NIO løsningsforslag finale 2026

2026-03-14

Skaar et al.

1 Kirkeklokker

Oppgave

Det er 49 tidspunkter hvor kirkeklokken skal slå. Hver gang klokken slår så husker innbyggerne også antallet slag de siste tre gangene. Vi skal velge antall slag for hvert tidspunkt, slik at innbyggerne kan vite hva klokken er ut fra kun informasjonen de husker. Målet er å oppnå dette med så få slag som mulig.

Dette er ekvivalent med å finne en liste med tall hvor hvert intervall av størrelse 4 er unikt, og minimere summen av listen.

En løsning er å ta inspirasjon fra hva kirken gjør, å slå ett slag hvert kvarter, og så indikere antall timer hver hele time:

1 1 1 1 2 1 1 1 3 1 1 1 4 1 1 1 5 1 1 1 6 1 1 1 7 1 1 1 8 1 1 1 9 1 1 1 10 1 1 1 11 1 1 1 12 1 1 1 13

Denne løsningen har totalt 127 slag og gir derfor 23p. Her blir «timeslagene» brukt for å annonsere info, og kvarter-slagene konstant. Ved å gjøre omvendt, enkode timene på kvarterslagene, og alltid slå samme antall slag hver hele time, kan vi få summen ned mot 100. Denne strategien fungerer best hvis vi lar 2 slag representere en hel time, og la 1,3 og 4 være tallene vi bruke for å enkode antall timer:

1 1 1 1 2 1 1 3 2 1 1 4 2 1 3 1 2 1 3 3 2 1 3 4 2 1 4 1 2 1 4 3 2 1 4 4 2 3 1 1 2 3 1 3 2 3 1 4

Internt i NIO var det også suksess med å prøve å finne løsninger for hånd, helt ned til 84 slag:

1 1 1 1 2 1 1 1 3 1 1 2 2 1 1 3 2 1 1 2 3 1 1 3 3 1 2 2 2 1 2 1 2 2 3 1 2 3 2 1 2 3 3 1 1 4 1 1 1

Full uttelling

Siden oppgaven er output only, har vi anledning til å kjøre søk ganske lenge, og levere inn løsninger som bare printer ut de beste sekvensene vi finner.

Med brute force, tilfeldig søk eller pann og papir kan man se at det finnes løsninger som bruker under 90 slag.

Det optimale antall slag viser seg å være 81. For å generere sekvenser med såpass lave summer så må snittverdien være ca. 1.65. Dette betyr at en optimal løsning må ha overvekt av «små tall». En måte å lete etter slike løsninger blir da å generere tilfeldige sekvenser, men å velge lave tall med høyere sannsynlighet, gjerne kalibrert slik at forventet verdi ligger på rundt 1.65.

Ved å generere sekvenser med å plukke tall på denne formen, og å kun velge gyldige løsninger (begynne på nytt hvis ugyldig) kan vi innen rundt 30 sekunder generere en eller flere løsninger.

Det bør også nevnes at tradisjonell brute force med godt implementert pruning vil også innen rimelig, dog en del tid, finne en optimal løsning.

Random søk i python


```
import random
import math

best = 1000
seq = []
seen = set()

def next_value():
    end = seq[-3:]

    for _ in range(100):
        val = random.randint(3, 30)
        new = 5 - int(math.log2(val))
        newend = tuple(end + [new])

        if newend not in seen:
            return new
```

Random søk i python (ii)


```
def solve(i):
    global seq, best

    if len(seq) == 49:
        val = sum(seq)
        if val < best :
            best = val
            print("Found new best with value", val)
            print(" ".join(map(str, seq)))
        return

    new = next_value()
    if new is None:
        return

    seq += [new]
    end = tuple(seq[-4:])
```

Random søk i python (iii)

```
    seen.add(end)
    solve(i+1)
    seen.remove(end)
    seq.pop()


for _ in range(100000000):
    solve(0)
```

Bruteforce i python


```python
best = 1000
seq = []
prev = set()

def solve(val, i):
    global seq, best

    if val + 49 - i >= best:
        return

    if len(seq) == 49:
        best = val
        print("Found new best with value", val)
        print(" ".join(map(str, seq)))
        return

    seq += [1]
```

Bruteforce i python (ii)


```
    for v in range(4):
        seq[-1] = v+1
        end = tuple(seq[-4:])
        if end in prev:
            continue
        prev.add(end)
        bf(val + v + 1, i+1)
        prev.remove(end)
    seq.pop()

solve(0, 0)
```

Bevis av optimum

Det er mulig å bevise at 81 er optimum, men dette er ikke nødvendig for å løse oppgaven.

En løsning må inneholde 46 unike intervaller av lengde fire. Vi ønsker at antall slag i hvert intervall skal være så liten som mulig. Hvis vi genererer alle intervaller av lengde fire, og velger de 46 med lavest sum så får vi:

- 1 intervall med sum 4
- 4 intervaller med sum 5
- 10 intervaller med sum 6
- 7 intervaller med sum 20
- 8 intervaller med sum 10

Hvis vi summerer alle slagene for disse intervallene, og teller med at tidspunkt vil tilhøre flere intervaller, så får vi at løsningen må inneholde minst 81 slag. Siden disse intervallene er minst mulig, er det ikke mulig å få en løsning som er bedre enn 81 slag.

