensen inte kan skrivas ut, förklara varför utskriften inte kan uppstå.

Lösningsförslag:

Låt A(...) vara add-funktionen och R() vara removeMin+print-funktionen.
Antag följande:

```
Insättningsordning: 6 3 1 0 2 4 6

a) 6 0 1 2 4 3 6

b) 0 1 2 3 4 6 6

c) 6 0 4 1 2 3 6
```

Vi försöker identifiera de kortaste sekvenserna som kan ge utdatat i fråga. Om vi hittar en motsägelse avbryter vi.

a) `6 0 1 2 4 3 6`

A(6) R() -> 6
För att kunna få ut 0 så måste vi först utföra A(3, 1, 0)
R() -> 0
R() -> 1
För att kunna få ut 2 så måste vi först lägga till A(2)
R() -> 2
För att kunna få ut 4 så måste vi först lägga till A(4)
R() -> 3 ?! Motsägelse. Alltså är sekvensen omöjlig.

b) `0 1 2 3 4 6 6` är lite olycksfall i arbetet från min sida; strikt stigande sekvenser kan ju trivialt fås genom A(...) följt av lika många R().

c) `6 0 4 1 2 3 6`

A(6) R() -> 6
För att kunna få ut 0 så måste vi först utföra A(3, 1, 0)
R() -> 0
För att kunna få ut 4 så måste vi först utföra A(2, 4)
R() -> 1 ?! Motsägelse. Alltså är sekvensen omöjlig.

Uppgift 4, grundläggande: Grafer

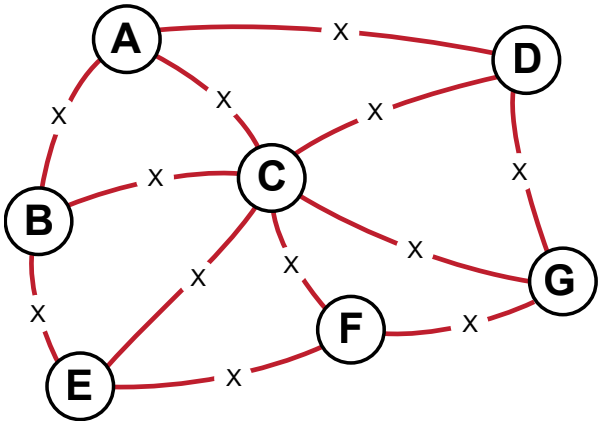
Antag R = 19. Vi löser uppgiften i flera steg.

Steg 1

Rita en graf som har 7 noder och där alla noder har minst tre kanter till andra noder än sig själv.

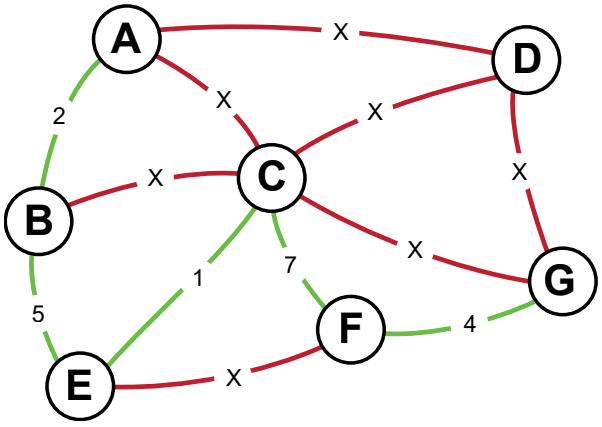
Steg 2

Välj en trevlig väg från A till G men som inte går genom *alla* noder; bivillkortet att vikten hos MST ska vara R + 10 kommer annars inte gå att uppfylla. Sätt ut vikter så att vägen får vikt R. (Man kunde här valt en enklare lösning där vägen A–G går genom färre noder.)

