DD2350 Mästarprov 1

Karla Rehn

September 2020

Uppgift 1. *Dator med extra bitoperationer*

Vi vill utnyttja den speciella funktionen IsZero som finns på datorn. Vi vet också att algoritmen ska vara en dekompositionsalgoritm. Det är alltså en bra idé att dela upp bitvektorn v i mindre vektorer, och ropa funktionen på dessa mindre bitar. Om IsZero returnerar true för en vektor vet vi att vi inte behöver leta efter 1:or i den vektorn, då den bara innehåller 0:or. Om IsZero returnerar false ropar vi funktionen rekursivt på de mindre vektorerna tills de små vektorerna bara innehåller ett element, eller IsZero returnerar true och vi alltså vet att det inte längre finns någon 1:a i vektorn. I pseudokod ser det här ut:

```
k = 0
NumberOfOnes(v, i, j):

  if i == j and v[i] == 1:
    k ++
    return

  if IsZero(v, i, j):
    return

  else:
    indx = (j-i)/2
    u = v[i:indx]
    w = v[indx:j]
    NumberOfOnes(u, 1, len(u))
    NumberOfOnes(w, 1, len(w))
```

Korrekthet

Korrektheten för dekompositionsalgoritmer visas med induktion. Basfallet är här då n = 1; i = j. Detta kommer ge k = 0 om det enda elementet är en 0:a, och k = 1 om elementet är en 1:a, vilket är korrekt.

Induktionsantagandet är att NumberOfOnes(v,i,p) beräknas korrekt.

Induktionssteget är att visa att algoritmen beräknar NumberOfOnes(v,i,p+1) korrekt, givet att NumberOfOnes(v,i,p) har beräknats korrekt. När detta är visat kan vi fylla på upp till godtyckligt p eftersom vi vet att NumberOfOnes(v,i,i) är korrekt.

Komplexitet

Den här algoritmen har tidskomplexitet $\mathcal{O}(k * log(n) + 1)$, där k är antalet 1:or i v och n totala antalet element.

Om vektorn inte innehåller några 1:or returnerar NumberOfOnes() direkt, och k kommer stanna på 0. Det här går i konstant tid, då IsZero tar konstant tid. Om vektorn innehåller en enda 1:a kommer NumberOfOnes() köras tills vi hittar denna 1:a. Detta kommer ta $log(n)$ tid, då hälften av vektorn försvinner i varje iteration med hjälp av IsZero. Att hitta alla k 1:orna kan då ses som att köra denna operation k gånger, till ett totalt antal körningar av $k * log(n)$, och tidskomplexiteten blir $\mathcal{O}(k * log(n) + 1)$, vilket var den önskade komplexiteten.

Uppgift 2. *Trött robot*

Som indata har vi matrisen `Os[i][j]`, som beskriver hur mycket osmium det finns i ruta `i, j` samt `Os`' dimensioner, $n * n$. Alla värden i `Os` är icke-negativa, och roboten får från ruta `i,j` bara röra sig till ruta `i+1,j` eller `i,j+1` utom en enda gång, då den kan röra sig till `i-1,j` eller `i,j-1`, givet att den varit där tidigare.

Algoritm

Vi konstruerar en till matris `kum_Os`, där varje ruta innehåller maxmängden osmium man kan ha plockat på sig på väg dit, utan att ha tagit något baksteg. Den här matrisen hittar vi enkelt med dynamisk programmering;

```
kum_Os[1][1] = Os[1][1]
md = 1
while md < (2*n + 1):
    k = 1
    while k < md:
        i = md - k - 1
        j = k
        if i == 1 and j == 1:
            skip

        kum_Os[i][j] = max(kum_Os[i-1][j], kum_Os[i][j-1])
        kum_Os += Os[i][j]
        j++
    i++
```