2 Paidag

Oppgave

Vi er gitt m personer, hvor hver person er villig til å lage 1 pai. For hver pai er vi gitt en liste med distinkte ingredienser (ut av n mulige) personen trenger for å lage paien sin. Vi er gitt i oppgave å velge ut så få ingredienser som mulig for å kunne lage minst k paier (for hver valgte ingrediens kjøper vi inn ubegrenset antall)

Subtask 1: $K \leq 3$

I dette subtasket er det veldig få mulige ingredisener, så man kan sjekke alle kombinasjonene og se om det gir nok paier.

Man kan bruke en bitmask for å representere hvilke ingredienser som kreves i en pai.

For eksempel, hvis en pai krever ingrediens 0 og 2 kan vi representere den som 101_2. Da er det enkelt å iterere over alle bitmasks opp til 2^K og så sjekke om vi har de nødvendige ingredisense ved å bitvis AND-e sammen tilbudet og bitmasken vi sjekker nå. Det er også mulig å bruke setoperasjoner.

Tidskompleksitet: $O(2^K N)$

Samme løsning fungerer også i subtask 3: $N \leq 1000$, $K \leq 10$.

Subtask 1: $K \leq 3$ (ii)


```
N, M, K = map(int, input().split())
offers = [sum(1 << int(x) for x in input().split()[1:]) for _ in range(N)]
minimum_ingredients = 1e10

def popcount(x):
    return bin(x).count("1")

for bm in range(1 << K):
    can_make = 0
    for pie in offers:
        if pie & bm == pie:
            can_make += 1

    if can_make >= M:
        minimum_ingredients = min(minimum_ingredients, popcount(bm))

print(minimum_ingredients)
```

Subtask 2: $M = N$

Her må vi lage alle paiene. Da må alle ingredienser som brukes i minst én pai kjøpes inn.

```
N, M, K = map(int, input().split())

offers = [list(map(int, input().split()))[1:] for _ in range(N)]

ingredients = set()

for offer in offers:
    for ing in offer:
        ingredients.add(ing)

print(len(ingredients))
```

Tidskompleksitet: $O(NK)$

Subtask 4: Fullstendig løsning

La $f(S)$ være antallet paier som har S som ingrediensmengde. Da er antallet paier som kan lages gitt at vi kjøper ingredienser S lik $g(S) = \sum_{S' \subseteq S} f(S')$. I prinsippet vil vi gjøre noe slik:

```
for S in all_possible_solutions:
  if g(S) >= k:
    best = min(best, len(S))
```

Problemet er bare å regne ut $g(S)$ effektivt for *alle* S samtidig.

Subtask 4: Fullstendig løsning (ii)

Zeta-transform

Gitt en funksjon $f : 2^U \to \mathbb{Z}$, er den zeta-transformerte $g(S) = \sum_{S' \subseteq S} f(S')$.

Anta $U = \{0, 1, ..., n-1\}$. La $g_i(S) = \sum_{S' \subseteq S} f(S')$, hvor S' er en delmengde av S hvor vi kun har lov til å «fjerne» elementene $\{0, ..., i-1\}$, slik at $g_0(S) = f(S)$ og $g_n(S) = g(S)$.

Vi kan regne ut g_{i+1} fra g_i slik:

$$g_{i+1}(S) = \begin{cases} g_i(S) & \text{hvis } i \notin S \\ g_i(S) + g_i(S \setminus \{i\}) & \text{hvis } i \in S \end{cases}$$

Det er $O(2^k)$ celler å regne ut i hver iterasjon, og n iterasjoner, som gir en kompleksitet på $O(k2^k)$ for Zeta-transformen. (Kodeeksempel neste slide)

Tidskompleksitet: $O(K2^K + NK)$

Subtask 4: Fullstendig løsning (iii)


```
n, m, k = map(int, input().split())
cnt = [0] * (1 << k)

for _ in range(n):
    v = [int(i) for i in input().split()][1:]
    cnt[sum([1<<i for i in v])] += 1

for i in range(k):
    for p in range(1 << k):
        if p & (1 << i):
            cnt[p] += cnt[p - (1 << i)]
output = k
for i in range(1 << k):
    if cnt[i] >= m:
        output = min(output, i.bit_count())

print(output)
```