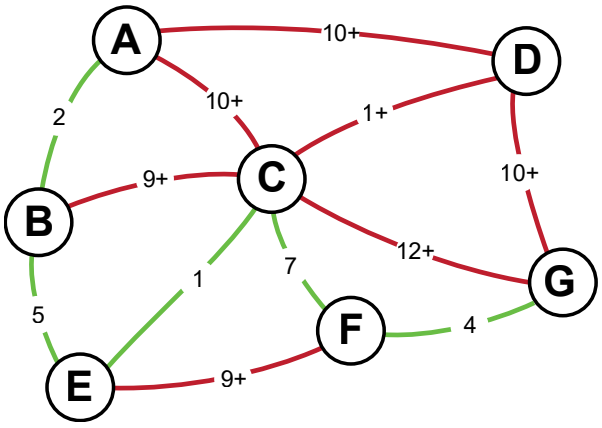


Steg 3

Förvissa oss om att det inte finns någon kortare väg A–G. De vägar som i antalet kanter är kortare är ganska många. Om vi börjar med att fokusera på de kortaste vägarna, i kanter räknat, så ser vi att AC + CG måste ha (minst) vikten 20. Likaså AD + DG.
Vi sätter AC, CG, AD, DG till minst 10.
EF > 8, annars kunde vi gått EF istället för ECF (vikt 8).
BC > 8, annars kunde vi gått BC istället för BEC (vikt 8).
CG > 11, annars kunde vi gått CG istället för CFG (vikt 11).
CD > 1, annars kunde vi gått CDG istället för CFG (vikt 11).
Nu har vi skapat en kandidatgraf där vi vet att kortaste vägen A–G är 19; alla andra vägar är måste vara längre.

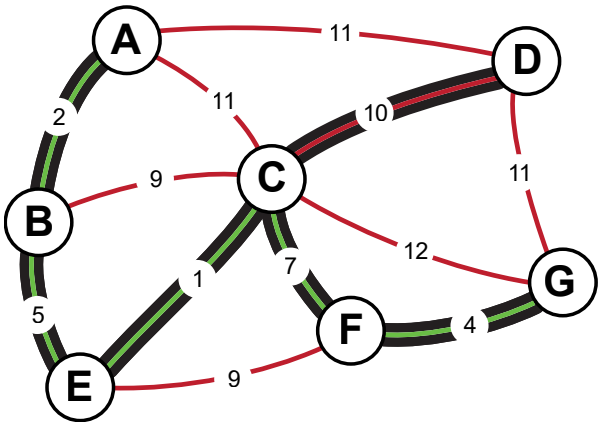


Vikterna på kanter som vi är fria att manipulera uppåt markerar vi också med ett +. Så länge vi bara justerar dem uppåt så är vikten för kortaste vägen oförändrad.



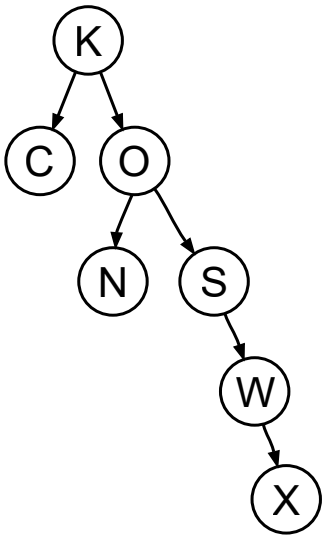
Steg 4

Återstår att välja ett antal kanter som vi vill ska ingå i vårat MST och sedan justera kanterna. Den enda nod som inte är inkluderad i vår SP är D, vår SP är ju också ett MST för de noder som är inkluderade i det.
Vi sätter CD till 10. Återstår att fixera de justerbara vikterna så att det inte kan finnas ett MST med vikt < 29. Vi kan sätta alla vikter 10+ till 11 och CG till 12. Därmed är vikten för vårt MST också 29 och vi är klara.



Uppgift 5, grundläggande: Binära sökträd

Utgångspunkten är trädet till höger.



(A) Visa tre olika insättningssekvenser som ger trädet i fråga.

(B) Visa 1 insättningssekvens som *inte* hade kunnat skapa trädet. Första elementet i ditt motexempel måste vara samma element som rotnoden i ditt träd. Om trädet t.ex. har ”P” som rot måste din sekvens alltså lyda ”P ...”

(C) Visa hur ditt träd ser ut efter att du tagit bort det element som utgör rotnoden. Förklara hur du går tillväga och rita det nya trädet.

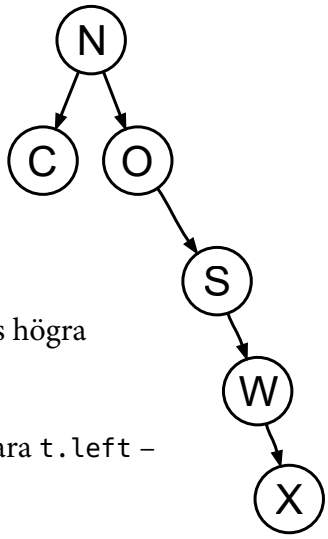
Lösningsförslag

(A) Det finns minst två enkla strategier här:
1. Traversera uppifrån och ned från vänster till höger; `{K C O N S W X}`
Om vi istället går från höger till vänster kan vi få `{K O C S N W X}`
Båda dessa varianter innebär att vi lägger till element genom BFS.
2. Traversera via DFS; gå nedåt i samma riktning tills vi når ett löv. Backa sedan uppåt.
Om vi prioriterar att gå åt vänster så får vi `{K C O N S W X}`
Om vi prioriterar att gå åt höger så får vi istället `{K O S W X N C}`