Därefter konstruerar vi en tredje matris `kum_Os_B`, där varje ruta innehåller maxmängden osmium man kan ha plockat på sig på väg dit, *om* man tagit ett backsteg. Den här matrisen är lite klurigare att fylla, men det görs enligt följande rekursionsrelation:

```
kum_Os_B[1][1] = Os[1][1]
md = 1
while md < (2*n + 1):
    k = 1
    while k < md:
        i = md - k - 1
        j = k
        if i == 1 and j == 1:
            # vi vill fylla på första raden och kolumnen
            # men hoppa över [1][1], eftersom vi redan fyllt i där
            skip
        kum_Os_B[i][j] = max(kum_Os_B[i-1][j],
                             kum_Os_B[i][j-1],
                             kum_Os[i][j-1] + Os[i+1][j-1],
                             kum_Os[i-1][j] + Os[i-1][j+1])
        kum_Os_B[i][j] += Os[i][j]
        k++
    md++
```

Vi fyller på enligt manhattan-avståndet från `[1][1]`, så beräkningsordningen blir "diagonalvis" snarare än rad- eller kolumnvis.

`kum_Os_B[i][j]`, `kum_Os[i][j]` och `Os[i][j]` med `i,j` som inte uppfyller $\{\texttt{i,j} \in \mathbb{Z} | 1 \leq \texttt{i,j} \leq n\}$ blir -1.

Det största värdet i varje ruta i bakstegsmatrisen `kum_Os_B` är alltså det största av att ha spenderat baksteget i en tidigare ruta, och att spendera baksteget i den här rutan, åt antingen det ena eller andra hållet. Ett baksteg kan ses som att i ett enda steg ta både rutan vi går till, och rutan snett under den till höger, eller rutan snett över den till vänster. Även om ett baksteg *skulle* kunna användas till att bara springa fram och tillbaka, finns det inget att tjäna på det då osmiumet plockas upp vid den första passagen genom rutan.

Korrekthet

Vi visar att algoritmen är korrekt med hjälp av induktion. I ruta `1,1` har vi inte tagit några tidigare steg, så mängden vi samlat på oss är helt enkelt värdet i rutan.

I ruta m,n > 1 är mängden vi samlat på oss maximal om mängden vi samlat på oss i ruta [m-1][n] och [m][n-1] för både `kum_Os_B` och `kum_Os` maximal. (Och max() verkligen returnerar det maximala värdet, vilket vi antar.) Eftersom vi för varje ruta måste komma från någon av dessa 4 rutor - `kum_Os_B[m-1][n]`,`kum_Os_B[m][n-1]`, `kum_Os[m-1][n]` eller `kum_Os[m][n-1]` -

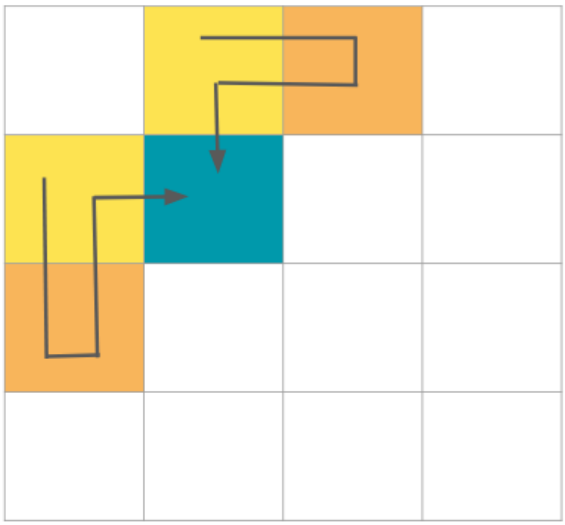


Figure 1: För att komma till den blåa rutan kan vi gå med baksteget via de orangea, eller direkt från de gula rutorna.

och vi tar den maximala av de fyra, måste mängden i ruta m, n alltså också vara maximal.