3 Skattekartet

Oppgave

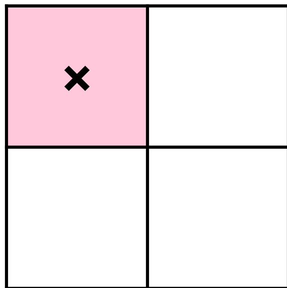
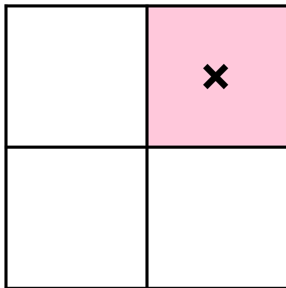
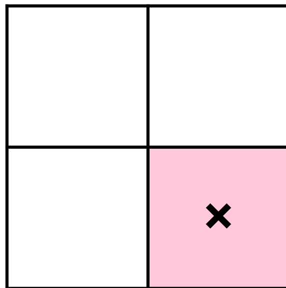
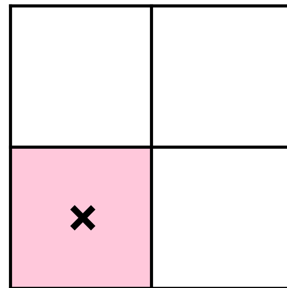
Man får oppgitt dimensjonene på et kvadratisk rutenett, N, samt et koordinatpar (x, y) der skatten er plassert.

I første pass skal man markere et valgfritt antall celler, før man i videre passes må finne skatten når rutenettet er rotert.

På denne oppgaven finnes det flere forskjellige løsninger.

Subtask 1: $N = 2$, maks 3 spørringer

Her kan man markere skatten og så sjekke tre ruter, hvis man ikke finner skatten blant dem, er det den siste ruta.

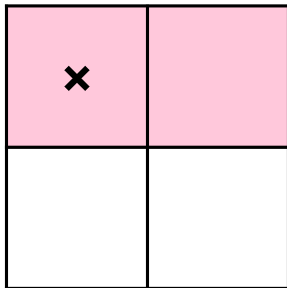
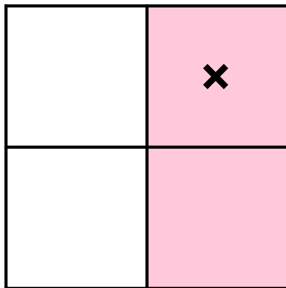
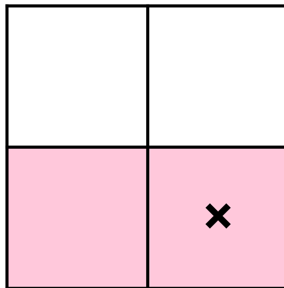
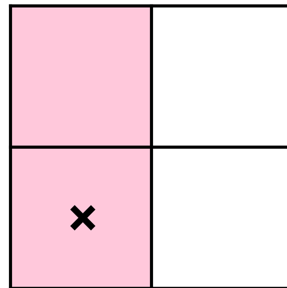





Subtask 1: $N = 2$, maks 3 spørringer (ii)


```
current_pass = input()
if current_pass == "kartlegg":
    n, x, y = map(int, input().split())
    print(f"marker {x} {y}")
    print("avslutt")
else:
    n = int(input())
    checks = [(1, 1), (1, 2), (2, 1)]
    found = False
    for (x, y) in checks:
        print(f"sjekk {x} {y}", flush=True)
        ans = input()
        if ans == "mynt":
            print(f"skatt {x} {y}")
            found = True
            break
    if not found: print(f"skatt 2 2")
```

Subtask 2: $N = 2$, maks 2 spørringer

Her kan man for eksempel markere skatten og ruta med klokka for den. Hvis man spør om to ruter med klokken for hverandre kan man finne ut hvor skatten er.

Subtask 2: $N = 2$, maks 2 spørringer (ii)

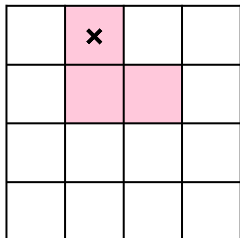
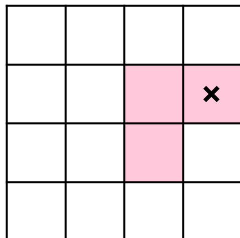
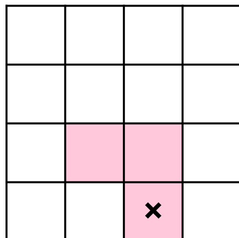
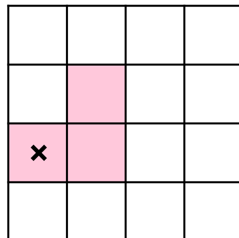
```python
current_pass = input()
def next_cell(x,y):
    return {
        (1,1): (2,1),
        (2,1): (2,2),
        (2,2): (1,2),
        (1,2): (1,1)
    }[(x,y)]
def mark(x, y):
    print(f"marker {x} {y}")
def query(x, y):
    print(f"sjekk {x} {y}", flush=True)
    ans = input()
    return ans == "mynt"
def answer(x, y):
    print(f"skatt {x} {y}")
    exit(0)
```

Subtask 2: $N = 2$, maks 2 spørringer (iii)


```
if current_pass == "kartlegg":
    n, x, y = map(int, input().split())
    mark(x, y)
    mark(*next_cell(x, y))
    print("avslutt")
else:
    n = int(input())
    tl = query(1,1)
    tr = query(2,1)
    treasure = {
        (True, True): (1,1),
        (True, False): (1,2),
        (False, True): (2,1),
        (False, False): (2,2)
    }
    answer(*treasure[(tl,tr)])
```

Subtask 3: $N = 4$, maks 6 spørringer

Her kan man bruke 2 spørringer på å finne hvilken kvadrant skatten ligger i, for eksempel med forrige løsning og så sjekke hver celle i den kvadranten. Hvis den ligger i midten vil ingen av de tre andre være markert.

