Algoritmer, datastrukturer och komplexitet, hösten 2020
Lösningsförslag till mästarprov 1: algoritmer

1. **Dator med extra bitoperationer**
   Vi löser problemet med dekomposition. Vektorn delas upp i två halvor och rekursiva anrop görs när det finns minst en etta i den halvan.

```
// NumberOfOnes returnerar antalet ettor i bitvektorn v.
func NumberOfOnes(v):
    n = v.len
    if n == 0
        return 0
    if n == 1
        if v[0] == 0
            return 0
        else
            return 1
    sum = 0
    mid = 1 + (n-1)/2
    if !IsZero(v, 0, mid-1)
        sum += NumberOfOnes(v[:mid])
    if !IsZero(v, mid, n-1)
        sum += NumberOfOnes(v[mid:])
    return sum
```

   Tidskomplexitet

   Vi kan använda anropet till funktionen NumberOfOnes som karakteristisk operation när vi beräknar tidskomplexitet – alla övriga operationer i funktionen tar nämligen $O(1)$ tid att beräkna.

   Varje etta i vektorn motsvarar ett av löven i anropsträdet, och för att nå ett löv krävs det $O(\log n)$ anrop – vektorns storlek halveras nämligen vid varje anrop. Det totala antalet funktionsanrop blir därför $O(1 + k \log n)$.

   Korrekthet

   Algoritmen är korrekt om den uppfyller sin specifikation, dvs för varje bitvektor v ska den returnera antalet ettor i bitvektorn. Man kan bevisa korrektheten av algoritmen med ett induktionsbevis. Basfallet visar man genom att titta på koden där n är noll eller ett. Induktionsantagandet innebär att funktionen returnerar det korrekta värdet för alla vektorer kortare än n, $n \geq 2$. Då återstår bara att kontrollera att de rekursiva anropen sker på vektorer av korrekt längd och att summan beräknas korrekt.

   Alternativ lösning

   Man kan också lösa uppgiften med hjälp av upprepad binärsökning. Man kan till exempel börja med att binärsöka efter den första ettan i vektorn. Detta tar $O(\log n)$ tid. Därefter nollställer man denna etta och fortsätter på samma sätt med den första ettan som återstår. Det blir då k sökningar, som var och en tar $O(\log n)$ tid.

2. **Trött robot**

Indata är ett naturligt tal n och en heltalsmatris Os[1..n,1..n] som för varje position anger hur många gram osmium det finns i motsvarande ruta.

Om roboten inte hade kunnat backa så hade problemet varit samma som triangelstigsproblemet i förberedelseuppgifterna till föreläsning 10, fast med två trianglar som bildar en kvadrat. Triangelstigsproblemet löstes med dynamisk programmering. Låt oss därför försöka lösa vårt problem med dynamisk programmering med en liknande rekursion.

Utvidga matrisen med en ram av element som är $-\infty$, dvs på rader och kolumner med index $n+1$. Detta gör att rekursionen blir enklare att formulera, eftersom vi inte behöver specialbehandla kanten av rutnätet då.

Låt delproblemen vara den maximala mängden osmium som kan samlas in från och med ruta (x, y) ner till (n, n) för varje värde på x och y mellan 1 och n. Vi behöver också hålla reda på om roboten har genomfört sin backning eller inte. Vi lagrar delproblemens lösningar i en tredimensionell struktur $S[0..n, 0..n, 0..1]$ där den första dimensionen anger x, andra dimensionen anger y och tredje dimensionen anger antalet backningar som gjorts på vägen från (x, y) till (n, n) (0 eller 1). Roboten börjar inte med att backa, så från (x, y) går den antingen till $(x+1, y)$ eller $(x, y+1)$.

Det värde vi söker är $S[1, 1, 1]$. Basfallet är $S[n, n, 0]$ som är $Os[n, n]$ per definition. Vi kan också sätta $S[n, n, 1]$ till $Os[n, n]$ trots att roboten inte kommer att backa. Det fallet inträffar om roboten inte behöver göra någon backning i en optimal väg från $(1, 1)$ till (n, n). Rad $n+1$ och kolumn $n+1$ i S kan vi fylla med nollor så blir rekursionen enklare att formulera, eftersom vi inte behöver tänka på att vi är i kanten av rutnätet då.

