Algoritmer og Sandsynlighed

Kompendium

Kevin Vinther

8. januar 2024

Indhold

1 Basic Counting Problems

- Pigeonhole (inkl Generalized)
- Permutationer og Kombinationer
- Subsets med repetition
- Pascal's Trekant
- Binomialkoefficienter
- Bevis for binomialsætning vha kombinatorisk argument
- Bevis $n^2 + 1$ delsekvenser med mindst $n + 1$ der er strikt nedad- eller opadgående.

1.1 Pigeonhole

Dueslagsprincippet (pigeonhole principle) er et simpelt princip, men kan bruges til meget i beviser.

Theorem 1 (Dueslagsprincippet)**.** *Hvis $k + 1$ objekter er i k bokse, så er der mindst en boks som vil have mere end ét element.*

Bevis. Vi beviser gennem modsigelse.

Hvis vi antager at ingen boks har mere end ét element, så er der højest $k \cdot 1 = k$ objekter i alt. Dette modsiger vores antagelse om at der er $k + 1$ obekter. $\square$

Theorem 2 (Generaliseret Dueslagsprincip)**.** *Hvis vi placerer N objekter i k bokse, så har en givet boks* ***mindst*** $\lceil \frac{N}{k} \rceil$

Bevis. Hvis vi antager at der er $\leq \lceil \frac{N}{k} \rceil - 1$ i hver boks, så $N \leq k \cdot (\lceil N/k \rceil - 1) < k((\frac{N}{k} + 1) - 1) = N$ $\square$

Theorem 3. *Hver sekvens af $n^2 + 1$ distinkte reelle tal $a_1, a_2, \cdots, a_{n^2+1}$ indeholder en subsekvens af n tal der enten er strictly increasing eller decreasing.*

Bevis. Lad i_k være længde af den længste increasing sekvens med start i a_k, og d_k den længste decreasing med start i a_k.

Hvis vores teorem ikke passer, så, for alle $k \in \{1, 2, \ldots, n^2 + 1\}$ gælder det for alle pair (i_k, d_k) at $i_k, d_k \in \{1, 2, \ldots, n\}$.

Men siden der er $n^2 + 1$ par gælder det at $\exists p, q$, således at $(i_q, d_q) = (i_p, d_p)$ hvor vi kan antage at $p < q$. Hvis $a_p < a_q$ så $i_p > i_q$, og på samme måde omvendt. $\square$

1.2 Kombinationer og Permutationer

Permutationer:
$P(n,r) =$ antal r-permutationer af et n-sæt
$P(n,r) = n \cdot (n-1) \cdot \dots \cdot (n-r+1) = \frac{n!}{(n-r)!}$

Bevis. Dette er tydeligt gennem product rule. □

Kombinationer:

$C(n,r) =$ Antallet af r-kombinationer af et k-sæt $C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r!(n-r)!}$

Bevis. Hver fixed r-permutation kan blive fundet fra en $r-$kombination ved at permutere de r elementer. Dermed $P(n,r) = r!C(n,r)$ □

1.3 Binomial Coefficients

Det her er nok det vigtigste.

$\binom{n}{r}$ er også kaldet den **binomielle** koefficient, fordi den forekommer som udvidelse i $(x+y)^n$.

Theorem 4 (The Binomial Theorem)**.** $(x+y)^n = \sum_{j=0}^{n} \binom{n}{j} x^{n-j} y^j$

Bevis. Leddene er af formen $x^{n-j}y^j$ for $j = 0, 1, 2, \dots, n$og koefficienterne til $x^{n-j}y^j$ er antallet af måder man kan vælge j y'er fra de n parenteser. □

Theorem 5 (Pascal's Identitet)**.** *$\forall n, j \in \mathbb{Z}$ hvor $n \geq k$ gælder det at* $\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$

Bevis. Fiks et element a i et $(n+1)$-sæt T lad $S = T - \{a\}$. Der er $\binom{n}{k}$ k-subsets af T som **ikke** indeholder a. Der er $\binom{n}{k-1}$ $k-$subsets af T som indeholder a, da de resterende $k-1$ elementer laver et $(k-1)$ subset af S. □

Vi kan bruge det her til pascal's trekant.