Komplexitet

Den innersta loopen i rekursionsrelationerna går i båda fallen i konstant tid, under antagandet att addition går i konstant tid. Att hitta max av 2 eller 4 element, att plocka ut element `i,j`, samt att tilldela element `i,j` går också i konstant tid. Den innersta loopen behöver köras $2n$ gånger över index md, n gånger över index k. Tidskomplexiteten för att konstruera matriserna blir alltså $\mathcal{O}(2n^2)$, och algoritmen blir totalt $\mathcal{O}(4 * n^2) = \mathcal{O}(n^2)$

Uppgift 3. *Lite mindre trött robot*

I den här uppgiften kan roboten backa två steg, givet att den varit i rutorna den backar till. Indatan är densamma som i uppgift 2, en osmiummatris och matrisens dimensioner.

Algoritm

För att kunna backa två gånger utvidgar vi lösningen i uppgift 2 med ytterligare en kumulativ matris, `kum_Os_B2`. Den här kommer alltså innehålla n gånger n rutor, där varje ruta innehåller den maximala mängden osmium vi samlat på

oss på väg dit, givet att vi backat två gånger under vägen. `kum_Os_B` och `kum_Os` konstrueras på samma sätt som i uppgift 2.

`kum_Os_B2[i][j]`, `kum_Os_B[i][j]`, `kum_Os[i][j]` och `Os[i][j]` med `i,j` som inte uppfyller $\{\texttt{i,j} \in \mathbb{Z} | 1 \leq \texttt{i,j} \leq n\}$ blir även här 0.

`kum_Os_B2` kan då fyllas på med

```
kum_Os_B2[1][1] = Os[1][1]
md = 1
while md < (2*n + 1):
    k = 1
    while k < md:
        i = md - k - 1
        j = k
        if i == 1 and j == 1:
            skip

        kum_Os_B2[i][j] = max(kum_Os_B2[i-1][j],
                              kum_Os_B2[i][j-1],
                              kum_Os_B[i][j-1] + Os[i+1][j-1],
                              kum_Os_B[i-1][j] + Os[i-1][j+1],
                              (kum_OS[i-1][j] + Os[i-1][j+1]
                              + Os[i-1][j+2]),
                              (kum_Os[i][j-1] + Os[i+1][j-1]
                              + Os[i+2][j-1]))
        kum_Os_B2[i][j] += Os[i][j]
        k++
    md++
```

En visualisering av de 6 möjliga stegen i `max()` finns i figur 2 .

Den uppmärksamma läsaren har säkert noterat att den här rekursionsrelationen inte innehåller de "krokiga" baksteg roboten skulle kunna ta. Anledningen till att dessa är exkluderade är för att deras resultat kan nås bättre med enkla baksteg. En längre förklaring finns i figur 3.

När vi har konstruerat våra tre kumulativa matriser går det att se vilken väg vi tagit för att komma till ruta `n,n` i `kum_Os_B2` genom att se vilken av de rutor som leder till `n,n` som har störst värde. Jag sparar dessa tidigare rutor i en "backpointer". Det som sparas i backpointern är en lista med `[m,i,j]`, där `m` är vilken matris vi varit i, `i` vilken rad och `j` vilken kolumn.

Pseudokod för detta presenteras nedan:

```
bp = []
nbp = [3,n,n] # vet att vi börjar backningen i
# den tredje matrisen, i den sista rutan
bp.append(nbp)
m = nbp[1]
```

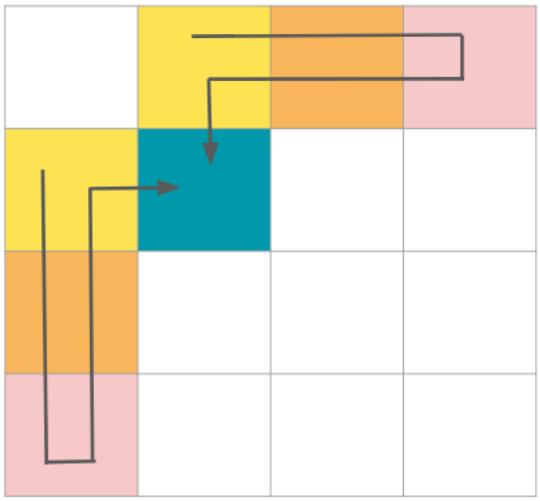
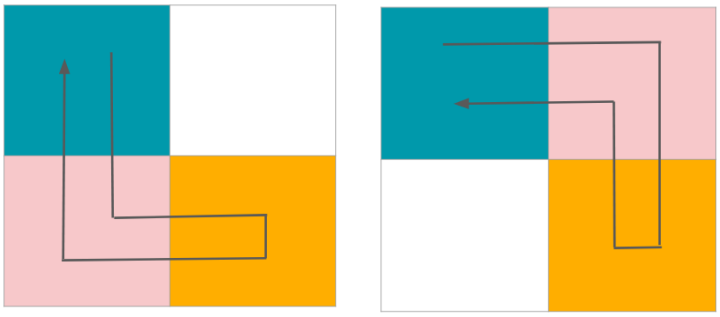