(B) `K` måste finnas med som första steg. En enkel strategi är sedan att sätta in elementen med start nedifrån i trädet. Exempel: `{K X W S N O C}`

(C) Om `K` bara hade haft ett barn så kunde vi ersatt `K` med dess enda barn. Det finns flera sätt att lösa problemet då vi tar bort en nod med 2 barn i ett BST. Metoden vi använder här finns i S&W som ”Hibbard deletion”, sid 410–411.
Grundidén är att vi ersätter en borttagen nod med dess *efterföljande element* (successor).

Hibbard-borttagning i trädet uppe till höger:



(1) Spara en länk `t` till noden som ska tas bort (`K`).

(2) Låt `x` vara `min(K.right)` (det minsta elementet till höger om `K`), det minsta elementet större än `K` är i vårt fall `N`.

(3) Sätt `x.right` till `deleteMin(t.right)`, som returnerar noden som utgör `t`:s högra barn – alltså `N.right = O`.

(4) Låt `x.left` (som måste vara null: det var ju det minsta värdet och ett löv) vara `t.left` – alltså `N.left = C`.

Resultatet syns till höger.

Uppgift 6, grundläggande: Sortering

Skapa en ”pessimal” sekvens (motsatsen till optimal) sådan att quicksortimplementationen som gavs utför maximalt antal steg.

(A) Din lista med tal.

(B) Din lista med tal ordnade så att quicksort med mittersta elementet som pivot-strategi kommer att partitionera så obalanserat som möjligt.

(C) Varje steg i partitioneringen (se illustration ovan).

Lösningsförslag

Flest antal operationer med quicksort kommer vi att få då partitioneringen är maximalt ojämn. Idealiskt skulle vi vilja åstadkomma partitioner där antingen höger eller vänster sida om pivotelementet alltid är tom.

Utgångspunkten är följande tal: `{46 19 98 87 66 95}`

Vi underlättar för oss genom att först sortera listan: `{19 46 66 87 95 98}`

Antag att varje element i listan är en variabel: `{A B C D E F}`

Pivot är `C` som då måste vara lägsta värdet, `C` = 19. Nu kan vi partitionera (byta plats på `A` och `B`):
`{A B 19C D E F} -> {19C B A D E F}`

Ny pivot är `D` = 46. Partitionera, d.v.s. byt plats på `B` och `D`: `{19C B A 46D E F} -> {19C 46D A B E F}`

Ny pivot är `B` = 66. Partitionera: `{19C 46D A 66B E F} -> {19C 46D 66B A E F}`

Ny pivot, `E` = 87: `{19C 46D 66B A 87E F} -> {19C 46D 66B 87E A F}`

Ny pivot, `A` = 95: `{19C 46D 66B 87E 95A F} -> {19C 46D 66B 87E 95A F}`

Ny pivot, `F` = 98: `{19C 46D 66B 87E 95A 98F} -> {19C 46D 66B 87E 95A 98F}`

dvs: `{A B C D E F} = {95 66 19 46 87 98} (med tom högerdel: {46 87 98 66 19 95})`

Referens

Om vi istället hade haft elementen optimalt ordnade, så att samma pivotstrategi skulle ge så lågt antal operationer som möjligt hade det istället kunnat se ut så här:

1: `98 95 **87** 66 46 19 -> 19 46 66 87 95 98`

2: `19 **46** 66 -> 19 46 66 (vänstra delen av första partitionen)`

3: `**19** -> 19`

4: `**66** -> 66`

5: `**95** 98 -> 95 98 (högra delen av första partitionen)`

6: `**98** -> 98`

Antalet steg blev lika många, 6, men antalet operationer i varje steg blev färre.

Uppgift 7, avancerad: Komplexitet

```
// Antag att 0 < m < n
verySlow(n, m)
    if n ≤ m
        return 1
    else
        return verySlow(n - 1, m) + verySlow(n - m, m)
```

”Formulera om funktionen `verySlow()` så att du istället får funktionen `ratherFast()`.
Tidskomplexiteten för `ratherFast()` ska som sämst vara *O(n)*.
För alla $0 < m < n$ så ska `ratherFast(n, m)` ge samma värde som `verySlow(n, m)`.