När roboten kommer till rutan (x, y) så hämtar den upp $Os[x, y]$ gram osmium. Sedan går den antingen till $(x+1, y)$ eller $(x, y+1)$. Om roboten sedan inte backar tillbaka till (x, y) så är rekursionen för $S[x, y, b]$ enkel. Vi tar helt enkelt största värdet av $S[x+1, y, b]$ och $S[x, y+1, b]$ och lägger till $Os[x, y]$.

Om roboten gör en backning till (x, y) så måste den ha backat från antingen $(x+1, y)$ eller $(x, y+1)$. Eftersom det inte finns någon vinst med att gå ner till en ruta, backa tillbaka ett steg och sedan gå ner till samma ruta igen så kan vi anta att roboten efter backningen går vidare till den andra möjliga rutan. Det finns därför bara två möjliga fall där roboten efter backningen hamnar i (x, y). Första fallet är att roboten går till $(x, y+1)$ och samlar $Os[x, y+1]$ gram osmium, backar tillbaka till (x, y) och sedan fortsätter till $(x+1, y)$ utan att därefter göra några ytterligare backningar (alltså $S[x+1, y, 0]$)). Andra fallet är att roboten går till $(x+1, y)$ och samlar $Os[x+1, y]$ gram osmium, backar tillbaka till (x, y) och sedan fortsätter till $(x, y+1)$ utan att därefter göra några ytterligare backningar.

Rekursionen blir därför:

$$S[x,y,b] = \begin{cases} 0 & \text{då } x = n+1 \\ 0 & \text{då } y = n+1 \\ Os[n,n] & \text{då } x = y = n \\ \max(S[x+1,y,0], S[x,y+1,0]) + Os[x,y] & \text{då } x \le n,\ y \le n,\ b = 0 \\ \max(S[x+1,y,1], & \\ \quad S[x,y+1,1], & \\ \quad Os[x,y+1] + S[x+1,y,0], & \text{då } x \le n,\ y \le n,\ b = 1 \\ \quad Os[x+1,y] + S[x,y+1,0]) + Os[x,y] & \end{cases}$$

Beräkningsordning

Först beräknar vi hela S för $b = 0$ och därefter hela S för $b = 1$. För varje b-värde beräknar vi matrisen diagonalvis efter ökande avstånd från den understa punkten i rutnätet, alltså det nedre högra hörnet i S, dvs efter avtagande värde på $x + y$. Då kommer vi alltid att ha

räknat ut varje S-element innan vi behöver använda det för att beräkna ett annat S-element enligt rekursionen.

```
TiredRobot(n, Os[1..n, 1..n]) =
    for i ← 1  to n + 1 do
        Os[i, n + 1] ← −∞
        Os[n + 1, i] ← −∞
        S[i, n + 1, 0] ← S[i, n + 1, 1] ← 0
        S[n + 1, i, 0] ← S[n + 1, i, 1] ← 0
    S[n, n, 0] ← S[n, n, 1] ← Os[n, n]
    assert Basfallen är korrekt beräknade, liksom kolumn n + 1 och rad n + 1 i matriserna
    for xysum ← 2n − 1 downto 2 do
        for x ← 1  to min(n, xysum) − 1 do
            y ← xysum − x
            S[x, y, 0] ← max(S[x + 1, y, 0], S[x, y + 1, 0]) + Os[x, y]
    assert S[x, y, 0] är korrekt beräknade för alla x och y
    for xysum ← 2n − 1 downto 2 do
        for x ← 1  to min(n, xysum) − 1 do
            y ← xysum − x
            S[x, y, 1] ← max(S[x + 1, y, 1], S[x, y + 1, 1],
                             Os[x, y + 1] + S[x + 1, y, 0],
                             Os[x + 1, y] + S[x, y + 1, 0]) + Os[x, y]
    assert S[x, y, 1] är korrekt beräknade för alla x och y
    return S[1, 1, 1]
```

Tidskomplexitet

Den första for-slingan går $O(n)$ varv. Dom nästlade forslingorna går igenom alla rutor (utom en), dvs går n^2 varv var. Inuti varje nästlad forslinga görs bara konstant arbete. Tidskomplexiteten blir därför $O(n^2)$.