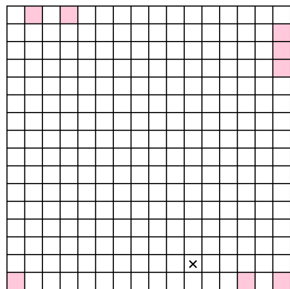
Generell løsning

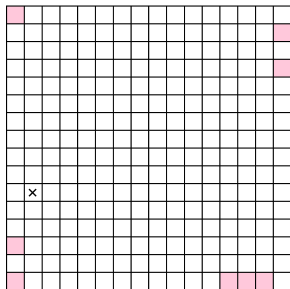
Man kan bruke binærkoding til å spesifisere koordinatet langs hver kant. Altså langs toppen spesifiserer du hvor langt skatten er fra venstre kant, langs høyre side avstand fra toppen, etc. Dette vil da fungere for alle rotasjoner.

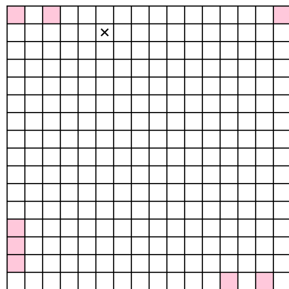
Dette krever $\log_2(N)$ bits og dette vil alltid være mindre enn eller lik N.

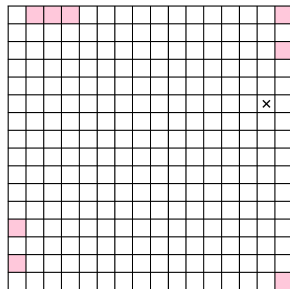
Her er $x = 11$, $y = 15$

Altså, med 0-indeksering: $x = 10 = 1010_2$, $y = 14 = 1110$









Generell løsning (ii)


```python
import math
current_pass = input()

def mark(x, y):
    print(f"marker {x+1} {y+1}")

def query(x, y):
    print(f"sjekk {x+1} {y+1}", flush=True)
    ans = input()
    return ans == "mynt"

def answer(x, y):
    print(f"skatt {x+1} {y+1}")


if current_pass == "kartlegg":
    n, x, y = map(int, input().split())
```

Generell løsning (iii)


```
bits_needed = int(math.log2(n))

val_top = (x-1)
val_right = (y-1)
# Works since n is a power of 2
val_bottom = (~val_top) & ((1 << bits_needed) - 1)
val_left = (~val_right) & ((1 << bits_needed) - 1)

for i in range(bits_needed):
    if val_top & (1 << i):
        mark(i, 0)
    if val_right & (1 << i):
        mark(n-1, i)
    if val_bottom & (1 << i):
        mark(n-1-i, n-1)
    if val_left & (1 << i):
```

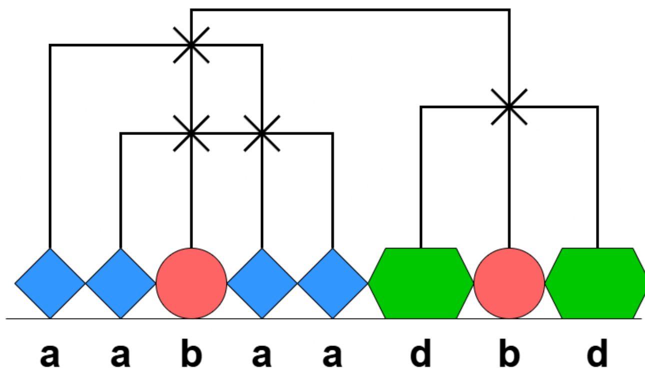
Generell løsning (iv)


```
            mark(0, n-1-i)

    print("avslutt")

else:
    n = int(input())

    bits_needed = int(math.log2(n))

    x = 0
    y = 0

    for i in range(bits_needed):
        x += query(i, 0) << i
        y += query(n-1, i) << i

    answer(x, y)
```

4 Kretskort

Oppgave

Man får oppgitt en sekvens av ulike porter på et kretskort, og så skal man parre sammen porter av samme type slik at man får maksimalt antall krysninger av banene i mellom.