Använd som mest använd *O(m)* minne, inklusive anropsstack för full poäng.”

Lösningsförslag:

Problemet med funktionen som den är given är att varje anrop potentiellt resulterar i ytterligare två anrop som i sin tur resulteterar i ytterligare två, ... , tills fallet $n \leq m$ nås.

Genom att lagra de senaste m relevanta värdena i en cirkulär buffert så kan vi uppnå linjär tid i $O(n)$ och linjär mängd minne i $O(m)$. Om vi inte hade använt oss av en cirkulär buffert så hade minnesåtgången istället hamnat i $O(n)$.

```
// Helper function for the circular buffer.
m(prevs, i, m)
    if i - 1 ≤ m
        t1 = 1
    else
        t1 = prevs[(i - 1) % length(prevs)]
    if i - m ≤ m
        t2 = 1
    else
        t2 = prevs[(i - m) % length(prevs)]

    return t1 + t2
```

```
ratherFast(n, m)
    if(n ≤ m)
        return 1
    else
        prevs = new array of size m.
        for i = {0, ..., n}
            prevs[i % m] = m(prevs, i, m)

        return prevs[(m + n) % m]
```

Det existerar också en lösning som går i tid $O(m^3 \log n)$ och minne $O(m)$ men den är markant trassligare.

Uppgift 8, avancerad: Felsökning

Ett stort program, K, har helt plötsligt börjat krascha för vissa indata.
Indata till programmet kan beskrivas som $X = \{x_1, x_2, ..., x_n\}$.

Beskriv en algoritm (tydlig pseudokod) som *löser problemet korrekt* samt beskriv algoritmens komplexitet.

Lösningsförslag:

Vi byter index så att första elementet har index 0 istället för 1 (likadant som i Java, C, ...).

Vi inför en hjälpfunktion $H(K, X, a, b)$ som är sann om K kraschar med indata från position a till och med b i X, annars falsk.

Här är det lätt hänt att man gör diverse ”off-by-one”-fel men har man i stora drag gjort/beskrivit något som går att översätta till exemplen nedan så är det ok.

Troligen är lösningen till höger (asymptotiskt) optimal. Själva idén är enkel:
Gör först en binärsökning för att hitta a (början), sen likadant för att hitta b (slutet).
Maximalt antal operationer (inträffar då $a = 0$, $b = n - 1$) blir $\sim n + 2\ (n \lg n) = O(n \log n)$.

```
// O(n^2)-solution:
findCrash(K, X)
   s = length(X)
   if not H(K, X, 0, s)
      return {-1, -1}

   // Find a
   for i in {0, ..., s - 1}
      if not H(K, X, i, s)
      // For all i less than current i,
      // there was a crash =>
      // a must be the previous index.
         a = i - 1
         break

   // Find b, note that i is iterated in
   // decreasing order.
   for i in {s - 1, ..., a}
      if not H(K, X, a, i}
      // For all i greater than the
      // current i, there was a crash =>
      // b must be the previous index.
         b = i + 1
         break

   return {a, b}
```

```
// O(n log n)-solution
findCrash(K, X)
   s = length(X)
   if not H(K, X, 0, s)
      return {-1, -1}

   minA = 0, maxA = s
   while(minA < maxA)
      mid = ceil((minA + maxA) / 2)
      if H(K, X, mid, s)
         minA = mid
      else
      // Since the program didn't crash
      // when we didn't include any
      // elements before mid, a can at
      // most be mid - 1
         maxA = mid - 1

   a = minA
   minB = a, maxB = s
   while(minB < maxB)
      mid = floor((minB + maxB) / 2)
      if H(K, X, a, mid)
         maxB = mid
      else
         minB = mid + 1

   b = maxB
   return {a, b}
```

Uppgift 9, avancerad:

Lösningsförslag:

(A) och (B), se nästa sida.

Låt e beteckna antalet kanter i grafen och v antalet noder.

Vi väljer följande datastrukturer (inte nödvändigtvis optimala) för N, V och G:

N kan vara en stack. Det spelar ingen roll vilken ordning vi tar bort element från N. $O(1)$ tid, $O(e)$ minne.

V kan vara en hashtabell för genomsnittlig tidskomplexitet för uppslagning och insättning $O(1)$, $O(v)$ minne.