Korrekthet

För att motivera korrektheten för algoritmen behöver vi visa att för varje probleminstans (korrekt indata) returnerar algoritmen den maximala mängd osmium som roboten kan plocka upp.

Eftersom det är en dynamisk programmeringsalgoritm ska vi dels visa att rekursionen korrekt beskriver delproblemlösningarna och dels att pseudokoden korrekt implementerar rekursionen och returnerar rätt värde.

Att rekursionen är korrekt har vi motiverat ovan. Vi har också motiverat att beräkningsordningen fungerar. Delproblemet $S[1, 1, 1]$ ger den sökta lösningen, vilket är det som algoritmen returnerar på sista raden i pseudokoden. Det enda som återstår är att visa att algoritmen innan den returnerar har beräknat alla värden för $S[x, y, b]$ enligt rekursionen och att beräkningsordningen följs. Beräkningsordningen säger först att S först ska beräknas för $b = 0$ och därefter för $b = 1$, och för varje b ska $S[x, y, b]$ beräknas diagonalvis efter ökande avstånd från nedre högra hörnet, dvs för minskande värde på summan $x + y$, vilket stämmer med algoritmen. (Vi behöver inte ange invarianter i lösningen till uppgift 2.) Det är enkelt att kontrollera att tilldelningarna till S görs enligt rekursionen. Därmed är korrektheten visad.

Alternativ lösning

Alternativt till ovanstående kan vi låta delproblemen vara den maximala mängden osmium som kan samlas in från startrutan $(1, 1)$ fram till ruta (x, y) för varje värde på x och y mellan 1 och n. Vi lagrar delproblemens lösningar i en tredimensionell struktur $M[0..n, 0..n, 0..1]$ där den tredje dimensionen anger antalet backningar som gjorts fram till ruta (x, y) (0 eller 1).

Det värde vi söker är $M[n, n, 1]$.

När ingen backning görs är rekursionen för $M[x, y, b]$ enkel. I förra steget måste roboten ha varit i antingen $(x-1, y)$ eller $(x, y-1)$. Och när roboten kommer till rutan (x, y) så hämtar den upp $Os[x, y]$ gram osmium. Vi tar därför det största värdet av $M[x-1, y, b]$ och $M[x, y-1, 0]$ och lägger till $Os[x, y]$.

Om roboten har gjort en backning och sedan går ner ett steg och hamnar i (x, y) så måste den ha utgått från antingen $(x-1, y)$ eller $(x, y-1)$, gått ner ett steg, backat tillbaka och sedan gått ner till (x, y). Naturligtvis finns det ingen vinst med att gå ner till (x, y), backa tillbaka ett steg och sedan gå ner till (x, y) igen, så detta kan vi bortse från. Det finns därför bara två möjliga fall där roboten kan ha backat och sedan hamnat i $x, y)$. Första fallet är att roboten kom från $(x-1, y)$, gick ner till $(x-1, y+1)$ och samlade $Os[x-1, y+1]$ gram osmium, backade tillbaka till $(x-1, y)$ och sedan gick till (x, y) och samlade in $Os[x, y]$. Andra fallet är att roboten kom från $(x, y-1)$, gick ner till $(x+1, y-1)$ och samlade $Os[x+1, y-1]$ gram osmium, backade tillbaka till $(x, y-1)$ och sedan gick till (x, y) och samlade in $Os[x, y]$.