Theorem 6 (Vandermonde's Teorem)**.** *Når $r \leq \min\{m,n\}$ så gælder det at:*

$$\binom{n+m}{r} = \sum_{k=0}^{r} \binom{m}{r-k} \binom{n}{k}$$

1.4 Generaliseret Permutation og Kombinationer

1.4.1 Permutationer med gentagelse

Theorem 7. *Antallet af $r-$permutations af et $n-$sæt med getagelser tilladt er n^r.*

Bevis. Siden vi må vælge det samme igen og igen, har vi n valg hver gang. Hvis vi skal vælge en ting r gange bliver det $n \cdot n \cdots n = n^r$ □

1.4.2 Kombinationer med gentagelse

Theorem 8. *Antallet af r-kombinationer fra et $n-$sæt med gentagelser tilladt er $\binom{n+r-1}{r}$*

Bevis. Vi kan representere hver $r-$kombination med en streng af $n-1$ 'l' og r '*', hvor de $n-1$ 'l' er brugt til at markere hvilke af de n 'bokse' vi tager fra, og '*' elementer. For eksempel: "***ll*l**lll*". Der er $\binom{n-1+r}{r}$ måder at vælge de r '*' 'er □

Vi kan for eksempel bruge det her til at finde ud af hvor mange løsninger der er til en ligning, f.eks. $x_1 + x_2 + x_3 = 11$. Dette er bare antallet af 3-kombinationer fra et $3-$sæt med gentagelser. Derfor $\binom{11+3-1}{11} = \binom{13}{11} = \binom{13}{2} = 78$. Vi kan også gøre det med bounds. Tag samme eksempel, men hvor $x_1 \geq 1, x_2 \geq 2, x_3 \geq 3$. Med disse bounds har vi allerede placeret 6 items, og har derfor 5 tilbage. Dvs. at vi bare udregner 5-kombinationer: $\binom{5+3-1}{5} = 21$.

1.4.3 Permutationer med objekter man ikke kan skalne imellem

Før teoremet giver vi et eksempel: Find antal strenge du kan lave ud af bogstaverne fra ordet SUCCESSORo Du kan placere

- 3 'S' på $\binom{9}{3}$ måder
- 2 'C' på $\binom{6}{2}$ måder
- 1 'E' på $\binom{4}{1}$ måder
- 1 'O' på $\binom{3}{1}$ måder
- 1 'R' på $\binom{2}{1}$ måder
- 1 'U' på $\binom{1}{1}$ måder

Dermed får vi antal ved at gange dem sammen: $\binom{9}{3}\binom{6}{2}\binom{4}{1}\binom{3}{1}\binom{2}{1}\binom{1}{1}$.

Theorem 9. *Antallet af distinkte permutationer af n objekter af k typer med n_i af type k er $\frac{n!}{n_1!n_2!\cdots n_k!}$*

1.5 Distribuering af objekter i bokse

Der er 4 typer problemer her.

- Distribuering af objekter der ikke kan skelnes mellem i bokse der ikke kan skelnes mellem
- Distribuering af objekter der kan skelnes mellem i bokse der ikke kan skelnes mellem
- Distribuering af objekter der kan skelnes mellem i bokse der kan skelnes mellem
- Distribuering af objekter der ikke kan skelnes mellem i bokse der kan skelnes mellem

Theorem 10 (Sknelnet Objekter, Skelnet Bokse)**.** *Antal måder at dsitribuere n objekter man kan skelne imellem i k bokse man kan skelne imllem, med n_i objekter i box i er*

$$\frac{n!}{n_1!n_2!\cdots n_k!}$$

Theorem 11 (Ikke Sknelnet Objekter, Skelnet Bokse)**.** *Antal måder at distribuere n identiske objekter i k distinke bokse er lig antal n kombinationer af et n-sæt med gentagelse:*

$$\binom{n+k-1}{n}$$

Theorem 12 (Sknelnet Objekter, Ikke Skelnet Bokse)**.** *Her kan du bruge Stirling Numbers of the second kind.*

Theorem 13 (Ikke Sknelnet Objekter, Ikke Skelnet Bokse)**.** *Der må du bare tælle. Desværre*

2 Inclusion Exclusion

3 Discrete Probability

4 Randomized Algorithms

- Sandsynligheden for et korrekt min-cut med Karger's
- Max K-sat
- Max-3-SAT
- Quicksort og Median-Finding
- Monte Carlo
- Hiring Problem
- Majority Element

4.1 Randomized Majority Element

4.2 Majority Element med Uendelig Datastrøm

Ved Majority Element (Subsection 4.1) må vi læse array'et mere end én gang, og array'et er af en endelig mængde. Men hvad hvis vi har en datastrøm af en uendelig mængde, som vi kun må læse én gang?