Figure 2: Vi kommer till den blå rutan i `kum_Os_B2`. Har vi kvar båda bakstegen går vi från `kum_Os` till den blåa rutan och plockar upp den orangea och den rosa på vägen.

```
i = nbp[2]
j = nbp[3]

while i != -1 and j != -1:
    if m == 2:
        prev_ruta = [kum_Os_B2[i-1][j], kum_Os_B2[i][j-1]),
                kum_Os_B[i-1][j] + OS[i-1][j+1],
                kum_Os_B[i][j-1] + OS[i+1][j-1],
                kum_Os[i-1][j] + OS[i-1][j+1] + OS[i-1][j+2],
                kum_Os[i][j-1] + OS[i+1][j-1] + OS[i+2][j-1]]
    if m == 1:
        prev_ruta = [-1,-1,
                    kum_Os_B[i-1][j],
                    kum_Os_B[i][j-1],
                    kum_Os[i][j-1] + Os[i+1][j-1],
                    kum_Os[i-1][j] + Os[i-1][j+1]]
                # vi kan aldrig gå till matris 3
                # från matris 2 när vi backar
    if m == 0:
        prev_ruta = [-1,-1,-1,-1,
                    kum_Os[i-1][j],
                    kum_Os[i][j-1]]
            # vi kan aldrig gå till matris 2 eller 3
```



(a) Vänster-upp (b) Upp-vänster

Figure 3: De ”krokiga” bakstegen, uppåt-vänster och vänster-uppåt. Dessa steg används för att få tag på den orangea och rosa rutan, och sedan gå till den vita. Detta innebär att den orange rutan bättre nås efter den vita, och om rutan till höger om (i fall (a)) eller nedanför (i fall (b)) den vita fortfarande är åtråvärd är det sundare att använda det andra baksteget till att gå tillbaka till den vita rutan. Dessa steg är alltså **inte** med i rekursionsrelationen.

```
            # från matris 1 när vi backar
            # eftersom vi aldrig kan gå till matris 1
            # från matris 2 eller 3 när vi går framåt

    ind = argmax(prev_ruta)
    sm = [[3,i-1,j],[3,i,j-1],[2,i-1,j],
    [2,i,j-1],[1,i-1,j],[1,i,j-1]]
    nbp = sm[ind]
    m = nbp[1]
    i = nbp[2]
    j = nbp[3]
    bp.append(nbp)
```

När vi väl vill titta på vilken stig vi tagit genom det snöiga osmium-landskapet är det inte längre särskilt intressant hur många baksteg vi tagit (vilket är vad vi håller koll på med hjälp av matris `m`). Stig-instruktionerna som skrivs ut blir tydligare om de bara är en lista med `i,j` där `i` är vilken rad vi är på, och `j` vilken kolumn. Om vi däremot går från en matris till en annan vet vi att vi använt bakstegen, så fler steg behöver läggas till i stigen.

```
bp.reverse()
path = []
for step in bp:
    path.append([step[2],step[3]])
    # lägg till vilken rad och vilken kolumn vi ska till
```

```
        if step[1] != next_step[1]:
            if step[1] - next_step[1]  == 2:
                # 2 skillnad i matris:
                # vi tog två baksteg på en gång
                if step[2] - next_step[2]  == 1:
                    #skillnad i rader: baksteget togs åt vänster
                    path.append([step[2],step[3]+1])
                    path.append([step[2],step[3]+2])
                    path.append([step[2],step[3]+1])
                    path.append([step[2],step[3]])
                else:
                    # skillnad i kolumner: baksteget togs uppåt
                    path.append([step[2]+1,step[3]])
                    path.append([step[2]+2,step[3]])
                    path.append([step[2]+1,step[3]])
                    path.append([step[2],step[3]])

            if step[1][0] - step[0][0] == 1:
                # vi tog ett baksteg
                if step[1] - next_step[1]  == 1:
                    # skillnad i rader: baksteget togs åt vänster
                    path.append([step[0][1],step[0][2]+1])
                    path.append([step[0][1],step[0][2]])
                else:
                    # skillnad i kolumner: baksteget togs uppåt
                    path.append([step[0][1]+1,step[0][2]])
                    path.append([step[0][1],step[0][2]])
    print(path)
```