Rekursionen blir därför:

$$M[x,y,b] = \begin{cases} 0 & \text{då } x = 0 \\ 0 & \text{då } y = 0 \\ \max(M[x-1,y,0], M[x,y-1,0]) + Os[x,y] & \text{då } x \geq 1,\ y \geq 1 \text{ och } b = 0 \\ \max(M[x-1,y,1], & \\ \quad M[x,y-1,1], & \\ \quad M[x-1,y,0] + Os[x-1,y+1], & \text{då } x \geq 1,\ y \geq 1 \text{ och } b = 1 \\ \quad M[x,y-1,0] + Os[x+1,y-1]) + Os[x,y] & \end{cases}$$

Beräkningsordning? Först beräknar vi hela M för $b = 0$ och därefter hela M för $b = 1$. För varje b-värde beräknar vi matrisen efter ökande avstånd från övre vänstra hörnet, dvs efter ökande värde på $x + y$. Då kommer vi alltid att ha räknat ut varje M-element innan vi behöver använda det för att beräkna ett annat M-element enligt rekursionen.

3. Lite mindre trött robot

Jämfört med lösningen till uppgift 2 behöver vi utöka strukturen S med ett tredje lager för $b = 2$. Rekursionen för $b = 0$ och $b = 1$ är oförändrad. Vi behöver bara lägga till fallet $x \leq n$, $y \leq n$, $b = 2$. Om roboten backar vid två olika tillfällen under sin vandring (dvs inte två gånger i rad) så ser rekursionen för den andra backningen ut på motsvarande sätt som den första backningen, bara med skillnaden att b är ett snäpp högre.

Vi behöver nu utreda vad som händer om roboten backar två gånger i rad, och efter dessa backningar är tillbaka i ruta (x, y). När första backningen sker måste roboten vara i någon av rutorna $(x+2, y)$, $(x+1, y+1)$ och $(x, y+2)$. Efter första backningen måste roboten befinna sig i någon av rutorna $(x+1, y)$ och $(x, y+1)$. När roboten backat tillbaka till (x, y) och går framåt igen så vinner den inget på att gå tillbaka till samma ruta igen, så i en optimal vandring kommer båda rutorna $(x+1, y)$ och $(x, y+1)$ att besökas. Efter nästa steg är roboten i en av rutorna $(x+2, y)$, $(x+1, y+1)$ och $(x, y+2)$, och av samma skäl så kan det i en optimal vandring inte vara samma ruta som den varit i tidigare.

Rekursionen blir därför:

$$S[x,y,b] = \begin{cases} 0 & \text{då } n+1 \le x \le n+2 \\ 0 & \text{då } n+1 \le y \le n+2 \\ Os[n,n] & \text{då } x = y = n \\ \max(S[x+1,y,0], S[x,y+1,0]) + Os[x,y] & \text{då } x \le n,\ y \le n,\ b = 0 \\ \max(S[x+1,y,1], & \\ \quad S[x,y+1,1], & \\ \quad Os[x,y+1] + S[x+1,y,0], & \text{då } x \le n,\ y \le n,\ b = 1 \\ \quad Os[x+1,y] + S[x,y+1,0]) + Os[x,y] & \\ \max(S[x+1,y,2], & \\ \quad S[x,y+1,2], & \\ \quad Os[x,y+1] + S[x+1,y,1], & \\ \quad Os[x+1,y] + S[x,y+1,1], & \\ \quad \max(S[x+2,y,0] + Os[x+1,y+1], & \\ \qquad S[x+2,y,0] + Os[x,y+2], & \text{då } x \le n,\ y \le n,\ b = 2 \\ \qquad S[x+1,y+1,0] + Os[x+2,y], & \\ \qquad S[x+1,y+1,0] + Os[x,y+2], & \\ \qquad S[x,y+2,0] + Os[x+2,y], & \\ \qquad S[x,y+2,0] + Os[x+1,y+1]) + & \\ \quad + Os[x+1,y] + Os[x,y+1]) + & \\ + Os[x,y] & \end{cases}$$

Det maximala värdet finns sedan i $S[1,1,2]$.

Beräkningsordningen blir samma som i lösningen till uppgift 2 utvidgad med ett tredje lager. Först beräknar vi hela S för $b = 0$, därefter hela S för $b = 1$ och sist hela S för $b = 2$. För varje b-värde beräknar vi matrisen diagonalvis efter ökande avstånd från den understa punkten i rutnätet, alltså det nedre högra hörnet i S, dvs efter avtagande värde på $x + y$. Då kommer vi alltid att ha räknat ut varje S-element innan vi behöver använda det för att beräkna ett annat S-element enligt rekursionen.