Følgende er en algoritme der kan finde majority element (eller, hvis der ikke er et majority element, så finder den et element der ikke er) ved at kigge på datastrømmen kun én gang:

```
Majority(S):
c:= 0, l := ∅
for i := 1 to m
    if (x_i = l) then c := c + 1
    else c := c - 1
    if c <= 0 then
        c := 1, l := a_i
return l
```

Hvis der er et majority element, så vil den her algoritme returnere det. Algoritmen er positiv fordi den kun bruger $O(\log m)$ hukommelse for tælleren og $O(\log n)$ hukommelse fordi værdierne. Dette er godt, da datstrømmen i sig selv kan være kæmpe stor.

Vi antager at x_j forekommer mere end $m/2$ gange i S. Lad $X = x_j$ være værdien af majority elementet. For hvert i således at $x_{i=X}$ gør vi følgende:

1. Enten er $l \neq X$ og vi decreaser tælleren (og sætter måske $l = x_i$)

2. Eller $l = X$ og vi increase tælleren

33 kan ske mindre end $m/2$ gange
33 Tælleren er ≥ 1 ved termation af hvert loop. Så 33 vil ske til sidst.

4.2.1 Heavy Hitters Problem

Målet er at lave **k-frequency estimation** (også kaldet k-counters).

Lad f_j være frekvensen af værdi j, altså, antallet af gange j forekommer i datastrømmen $A = \langle a_1, a_2, \ldots, a_m \rangle \quad a_i \in \{1, 2, \ldots, n\}$

Vi vil gerne finde et estimat, $\hat{f}_j$, således at $f_j - \frac{m}{k} \leq \hat{f}_j \leq f_j$ for alle værdier j i datastrømmen.

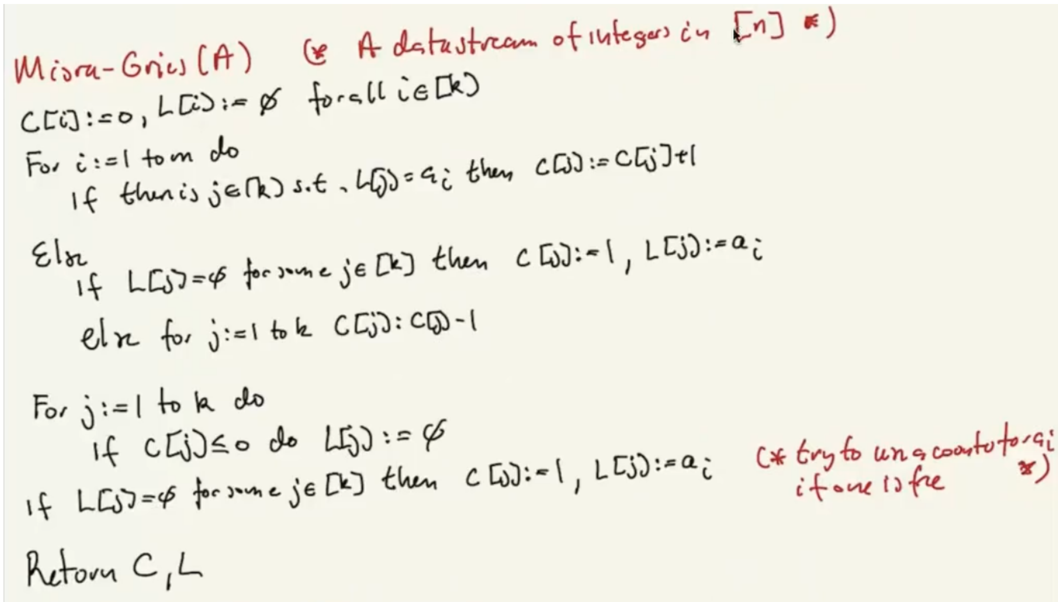
Antag at vi er givet en variabel $\varepsilon, 0 < \varepsilon < 1$. Så vil vi gerne have en datastruktur for en ε-approximate heavy hitters, så vi kan returnere:

- Alle f_j således at $f_j \geq \frac{m}{k}$ er i listen.