Korrekthet

Som i uppgift 2 bevisar vi korrektheten med hjälp av induktion.

Basfallet är `n = 1`, då algoritmen returnerar värdet i den enda rutan utan att gå in i `while`-looparna. Detta är korrekt.

Vi antar att algoritmen beräknar den maximala mängden osmium vi kan samla på oss i ruta `p,p` korrekt.

Vi vill då visa att givet att matriserna `kum_Os`, `kum_Os_B` och `kum_Os_B2` är korrekt beräknade upp till ruta `p,p` beräknas `kum_Os_B2[p][p+1]`, `kum_Os_B2[p+1][p]` och `kum_Os_B2[p+1][p+1]` också korrekt. Vi börjar med att visa `kum_Os_B2[p][p+1]`.

Om `p` < 2 kommer `kum_Os_B2[p-1][i]` $=$ `kum_Os_B2[i][p-1]` $= 0$ $\forall$`i`. Dessutom gäller

```
kum_Os[p][p+1]= max(kum_Os[p-1][p+1], kum_Os[p][p]) + Os[p][p+1]

kum_Os_B[p][p+1]= max(kum_Os_B[p-1][p+1], kum_Os_B[p][p],
                kum_Os[p-1][p+1] + Os[p-1][p+2],
```

```
                kum_Os[p][p] + Os[p+1][p]) + Os[p][p+1]

kum_Os_B2[p][p+1]= max(kum_Os_B2[p-1][p], kum_Os_B2[p][p]),
                kum_Os_B[p-1][p+1] + Os[p-1][p+2],
                kum_Os_B[p][p] + Os[p+1][p],
                kum_Os[p-1][p] + Os[p-1][p+2] + Os[p-1][p+3],
                kum_Os[p][p] + Os[p+1][p] + Os[p+2][p]) + Os[p][p+1])
                + Os[p][p+1]
```

En längre motivering av varför man måste komma från någon av de 2, 4, eller 6 rutorna som dyker upp i rekursionsrelationerna finnes nedan. Vi går däremot först vidare med rekursionsbeviset, och antar att dessa möjligheter är korrekta.

Eftersom matriserna beräknas längs diagonalerna, är `kum_Os[p-1][p+1]` korrekt beräknad för alla `i` när `kum_Os` är korrekt beräknad upp till `kum_Os[p][p]`. Alla värden på `Os` var givna som indata. Vi antar att vår max-funktion och additionen verkligen fungerar korrekt, och att vi måste komma från antingen `kum_Os[p-1][p+1]` eller `kum_Os[p][p]`; `kum_Os[p][p+1]` blir det största möjliga värdet vi kan ha samlat på oss. Samma resonemang upprepas för `kum_Os_B` och `kum_Os_B2`, med skillnaden att vi kan komma från fler rutor. Ingen av dessa rutor ligger på en senare diagonal än `[p][p]` däremot, så resonemanget håller.

Vi vet alltså att när `kum_Os_B` är korrekt beräknad upp till `kum_Os_B[p][p]` är den också korrekt beräknad upp till `kum_Os_B[p][p+1]` (och så vidare för de andra två matriserna).