Låt oss beskriva hur roboten ska röra sig genom rutnätet med hjälp av instruktionerna $\vee$ (gå snett ner till vänster), $>$ (gå snett ner till höger), $\wedge$ (backa snett upp till höger), $<$ (backa snett upp till vänster). Under beräkningens gång lagrar vi robotens optimala rörelse (som kan vara ett steg, tre steg eller sex steg beroende om 0, 1 eller 2 backningar görs) vid det aktuella läget i $D[x,y,b]$.

LessTiredRobot$(n, Os[1..n, 1..n]) =$
  for $i \leftarrow 1$ **to** $n+2$ **do**
    $Os[i,n+1] \leftarrow Os[i,n+2] \leftarrow -\infty$
    $Os[n+1,i] \leftarrow Os[n+2,i] \leftarrow -\infty$
    $S[i,n+1,0] \leftarrow S[i,n+1,1] \leftarrow S[i,n+1,2] \leftarrow 0$
    $S[n+1,i,0] \leftarrow S[n+1,i,1] \leftarrow S[n+1,i,2] \leftarrow 0$
    $S[i,n+2,0] \leftarrow S[i,n+2,1] \leftarrow S[i,n+2,2] \leftarrow 0$
    $S[n+2,i,0] \leftarrow S[n+2,i,1] \leftarrow S[n+2,i,2] \leftarrow 0$
  $S[n,n,0] \leftarrow S[n,n,1] \leftarrow S[n,n,2] \leftarrow Os[n,n]$
  $D[n,n,0] \leftarrow D[n,n,1] \leftarrow D[n,n,2] \leftarrow$ ""
  assert $ASS_1 =$
      Basfallen är korrekt beräknade, liksom kolumn $n+1, n+2$ och rad $n+1, n+2$ i matriserna
  for $xysum \leftarrow 2n-1$ **downto** 2 **do**
    inv $ASS_1 \wedge$
      ($S[x,y,0]$ och $D[x,y,0]$ är korrekt beräknade för alla $x + y > xysum$, $1 \le x \le n$, $1 \le y \le n$)
    for $x \leftarrow 1$ **to** $\min(n, xysum) - 1$ **do**
      $y \leftarrow xysum - x$

