Lösningsförslag 1

Uppgift 1.

Algoritm Antag att indata är $x_1, \ldots, x_n$ och anropa följande algoritm med $f(1, n)$.

algorithm $f(l, r)$
$k := \lfloor \frac{l+r}{2} \rfloor$
if $x_k < x_r$ **then return** $f(l, k)$
elsif $x_l < x_k$ **then return** $f(k, r)$
else return r

Korrekthet. Vi visar att $f(l, r)$ returnerar det minsta talet mellan index l och r. Detta implicerar att algoritmen ger korrekt svar för $x_1, \ldots, x_n$ då den anropas med $f(1, n)$. Observation: Om $x_1, \ldots, x_n$ är cykliskt sorterad så är $x_l, \ldots, x_r$ cykliskt sorterad för godtyckliga $1 \leq l \leq r \leq n$.

Induktion över $k = \lfloor \frac{l+r}{2} \rfloor$

Bassteg. Om $k = 1$ så är antingen $l = r = 1$ eller $l = 1$ och $r = 2$. Om $l = r = 1$ så returnerar algoritmen r vilket är korrekt. Om $l = 1$ och $r = 2$ så testas först om $x_1 < x_2$ och isåfall returneras $f(1, 1) = 1$ vilket är korrekt. Därefter testas om $x_1 < x_1$ vilket inte kan vara fallet. Om sista fallet nås så gäller att $x_r < x_l$, dvs $r_2 < r_1$, och 2 returnernas vilket är korrekt.

Ind. ant. Antag att algoritmen är korrekt då $k \leq p$, $p \geq 0$.

Ind. steg. Vi visar att algoritmen är korrekt på $k = p + 1$. Vi betraktar tre fall.

Fall 1. Om $x_k < x_r$ så måste det minsta talet i $x_l, \ldots, x_r$ finnas i intervallet $x_l, \ldots, x_k$ eftersom $x_l, \ldots, x_r$ är cykliskt sorterad. Vi ser att

$$\left\lfloor \frac{l+k}{2} \right\rfloor \leq \left\lfloor \frac{(k-1)+k}{2} \right\rfloor < k = p + 1$$

så induktionsantagandet garanterar att algoritmen ger korrekt svar i detta fall.

Fall 2. Om $x_l < x_k$ så måste det minsta talet i $x_l, \ldots, x_r$ finnas i intervallet $x_k, \ldots, x_r$. Beviset är analogt med föregående fall.

Fall 3. Om varken fall 1 eller 2 är applicerbart så returnerar algoritmen r. Detta är korrekt eftersom i detta fall gäller $x_l \geq x_k \geq x_r$ och x_r är det minsta talet i $x_l, \ldots, x_r$.

Tidskomplexitet. Notera att algoritmens tidskomplexitet beskrivs av $T(1) = 1$ och $T(n) = T(n/2) + c$. Detta ger att

$$T(n) = T(n/2) + c + T(n/4) + c + T(n/8) + c + \cdots + 1 =$$

$$c \cdot \log_2(n) + 1 \in O(\log n)$$

Uppgift 2.

Algoritmen baseras på dynamisk programmering. Bilda en $n \times m$-tabell M med hetal. Låt $M[i, j]$ beteckna antalet tecken till vänster om (och inklusive) a_i som överensstämmer med lika många tecken till vänster (och inklusive) b_j. Längden av den längsta gemensamma strängen är då det största talet i M. M kan beräknas på följande sätt:

$M[i, j] = 0$ om $i = 0$ eller $j = 0$

$M[i, j] = M[i - 1, j - 1] + 1$ om $a_i = b_j$

$M[i, j] = 0$ annars.

En fullständig algoritm kan se ut så här.

for $i = 0$ **to** n
  $M[i, 0] := 0$
for $j = 0$ **to** m
  $M[0, j] := 0$
for $i := 1$ **to** n
  for $j := 1$ **to** m
    if $a_i = b_j$ **then** $M[i, j] := M[i - 1, j - 1] + 1$ **else** $M[i, j] := 0$
$max := 0$
for $i := 1$ **to** n
  for $j := 1$ **to** m
    if $M[i, j] > max$ **then** $max := M[i, j]$
return max

Det framgår från algoritmen att den går i $O(n + m + nm + nm) = O(nm)$ tid.

Uppgift 3.

Algoritm. Bilda en en-dimensionell array A som innehåller alla element i S_1 och alla element i S_2. Sortera A med Mergesort. Om det finns en position i sådan att $A[i] = A[i + 1]$ så svara ”nej” och svara i annat fall ”ja”.

Tidskomplexitet. Arrayen A innehåller $n = |S_1| + |S_n|$ element så sorteringen tar $O(n \log n)$ tid. Den avslutande kontrollen tar $O(n)$ tid så hela algoritmen går i $O(n \log n)$ tid.

Korrekthet. Det följer direkt från algoritmen att den svarar "nej" om och endast om $S_1 \cap S_2 \neq \emptyset$.

Uppgift 4.

Algoritm. Tag bort den billigaste bågen i grafen G tills grafen inte längre är sammanhängande. Lägg tillbaka den sista bågen i grafen. Kalla denna graf H. Beräkna ett minsta uppspännande träd T för H och returnera T.

Tidskomplexitet. Algoritmen går uppenbart i polynomisk tid eftersom det minsta uppspännande trädet kan beräknas i polynomisk tid med, exempelvis, Prim eller Kruskals algoritm.

Korrekthet. Antag att det finns en graf G som algoritmen inte lyckas lösa optimalt. Låt T vara det träd algoritmen returnerar och låt U vara ett optimalt träd. Låt e_T vara den billigaste bågen i T och låt e_U vara billigaste bågen i U.

Vi vet att $c(e_T) < c(e_U)$ eftersom T inte är en optimal lösning. Eftersom e_U är den billigaste bågen i U så innehåller U inte någon båge som är billigare än e_U. Speciellt innehåller U inte bågen e_T. Notera att detta innebär att U inte är sammanhängande och sålunda inte är ett uppspännande träd. Motsägelse och vi har visat att algoritmen alltid producerar en optimal lösning.