- Hvert element i listen forekommer mindst $\frac{m}{k} - \varepsilon m$ gange i A.

Misra-Gries Algoritme

Dette er en algoritmen med k tællere, $c[1], c[2], \ldots, c[n]$ fremfor 1. Lad $L[1], L[2], \ldots, L[k]$ være et array af k lokationer.



Givet denne algoritme, hvis du giver en værdi $q \in [n]$, så,

- Hvis $\exists j \in [k]$ med $L[j] = q$, returnerer den $\hat{f}_q = C[j]$
- Ellers returnerer $\hat{f}_q = 0$

Korrekthed

- En tæller $C[j]$ med $L[j] = q$ er kun incremented hvis $a_i = q$ så $f_q \leq f_q$ holder altid
- Hvis $C[j]$ med $L[j] = q$ er decremtned, så er alle andre counters decremented

5 Probabilistic Analysis

6 Indicator Random Variables

7 Universal Hashing

I hashing har vi et univers der er meget større end størrelsen på tabellen hvortil hash funktionen finder et index. Dvs. $|U| >> m$, hvor m er størrelsen på tabellen.

Problem med normal hashing: Der kan blive lavet angreb hvor en person der kender hash funktionens værdi kan lave en masse argumenter der alle hasher til det samme index. Vi fikser dette ved at tilfældigt og uafhængigt af nøglerne, vælge en universal hash funktion. På en universal hash funktion er sandsynligheden for at to værdier hasher til det samme $\leq 1/m$.

Vi kalder det en **kollision**, hvis to værdier hasher til den samme værdi. Vi ved naturligvis fra dueslagsprincippet (Teorem 1) at hvis der er mere end m værdier der bliver hashet, så vil der være mindst én kollision.

7.1 Universal Hashing

Lad $\mathcal{H}$ være en endelig kollektion af hash funktioner således at $h : U \rightarrow [m]$ for hver $h \in \mathcal{H}$.

Theorem 14. *Hash kollektionen $\mathcal{H}$ er **universal** hvis følgende holder:*

Lad $h \in \mathcal{H}$ være valgt tilfældigt. Så $\forall k, l \in U$ med $k \neq l$ $p(h(k) = h(l)) \leq \frac{1}{m}$

Theorem 15. *Antag at h er valgt tilfældigt fra en universal kollektino $\mathcal{H}$ af hash funktioner fra $U \rightarrow [m]$.*

Antag at vi har brugt h til at hashe et sæt $s \subseteq U$ med $|S| = n$ hvor vi bruger chaining til at løse kollisioner.

Lad $T = T[0], T[1], \ldots, T[m-1]$ være tabellen af linked lister vi får når $T[i]$ er en linked list med de elementer $x \in S$ hvorfra $h(x) = i$.

Derfra gælder følgende:

- *Hvis $k \notin S$, så $E[n_{h(k)}] \leq \frac{n}{m} = \alpha$ hvor $n_{h(k)}$ er længden af $T[h(k)]$*
- *Hvis $k \in S$ så $E(n_{h(k)}) \leq \alpha + 1$*

Bevis. $\forall k, l \in U, k \neq l$ definerer vi

$$X_{kl} = \begin{cases} 1 & \text{hvis } h(k) = h(l) \\ 0 & \text{hvis } h(k) \neq h(l) \end{cases}$$

For et fixed $k \in U$ definerer vi $Y_k = |\{l \in S \setminus \{k\} | h(k) = h(l)\}|$, altså er Y_k antallet af nøgler i $S \setminus \{k\}$ der hasher til samme værdi som k.