if $S[x+1,y,0] > S[x,y+1,0]$ **then**
$S[x,y,0] \leftarrow S[x+1,y,0] + Os[x,y]$; $D[x,y,0] \leftarrow$"$\vee$"
else $S[x,y,0] \leftarrow S[x,y+1,0] + Os[x,y]$; $D[x,y,0] \leftarrow$"$>$"
assert $ASS_2 =$
($S[x,y,0]$ och $D[x,y,0]$ är korrekt beräknade för alla x och y)
for $xysum \leftarrow 2n-1$ **downto** 2 **do**
inv $ASS_2 \wedge$
($S[x,y,1]$ och $D[x,y,1]$ är korrekt beräknade för alla $x+y > xysum$, $1 \le x \le n$, $1 \le y \le n$)
for $x \leftarrow 1$ **to** $\min(n, xysum) - 1$ **do**
$y \leftarrow xysum - x$
if $S[x+1,y,1] > S[x,y+1,1]$ **then**
$S[x,y,1] \leftarrow S[x+1,y,1] + Os[x,y]$; $D[x,y,1] \leftarrow$"$\vee$"
else $S[x,y,1] \leftarrow S[x,y+1,1] + Os[x,y]$; $D[x,y,1] \leftarrow$"$>$"
if $Os[x,y+1] + S[x+1,y,0] + Os[x,y] > S[x,y,1]$ **then**
$S[x,y,1] \leftarrow Os[x,y+1] + S[x+1,y,0] + Os[x,y]$; $D[x,y,1] \leftarrow$"$><\vee$"
if $Os[x+1,y] + S[x,y+1,0] + Os[x,y] > S[x,y,1]$ **then**
$S[x,y,1] \leftarrow Os[x+1,y] + S[x,y+1,0] + Os[x,y]$; $D[x,y,1] \leftarrow$"$\vee\wedge>$"
assert $ASS_3 =$
($S[x,y,b]$ och $D[x,y,b]$ är korrekt beräknade för alla x, y och $b \in \{0,1\}$)
for $xysum \leftarrow 2n-1$ **downto** 2 **do**
inv $ASS_3 \wedge$
($S[x,y,2]$ och $D[x,y,2]$ är korrekt beräknade för alla $x+y > xysum$, $1 \le x \le n$, $1 \le y \le n$)
for $x \leftarrow 1$ **to** $\min(n, xysum) - 1$ **do**
$y \leftarrow xysum - x$
if $S[x+1,y,2] > S[x,y+1,2]$ **then**
$S[x,y,2] \leftarrow S[x+1,y,2] + Os[x,y]$; $D[x,y,2] \leftarrow$"$\vee$"
else $S[x,y,2] \leftarrow S[x,y+1,2] + Os[x,y]$; $D[x,y,2] \leftarrow$"$>$"
if $Os[x,y+1] + S[x+1,y,1] + Os[x,y] > S[x,y,2]$ **then**
$S[x,y,2] \leftarrow Os[x,y+1] + S[x+1,y,1] + Os[x,y]$; $D[x,y,2] \leftarrow$"$><\vee$"
if $Os[x+1,y] + S[x,y+1,1] + Os[x,y] > S[x,y,2]$ **then**
$S[x,y,2] \leftarrow Os[x+1,y] + S[x,y+1,1] + Os[x,y]$; $D[x,y,2] \leftarrow$"$\vee\wedge>$"
if $Os[x+1,y+1] > Os[x,y+2]$ **then**
$maxpart \leftarrow S[x+2,y,0] + Os[x+1,y+1]$; $Dmax \leftarrow$"$>\vee\wedge<\vee\vee$"
else $maxpart \leftarrow S[x+2,y,0] + Os[x,y+2]$; $Dmax \leftarrow$"$>><<\vee\vee$"
if $S[x+1,y+1,0] + Os[x+2,y] > maxpart$ **then**
$maxpart \leftarrow S[x+1,y+1,0] + Os[x+2,y]$; $Dmax \leftarrow$"$\vee\ \vee\ \wedge\wedge>\vee$"
if $S[x+1,y+1,0] + Os[x,y+2] > maxpart$ **then**
$maxpart \leftarrow S[x+1,y+1,0] + Os[x,y+2]$; $Dmax \leftarrow$"$>><<\vee>$"
if $S[x,y+2,0] + Os[x+2,y] > maxpart$ **then**
$maxpart \leftarrow S[x,y+2,0] + Os[x+2,y]$; $Dmax \leftarrow$"$\vee\ \vee\ \wedge\wedge>>$"
if $S[x,y+2,0] + Os[x+1,y+1] > maxpart$ **then**
$maxpart \leftarrow S[x,y+2,0] + Os[x+1,y+1]$; $Dmax \leftarrow$"$\vee><\wedge>>$"
if $maxpart + Os[x+1,y] + Os[x,y+1] + Os[x,y] > S[x,y,2]$ **then**
$S[x,y,2] \leftarrow maxpart + Os[x+1,y] + Os[x,y+1] + Os[x,y]$; $D[x,y,2] \leftarrow Dmax$
assert $ASS_4 = (S[x,y,b]$ och $D[x,y,b]$ är korrekt beräknade för alla x, y och $b \in \{0,1,2\})$
$x \leftarrow 1$; $y \leftarrow 1$; $b \leftarrow 2$
$path \leftarrow$ ""
while not $(x = n$ **and** $y = n)$ **do**
inv $ASS_4 \wedge$ ($path$ beskriver robotens optimala rörelse från (1,1) till (x,y)
då roboten orkar göra b backningar till)
$dir \leftarrow D[x,y,b]$
for $d \leftarrow 0$ **to** $dir.len - 1$ **do**
case $dir[d]$ **of**
'$\vee$': $x \leftarrow x+1$

```
                '>': y ← y + 1
                '∧': x ← x − 1
                '<': y ← y − 1
            if dir.len > 1 then b ← b − dir.len/3
            path ← path.append(dir)
    print "Maximal mängd osmium: ", S[1, 1, 2]
    print "Instruktioner för roboten: ", path
```