Subtask 2: $n \leq 10$

Her kan man bruteforce alle mulige kombinasjoner av to porter og telle antall kollisjoner.

```
n,k = map(int, input().split())
s = input()

permutations = []

def gen_permutations(i, current):
    for y in range(i+1, n):
        if s[i]!=s[y]: continue

        j = i + 1
        while j in current or j == y: j+=1

        if j < n:
            gen_permutations(j, current+[i, y])
        elif j==n and len(current+[i, j])==n:
```

Subtask 2: $n \leq 10$ (ii)

```
            permutations.append(current+[i,y])

gen_permutations(0, [])

best = 0
for perm in permutations:
    score = 0
    for i in range(0, n, 2):
        for j in range(i+2, n, 2):
            if perm[i] < perm[j] <perm[i+1] < perm[j+1]:
                score += 1
            elif perm[j] < perm[i] < perm[j+1] < perm[i+1]:
                score += 1
    best = max(best, score)
print(best)
```

Tidskompleksitet: $O(N^2 N!!)$

Subtask 3: Det er nøyaktig to porter av hver type ($k = \frac{n}{2}$)

Her finnes det bare ett par av hver type, så kan slipper å bestemme hvilke som skal kobles sammen. Man kan derfor sjekke alle par av sammenkoblede par for å se om det finnes en kortslutning gitt ved

$a_1 < b_1 < a_2 < b_2$

Siden det er maks to av hver bokstav er $n \leq 52$, vi kan derfor ha en løsning med tidskompleksitet $O(n^2)$

```python
import itertools
n, k = map(int, input().split())
s = input()
pairs = []
for i in range(n):
    for j in range(i+1,n):
        if s[i] == s[j]: pairs.append((i,j))
shorts = 0
for (ar,al), (br,bl) in itertools.combinations(pairs, 2):
    if ar < br < al < bl or br < ar < bl < al: shorts += 1
print(shorts)
```

Subtask 1: $k = 1$

Her er vi gitt N `a`-er.

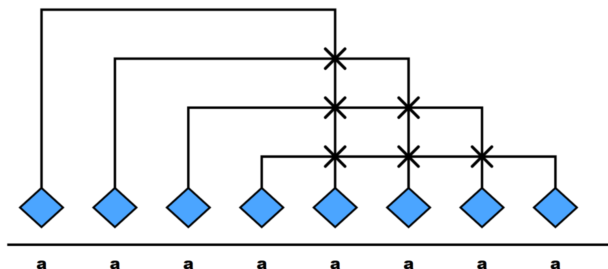
Observasjon: hvis man kobler sammen hver port i med port $i + \frac{n}{2}$ så vil man få maksimalt antall kortslutninger.

Da vil banene fra hver port i den venstre halvdelen av strengen kortslutte med banen til hver av portene til høyre for den i samme halvdel.

Dette gir totalt antall kortslutninger:

$$\frac{\frac{N}{2}\left(\frac{N}{2} - 1\right)}{2}$$

Tidskompleksitet: $O(1)$



Subtask 4: $n \leq 1000$ og strengen er alfabetisk sortert

Her gjør du subtask 1-løsningen og bare adderer sammen svarene for hver «blokk».

```
n, k = map(int, input().split())

s = input()

lens = [0] * 26

for c in s:
    lens[ord(c) - ord("a")] += 1

shorts = sum((n // 2) * (n // 2 - 1) // 2 for n in lens)
print(shorts)
```

Tidskompleksitet: $O(N + K)$

Subtask 5: $n \leq 1000$

Her kan vi gjøre samme observasjon som i subtask 1: hvis vi kobler sammen den i-te porten av type c med den $i + \frac{M}{2}$-te porten av type c hvor M er antall porter av type c vil vi få maksimalt antall kortsluttninger. Bevis for dette på senere slide.

Intuitivt trenger vi ikke tenke på de andre portene fordi det er ikke er noen andre konfigurasjoner som fører til flere baner mellom noen av portene av type c. (TODO sjekk at dette stemmer)

Vi sorterer banene på synkende startpunkter og markerer alle sluttpunktene på alle banene og når vi legger de til. Da teller vi hvor mange sluttpunker den nye banen krysser.

```
n, k = map(int, input().split())
s = input()

pos = [[] for i in range(k)]
for idx, c in enumerate(s):
    pos[ord(c)-ord('a')].append(idx)
```

Subtask 5: $n \leq 1000$ (ii)


```
par = []
for i in pos:
    for j in range(len(i) // 2):
        par.append((i[j], i[j + len(i) // 2]))

par.sort()

slutter = [0] * n
output = 0
for i, j in par:
    output += sum(slutter[i:j])
    slutter[j] += 1

print(output)
```

Tidkompleksitet: $O(N^2)$

Subtask 6: Generell løsning

Vi kan bruke samme løsning som for subtask 5, men for at det skal gå raskt nok bruker vi et segmenttre til å markere og telle sluttpunkter.

Dvs. bytte ut `slutter` med et segmenttre og

```
output += sum(slutter[i:j])
slutter[j] += 1
```

med segmenttre operasjoner i $O(\log N)$, i stedet for $O(N)$. For å lese mer om dette går det an å sjekke ut denne siden eller bare søke på nettet.

Det er mulig å gjøre det uten segmenttrær, men da må man være litt smartere i måten man ser på itervallene på.

Tidskompleksitet: $O(N \log N)$

Kode for den fulle løsningen ligger i vedlagt zip-fil på nettsiden.