Derfor ved vi at $Y_k = \sum_{l \neq k, l \in S} X_{kl}$

$$\begin{aligned} E(Y_k) &= E(\sum_{l \neq k, l \in S} X_{kl}) \\ &= \sum_{l \neq k, l \in S} E(X_{kl}) \\ &\leq \sum_{l \neq k, l \in S} \frac{1}{m} \end{aligned} \tag{1}$$

- Hvis $k \notin S$ så $n_{h(k)} = Y_k$ og $|\{l \in S | l \neq k\}| = |S| = n$ så

$$E(n_{h(k)}) = E(Y_k) \leq \sum_{l \neq k, l \in S} \frac{1}{m} = \frac{|S|}{m} = \frac{n}{m} = \alpha$$

- Hvis $k \in S$, så $n_{h(k)} = Y_k + 1$ og $|\{l \in S | l \neq k\}| = |S| - 1 = n - 1$ dermed

$$E(n_{h(k)}) = 1 + E(Y_k) \leq 1 + \sum_{l \neq k, l \in S} \frac{1}{m} = 1 + \frac{n-1}{m} \leq 1 + \alpha$$

$\square$

Corollary 16. *Ved brug af Universal hashing + chaining, ved at starte fra en tom tabel med m pladser, tager det forventet tid $O(n)$ til at lave en sekvens af `INSERT`, `SEARCH` og `DELETE` operations når $O(m)$ af dem er `INSERT`*

Bevis. Vi indsætter $O(m)$ elementer, hvilket betyder at $|S| \in O(m)$. Dermed $\alpha = \frac{n}{m} \in O(1)$, fordi det er mindre end m. Den forventede længde af hver liste i tabellen er $O(1)$, så hver operation tager $O(1)$ forventede tid, så $O(n)$ for alle operationer. $\square$

7.2 Design af Universal Class

7.2.1 Cormen

Følgende er Cormen's metode til at lave en universal class:

- Vælg et primtal $p \geq |U|$ og antag at $U \subseteq \{0, 1, 2, \ldots, p-1\}$, $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}, \mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$
- Fordi p er et primtal kan vi løse ligninger $\mod p$.
- $p \geq |U| > m$ så $p > m$.
- For $a \in \mathbb{Z}_p^*$ og $b \in Z_p$ definér $h_{ab}(k) = ((ak + b) \mod p) \mod m$, $h_{ab} : \mathbb{Z}_p \to \mathbb{Z}_m$
- Sæt $\mathcal{H} = \mathcal{H}_{pm} = \{h_{ab} | a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$

Theorem 17. *Klassen $\mathcal{H}_{pm}$ er universal.*

7.2.2 KT

I KT's metode identificerer vi universet U med tupler af formen $(x_1, x_2, \ldots, x_r)$ for et haltal når $0 \leq x_i < p$ for $i = 1, 2, \ldots, r$. Derudover antager vi at universet er $U \subseteq \{0, 1, 2, \ldots, N-1\}$. Antag ydermere at n er størrelsen af hashtabellen, i den tidligere metode fra Cormen var dette m.

Da vi antager at universet er lavet af tal, kan vi konvertere de tal til binære tal á $log_2 p$ bits. D.v.s, at hvis $p = 11$, og du har tallet 85, som du gerne vil hashe. $log_2(11) = 3.45 \approx 4$. Vi tager 85 i binær: 1010101. Længden af det binære tal er kun 7 ciffrer, derfor tager vi og tilføjer et 0 først (da dette ikke ændrer på tallet). Dermed bliver det 01010101 som har 8 ciffrer. Vi kan dermed dele det op i 2, så vi har en vektor der hedder $(0101, 0101)$. Vi konverterer dette tilbage til heltal, $(5, 5)$

Hvor mange bits skal vi bruge til at repreæsenterer et tal af størrelse N? $\log_2 N$. Hvor mange stykker af længde $log_2 P$ kan du lave? $\frac{\log_2 N}{\log_2 p} \approx r \approx \frac{\log N}{\log n}$. Det er stadig log 2, jeg er bare doven.

Lad $A = \{(a_1, a_2, \ldots, a_r) | a_i \in \{0, 1, 2, \ldots, p-1\} \forall i \in [r]\}$

For $a \in A$ lad

$$h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \mod p$$

$$\mathcal{H} = \{h_a | a \in A\}$$

Theorem 18. *$\mathcal{H}$ er en universal hashfamilie.*

Bevis. Lad $x = (x_1, x_2, \ldots, x_r)$ og $y = (y_1, y_2, \ldots, y_r)$ være distinkte elementer fra U.