Resonemanget för att visa att `kum_Os_B[p+1][p]` är korrekt är i stort likadant, de värden som behövs för att beräkna `kum_Os_B[p][p+1]` är korrekt beräknade när `kum_Os_B[p][p]` är korrekt beräknad.

När nu `kum_Os_B[p][p+1]` och `kum_Os_B[p+1][p]` är bevisat korrekta visar vi `kum_Os_B[p+1][p+1]`:

```
kum_Os[p+1][p+1]= max(kum_Os[p][p+1], kum_Os[p+1][p]) + Os[p+1][p+1]

kum_Os_B[p+1][p+1]= max(kum_Os_B[p][p+1], kum_Os_B[p+1][p],
                kum_Os[p][p+1] + Os[p][p+2],
                kum_Os[p+1][p] + Os[p+2][p]) + Os[p][p+1]

kum_Os_B2[p+1][p+1]= max(kum_Os_B2[p][p+1], kum_Os_B2[p+1][p]),
                kum_Os_B[p][p+1] + Os[p][p+2],
                kum_Os_B[p+1][p] + Os[p+2][p],
                kum_Os[p][p+1] + Os[p][p+2] + Os[p][p+3],
                kum_Os[p+1][p] + Os[p+2][p] + Os[p+3][p])
                + Os[p][p+1]
```

Utöver `Os`, som var given, är de värden som krävs för att kunna beräkna `kum_Os_B2[p+1][p+1]` korrekt `kum_Os_B2[p][p+1]` och `kum_Os_B2[p+1][p]`. Dessa har visats korrekta ovan. Induktionsbeviset är alltså slutfört, då algoritmen ger korrekt resultat för basfallet och för `p + 1`, givet att `p` beräknats korrekt.

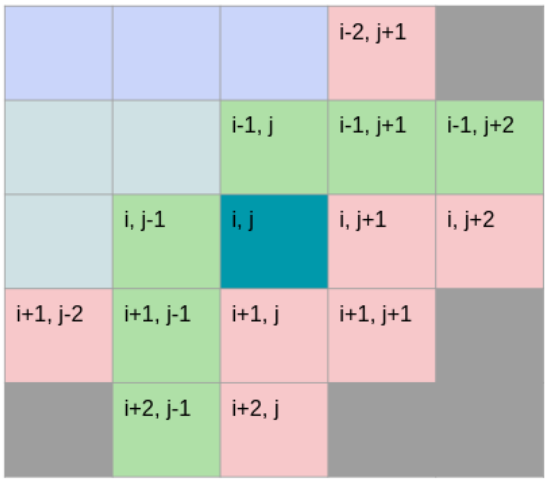


Figure 4: Steg till `kum_Os_B2[i][j]` vi kan få för oss att ta är markerade med rött och grönt. De gröna är med i rekursionsrelationen, de röda är inte.

Om de tre matriserna beräknas korrekt, kommer backpointern också beräknas korrekt – vi gör samma operation fast åt andra hållet –, och om backpointern beräknas korrekt skrivs stigen ut korrekt, då det i stort bara är en reversering av backpointern och ett print-statement.

En längre stigmotivering

Att vi i `kum_Os` inte kan komma från `kum_Os_B` eller `kum_Os_B2` är för att vi där inte har några baksteg att förbruka. Därför kan vi inte heller gå från `kum_Os_B` till `kum_Os_B2`, när vi har ett baksteg kvar får vi inte plötsligt mer energi och det andra baksteget åter.

Att man i `kum_Os` och `kum_Os_B` måste komma från någon av de 2 respektive 4 rutor som ingår i deras rekursionsrelationer inses någorlunda enkelt. Att vi efter baksteget inte vill fortsätta åt hållet vi kom ifrån (tillbaka till den orangea rutan i figur 1) är för att vi då bara slösat baksteget på att gå fram och tillbaka, utan att besöka några nya rutor. Då alla värden i `Os` är icke-negativa vill vi alltid besöka så många rutor som möjligt. Av samma anledning går vi inte till ruta `[i-2][j]`, `[i-1][j-1]` eller `[i][j-2]` på väg till ruta `[i][j]` med hjälp av baksteget; är det ett tillåtet drag har vi redan besökt rutan vi backade till och det finns alltså inget osmium kvar i den.