Notasjon: a_1 og a_2 er koblet med hverandre, b_i og b_2 er koblet med hverandre, $0 \leq a_1, a_2, b_1, b_2 < m$ og $a_1 < a_2$ og $b_1 < b_2$ og representerer indeksene blant de m portene av en gitt type. «Venstre halvdel» og «Høyte halvdel» refererer hhv til de første og siste $m/2$ av portene.

Hvis vi ser på porter av en gitt type: vi kan anta at portene i venstre halvdel er koblet opp mot porter i høyre halvdel, for hvis ikke vil vi ha et par a_1, a_2 med begge endepunkter i venstre halvdel, og da ved pidgeonhole principle også et part b_1, b_2 i høyre halvdel. Hvis vi nå bare bytter om på a_2 og b_1 vil vi få en minst like god løsning, som vi kan se ved å gå gjennom alle $\binom{5}{2}$ mulige måter et annet par kan krysse løsningen vår og se at i alle tilfeller ender vi opp med minst like mange krysninger etter som før.

Vi kan videre anta at i en optimal løsning er portene i venstre halvdel koblet opp mot tilsvarende port (i samme rekkefølge) i høyre halvdel, for hvis ikke vil vi ha en inversjon (en tuppel $a_1 < b_1 < b_2 < a_2$), og da vil å bytte om på a_2 og b_2 igjen gi en minst like god løsning, ved å gå gjennom alle mulighetene.

5 Spionnettverk

Oppgave

Vi er gitt et **tre** som vi traverserer i en bestemt rekkefølge. Fra hver node går vi **alltid til den naboen med lavest verdi** som vi **ikke** har **besøkt fra denne noden før**. Hvis vi har besøkt alle naboene starter vi på nytt. Alle verdiene er distinkte.

Vi får så Q spørsmål på formen: i hvilken node er vi etter K_i sendinger. Utfordringen ligger i at K_i kan være veldig stor, slik at vi ikke rekker å simulere alle stegene.

Subtask 1: Lav K

Her er K lav nok til at vi kan bare simulere rekkefølgen med DFS. Start med å sortere alle kantene til hver node. Hold styr på forrige nabo pakken ble sendt til for hver node og øk den for hver gang noden blir besøkt.

Tidskompleksitet: $O(N \log N + K + Q)$

Subtask 2: Ingen noder har flere enn to utgånde kanter

Her vil grafen være en linje. Hvis vi tenker oss at vi har utforsket et segment på linjen vil vi bare traversere fram og tilbake mellom de to endepunktene på segmentet og så utvide en en og en node. Hvis vi kommer til node i fra node $i - 1$ vil vi enten gå tilbake til andre siden av segmentet hvis $v[i + 1] > v[i - 1]$, ellers vil vi også besøke $i + 1$. Når vi hopper til andre siden av segmentet vet vi avstanden vi har gått og trenger ikke simulere alle stegene.

Merk at det kan ta $\frac{N(N+1)}{2}$ *steg for å besøke alle noder, noe som er større en 32-bit int. Likevel er* $K_i < 10^9$.

Subtask 3: Vi starter på den endelige sykelen

Generell observasjon: hvis vi lar det gå lang nok tid, vil vi ende opp med å gå i en bestemt sykel som bruker alle kantene begge veier. Dette gjør at vi ikke trenger å simulere alle stegene.

Subtask observasjon: Vi starter på den endelige sykelen. Det gjør at man kan ta å sette $K_i \to K_i \bmod 2(N - 1)$.

Videre kan man simulere en hel sykel og svare på alle Q.

Tidskompleksitet: $O(N \log N + Q)$

Subtask 4: Lav N

Den maksimale tiden det tar før man kommer inn i den endelige sykelen er $\frac{N(N+1)}{2}$ (rett linje), som gjør at med lav N kan man simulere inntil man har besøkt alle noder. Etter det er man på den endelige sykelen og man kan da løse oppgaven tilsvarened Subtask 3.

Tidskompeksitet: $O(N^2 \log N + Q)$

Vi har fortsatt lyst til å bruke det med at man kommer inn i en fast sykel, men vi kan ikke lenger simulere oss dit. Vi se noen mønstre i traverseringen som gjør at vi ikke trenger å simulere alle stegene.

Observasjon: Nodene man har besøkt danner alltid en sammenhengende enhet. Noen ganger vil man traversere fra en kant av det utforskede området til en annen. Måten man gjør det vil alltid tilsvare et segment i den endelige sykelen.

Hvis man først regner ut den hvordan den endelige sykelen vil være kan man bruke segmenter av den før man har utforsket hele grafen. Stort sett kan man beregne avstanden med å se på første og siste gang en node blir besøkt i sykelen, ellers vil avstanden være 1.