Indata består av n^2 tal, och varje algoritm som löser problemet måste titta på alla tal i indata (för en algoritm som missar att titta på något tal kommer inte att göra rätt eftersom det tal som den inte tittat på skulle kunna vara större än summan av alla andra tal, alternativt vara 0) så n^2 är en undre gräns för tidskomplexiteten. Algoritmens komplexitet är $O(n^2)$ (som mest två nästlade slingor som vardera går linjärt många varv), vilket alltså är optimalt.

För att motivera korrektheten för algoritmen behöver vi visa att för varje probleminstans (korrekt indata) skriver algoritmen ut den maximala mängd osmium som roboten kan plocka upp, följt av en beskrivning av en väg som ger detta värde.

Eftersom det är en dynamisk programmeringsalgoritm ska vi dels visa att rekursionen korrekt beskriver delproblemlösningarna och dels att pseudokoden korrekt implementerar rekursionen och returnerar rätt värde.

Att rekursionen är korrekt har vi motiverat ovan. Vi har också motiverat att beräkningsordningen fungerar. Delproblemet $S[1, 1, 2]$ ger den sökta lösningen, vilket också är det som algoritmen skriver ut först. Det som återstår är att visa att algoritmen beräknat alla värden för $S[x, y, b]$ enligt rekursionen, att $D[x, y, b]$ satts korrekt, och att en optimal stig beräknas och skrivs ut korrekt.

Beräkningsordningen säger att S först ska beräknas för $b = 0$ och därefter för $b = 1$ och $b = 2$, och för varje b ska $S[x, y, b]$ beräknas diagonalvis efter ökande avstånd från nedre högra hörnet, dvs för minskande värde på summan $x + y$, vilket stämmer med algoritmen.

När varje yttre for-slinga startar är $xysum = n - 1$ och enda värdena som stämmer in på villkoren i invarianten är $x = n$, $y = n$. Basfallen säger att invarianten då är sann. Efter sista varvet i for-slingan är $xysum = 1$ och alla $1 \leq x, y \leq n$ stämmer in på invariantvillkoren, så om invarianten är sann är den efterföljande assertsatsen sann.

Det är (någorlunda) enkelt att kontrollera att tilldelningarna till S görs enligt rekursionen och att motsvarande element i D sätts korrekt, vilket visar att invarianten i slingan är en invariant.

När while-slingan börjar säger dess invariant att *path* ska beskriva robotens optimala rörelse från (1,1) till $(1, 1)$ då roboten orkar göra 2 backningar till, vilket följer direkt av basfallen. Efter while-slingan säger invarianten att *path* beskriver robotens optimala rörelse från (1,1) till (n, n), dvs det vi vill ska skrivas ut. Sist i algoritmen skrivs också *path* ut. While-invarianten är en invariant för slingan om raderna i slingan utvidgar *path* på slutet med den rörelse som roboten i det aktuella läget optimalt ska göra och uppdaterar läget (x, y, b). Den optimala rörelsen i (x, y, b) ligger enligt ASS_4 i $D[x, y, b]$. Inuti slingan sätts *dir* till denna rörelse och sist i slingan läggs *dir* till på slutet av *path*. Uppdateringen av x och y görs korrekt eftersom varje rörelseinstruktion i *dir* gås igenom, avkodas och motsvarande x/y uppdateras. b ska bara ändras om en backning görs. Om två backningar görs har *dir* längd 6, och om en backning görs har *dir* längd 3. När algoritmen därför minskar b med längden av *dir* delat med 3 kommer därför b att uppdateras med antalet backningar som görs i den aktuella positionen, vilket är korrekt.

Därmed är korrektheten för algoritmen visad.