För `kum_Os_B2` finns det 13 (!) vägar in till ruta `[i][j]` man kan få för sig är möjliga. Trots det har jag bara tagit med 6 av dem i rekursionsrelationen. Även här vill vi inte gå igenom en ruta fler än 1 gång om det inte leder till att vi kan besöka en ny ruta. Figur 3 innehåller en förklaring till varför vi inte vill ha med `kum_Os[i][j-1] + Os[i+1][j-1] + Os[i+1][j]` och `kum_Os[i-1][j] +`

`Os[i-1][j] + Os[i+1][j+1]`. Fallen där baksteget plockar upp `Os[i-2][j+2]` eller `Os[i+2][j-2]` innehåller ett framåtsteg i mitten, så de ses istället som två separata baksteg. Fallet där baksteget plockar upp `Os[i+1][j+1]` är exkluderat för att vi inte kan gå in i ruta `kum_Os_B2[i][j]` från ruta `kum_Os_B2[i][j]`, utan vi kräver alltid att algoritmen rör sig framåt. Av samma anledning är fallen där baksteget plockar upp de två elementen till höger alternativt under rutan vi är på väg in exkluderade. Dessa kommer täckas när `kum_Os_B2[i+1][j]` eller `kum_Os_B2[i][j+1]` beräknas. På så sätt ströks 7 av de 13 möjligheterna, och vi har bara kvar vägar där vi inte behöver kontrollera att vi verkligen besökt rutorna vi backar till, då vi går dem framåt i samma omgång som vi backar, betrakta figur 2.

Komplexitet

Vi antar att osmiummängden i indatan är storleksbegränsad på så sätt att addition med maxmängden fortfarande tar konstant tid. Det är då berättigat att analysera tidskomplexiteten i enhetskostnad. (Eftersom mindre än ett ton (10^6g) osmium produceras per år[1] känns detta som ett rimligt antagande, och eftersom osmiummängden anges i heltal behöver vi inte heller oroa oss för decimaler.)

Tidskomplexiteten för algoritmen är samma som i uppgift 2, plus tiden det tar att konstruera den tredje matrisen och tiden det tar att plocka ut stigen.

Att konstruera `kum_Os_B2` går i $\mathcal{O}(n^2)$, en noggrannare motivation av varför finns i uppgift 2, med skillnaden att vi här tar max av 6 element istället för 2 eller 4. `while`-loopen för att hitta backpointern kommer köras n*2 - 1 gånger, då detta är antalet rutor som besöks för att ta sig från ena hörnet av en n*n-matris till det andra utan några baksteg. Det som görs i `while`-loopen är att plocka ut element ur en matris, vilket görs i konstant tid, att konstruera en lista med 6 element, vilket görs i konstant tid, att ta max av en lista med 6 element, vilket görs i konstant tid, samt att lägga till ett element i slutet av en lista, vilket görs i konstant tid. Allt i `while`-loopen görs alltså i konstant tid, och loopen behövde som sagt köras n*2 - 1 gång, så den totala tidskomplexiteten blir $\mathcal{O}(n*2-1) = \mathcal{O}(n)$. Att sedan skapa och printa stigen med bakstegen görs även det i $\mathcal{O}(n+3) = \mathcal{O}(n)$.

Algoritmens totala tidskomplexitet – tiden det tar att beräkna matriserna, backpointern och stigen – blir alltså $\mathcal{O}(6*n^2+2*n-1+2*n+3) = \mathcal{O}(n^2)$

Då vi har n^2 element, och behöver titta på varje element för att vara säkra på att vi har den maximala vägen är en undre gräns för algoritmens komplexitet $\mathcal{O}(n^2)$. Vi har matchande undre och övre gräns och algoritmen är alltså optimal.

❄

En körbar version av den här lösningen finns på min github efter den 16:e oktober.