G kan representeras som en symboltabell (hashtabell) av noder där varje nod lagrar både inkommande och utgående kanter. Att ta fram en nod är i O(1). Om vi snabbt vill kunna utföra operationen ”if b has no unvisited incoming edges” effektivt så behöver vi antingen förstöra G eller skapa en kopia av G:s noder. Givet en kopia, G’. Här finns det många valmöjligheter och avvägningar, beroende på det förväntade förhållandet e/v, men vi nöjer oss med det faktum att vi kan ta bort kanter samt se hur många kanter en nod har i konstant tid.

Då våra antaganden m.a.p. de ingående datastrukturerna är formulerade kan vi räkna ut hela algoritmens komplexitet:

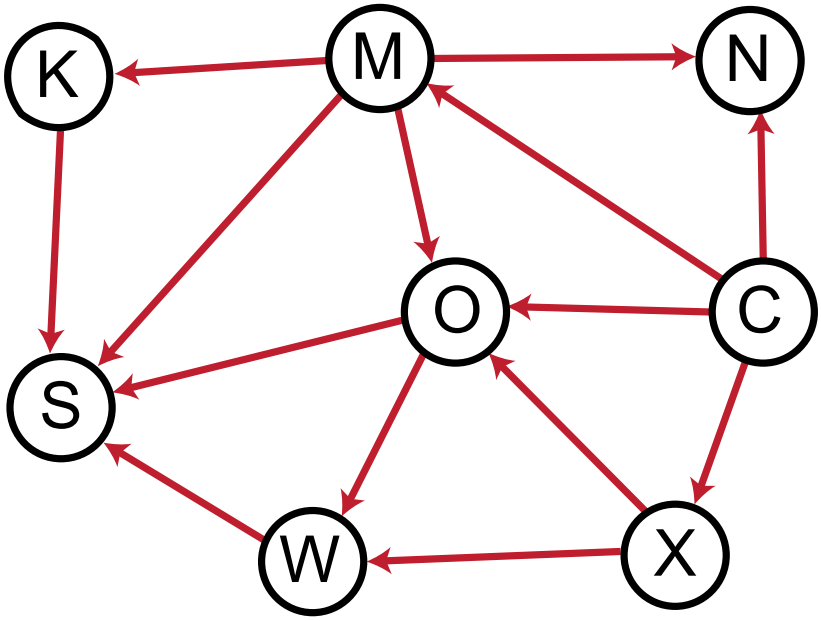
```
mystery(G)
   // Constant time, allocation.
   P ← Empty list
   // Loop over the nodes in G - O(v) - and for each node without incoming edges,
   // checking is in O(1), add to N - O(1).
   N ← The set of nodes without incoming edges in G
   // Constant time, allocation.
   V ← Empty set, to track of visited edges.
   // We also utilize G' to quickly update number of incoming and outgoing edges.
   // Copying G to G' takes O(e + v) time.

   while not empty(N)        // Can be true at most v times, O(v).
      a ← deleteRandomNode(N) // Remove from stack, O(1)
      Push a to the end of P // Constant time, O(1)
      for every non-visited edge e from a to b // At most O(e) times
         Mark e as visited.  // Update G', O(1), and V, also O(1)
         if b has no unvisited incoming edges left then // O(1)
            add b to N   // O(1)

   return P // O(1)
end mystery
```

Den sammantagna komplexiteten bli $O(v + e)$.

Anledningen till att den inte blir $O(ve)$, som man ju kan ledas till att tro i och med de nästlade looparna, är att antalet obesökta kanter hela tiden minskar.

Algoritmen genererar en topologisk sortering och finns beskriven här.

Tabellen nedan visar en körning för grafen till höger. Blanka celler betyder ”oförändrad”, ”-” betyder ”blivit tom”. För P och V, som bara byggs på, så indikerar ”+” att ett element lagts till.



Steg	P	N	V
Before while		C	-
N not empty	+C	-	
Visit CN			+CN
Visit CX		X	+CX
Visit CO			+CO
Visit CM		X M	+CM
Visit empty	+X	M	
Visit XO			+XO
Visit XW			+XW
N not empty	+M	-	
Visit MN		N	+MN
Visit MK		N K	+MK
Visit MS			+MS
Visit MO		N K O	+MO
N not empty	+O	N K	
Visit OW		N K W	+OW
Visit OS			+OS
N not empty	+W	N K	
Visit WS			+WS
N not empty	+K	-	
Visit KS		S	+KS
N not empty	+S	-	
Done. return {C X M O K W S			