Dette gir den omtrentlige algoritmen på neste side:

Full løsning: Del 2

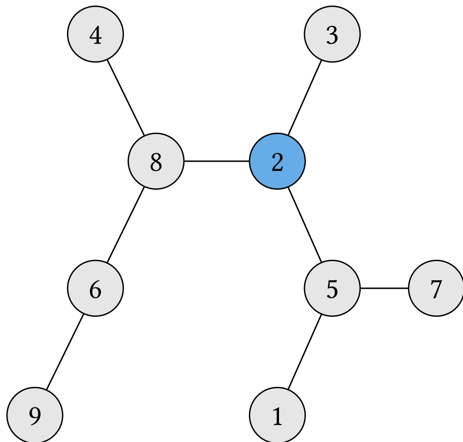
La `adj[i]` være en liste med naboene til i sortert i stigende rekkefølge. Lag en `stack st` og en `queue que` med S. Begge disse vil bare bestå av ubesøkte noder.

```
On new node p with parent:
  temporary_stack = []
  for i in adj[p]:
    if i == parent: continue
    if i < parent: temprorary_stack.push(i)
    if i > parent: que.push(i)
  st.push(temporary_stack.reverse())

  if !st.empty() go to top of st and account for distance to that point
  if !que.empty() go to top of qu and account for distance to that point
```

Dette blir en blanding av DFS og BFS der vi bare ser på noder vi ikke har besøkt for og når vi går til nye noder bruker vi det aktuelle segmentet i den endelige sykelen for å finne avstanden.
Kode for den fulle løsningen ligger i vedlagt zip-fil på nettsiden.

Eksempel

$N = 9, S = 2 : T = -1$



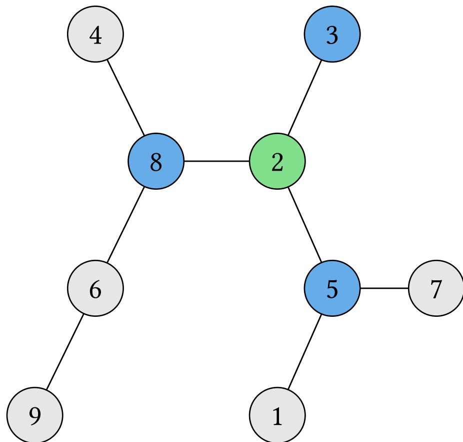
Endelig sykel:

2 3 2 5 7 5 1 5 2 8 4 8 6 9 6 8 2

```
qu = { 2 }
st = {}
```

Eksempel

$N = 9, S = 2 : T = 0$



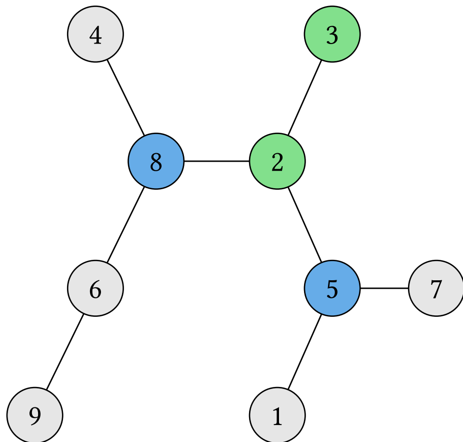
Endelig sykel:

2 3 2 5 7 5 1 5 2 8 4 8 6 9 6 8 2

```
qu = { 3,5,8 }
st = {}
```

Eksempel

$N = 9, S = 2 : T = 1$



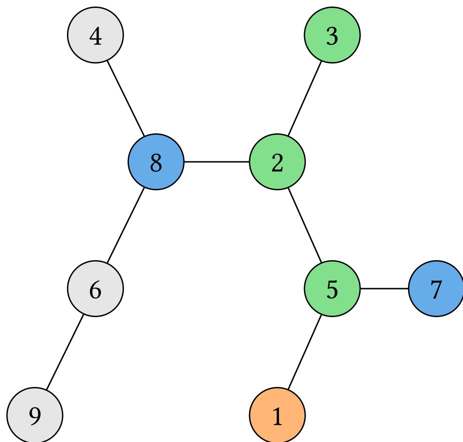
Endelig sykel:

2 3 2 5 7 5 1 5 2 8 4 8 6 9 6 8 2

$\Delta T = 1$

```
qu = { 5,8 }
st = { }
```

$N = 9, S = 2 : T = 3$



Endelig sykel:

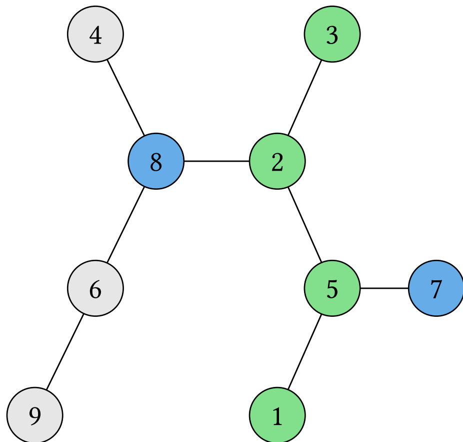
2 | 3 2 5 | 7 5 1 5 2 8 4 8 6 9 6 8 2

$\Delta T = 2$

```
qu = { 8, 7 }
st = { 1 }
```

Eksempel

$N = 9, S = 2 : T = 4$



Endelig sykel:

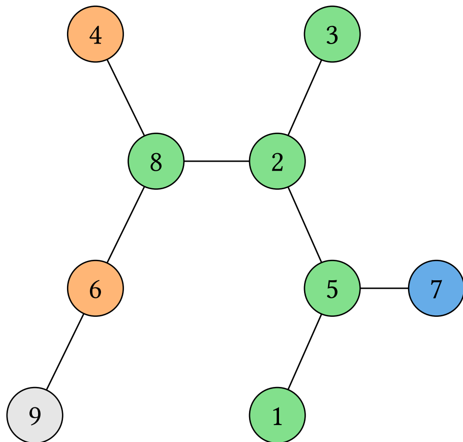
2 3 2 5 7 | 5 1 | 5 2 8 4 8 6 9 6 8 2

$\Delta T = 1$

```
qu = { 8, 7 }
st = { }
```

Eksempel

$N = 9, S = 2 : T = 7$



Endelig sykel:

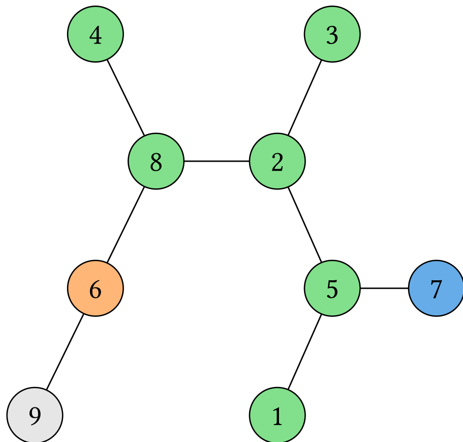
2 3 2 5 7 5 | 1 5 2 8 | 4 8 6 9 6 8 2

$\Delta T = 3$

```
qu = { 7 }
st = { 4, 6 }
```

Eksempel

$N = 9, S = 2 : T = 8$



Endelig sykel:

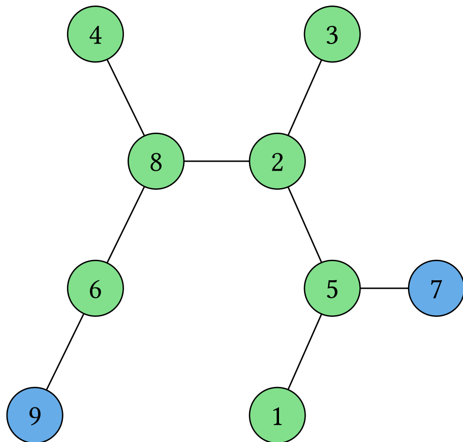
2 3 2 5 7 5 1 5 2 | 8 4 | 8 6 9 6 8 2

$\Delta T = 1$

```
qu = { 7 }
st = { 6 }
```

Eksempel

$N = 9, S = 2 : T = 10$



Endelig sykel:

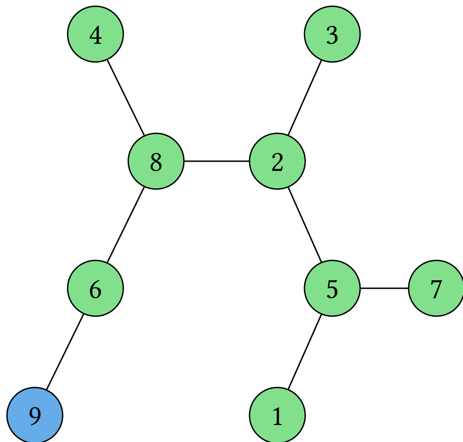
2 3 2 5 7 5 1 5 2 8 | 4 8 6 | 9 6 8 2

$\Delta T = 2$

```
qu = { 7, 9 }
st = { }
```

Eksempel

$N = 9, S = 2 : T = 17$



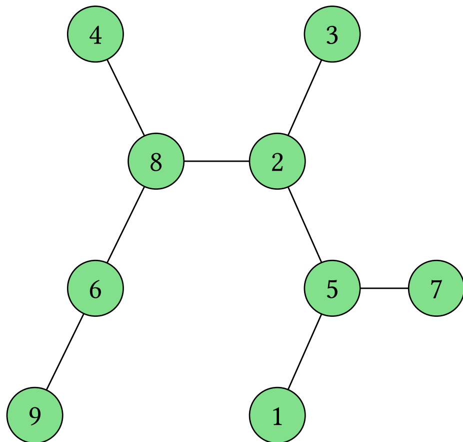
Endelig sykel:

| 2 3 2 5 7 | 5 1 5 2 8 4 8 6 9 | 6 8 2 |

$\Delta T = 7$

```
qu = { 9 }
st = { }
```

$N = 9, S = 2 : T = 26$



Endelig sykel:

2 3 2 5 | 7 5 1 5 2 8 4 8 6 9 | 6 8 2

$\Delta T = 9$

```
qu = { }
st = { }
```