Hvad vi nu vil vise er at når $a = (a_1, a_2, \ldots, a_r) \in A$ er valgt tilfældigt, så $p(h_a(x) = h_a(b)) \leq \frac{1}{p}$. Hvis det er højest $1/p$ er det også højest $1/n$, da $p > n$.

Da $x \neq y$ er der et $j \in [r]$ således at $x_j \neq y_j$, altså, der **må** være en koordinat hvorpå de er uenige.

Vi bruger følgende måde at vælge et tilfældigt $a \in A$ på: Først vælg alle $a_i, i \neq j$. Så vælg a_j.

Vi vil nu bevise at for hvert valg af a_i hvor $i \neq j$, så er sandsynligheden for at det sidste valg af a_j ender med $h_a(x) = h_a(y)$ er præcis $\frac{1}{p}$.

$$\begin{aligned} h_a(x) &= h_a(y) \\ \sum_{q=1}^{r} a_q x_q &= \sum_{q=1}^{r} a_q y_q \mod p \\ \sum_{q=1}^{r} a_q (x_q - y_q) &= 0 \mod p \\ \sum_{q \neq j} a_q (x_q - y_q) + a_j (x_j - y_j) &= 0 \mod p \\ \sum_{q \neq j} a_q (x_q - y_q) &= a_j (y_j - x_j) \mod p \end{aligned} \tag{2}$$

Grunden til vi skriver $\mod p$ til sidst, er fordi, hvis $a \mod p = b \mod p$ så $a = b \mod p$.

Efter vi har fikset a_i for $i \neq j$ har vi $\sum_{q \neq j} a_q (x_q - y_q) = s \mod p$ for some $s \in \{0, 1, 2, \ldots, p-1\}$

Dermed $h_a(x) = h_a(y)$ hvis og kun hvis $a_j(y_j - x_j) = s\ modp$ da $z = y_j - x_j \neq 0$ siden vi har sikret os at $x_j \neq y_j$ har ligningen $a_j(y_j - x_j) = s \mod p$ en unik løsning $a_j = s(y_j - x_j)^{-1} \mod p$ Vi ved at z ikke er 0, da vi antog at $i \neq j$.

a_j får en tilfældig værdi i $\{0, 1, 2, \ldots, p-1\}$ når $a = \{a_1, a_2, \ldots, a_r\}$ bliver konstrueret. Dermed er sandsynligheden at $a_j = s \cdot (y_j - x_j)^{-1}$ mod p holder (og dermed $h_a(x) = h_a(y)$) $1/p$. Q.E. FUCKING D. $\square$

7.3 Perfect Hashing

Målet med Perfect Hashing er at få en virkelig god worst case behavior. Vi kan dog kun få det når nøglerne er statiske, så når tabellen er blevet lavet, kan nøglerne ikke ændres. Dette kan eksempelvis bruges i CD/DVD.

Vi ønsker at få $O(1)$ memory access i worst case (dvs. konstant tid selv i worst case), og et lavt memory use.

Perfect hashing bruger to niveauer af hashing hvor begge bruger universal hashing. Så først hashes der til tabel 1, og derefter hashes der til tabel 2, frem for en linked list.

Ved level et finder vi et nøjagte valgt hash funktion $h \in \mathcal{H}$ når h er universal.

I stedet for at bruge en linked list bruger vi en sekundær hashtabel S_j sammen med en associeret hashfunktion h_j for at undgå kollisioner i level 2. Størrelsen af n_j vil være n_j^2 hvor $n_j = |\{x \in S | h(x) = j\}|$

Ved level 1 bruger vi $h \in \mathcal{H}_{pm}$ når $p > |S|$ $(S \subseteq \{0, 1, 2, \ldots, p-1\})$ (dette er familien af hash funktioner defineret fra Cormen)

Nøgler hvor $h(x) = j$ bliver hashet til tabellen S_j af størrelsen m_j ved brug af $h_j \in \mathcal{H}_{pm_j}$

Vores første mål er at vi skal blive sikre på at der er ingen collisions på level 2. Derefter skal vi vise at de forventede hukkommelsesbrug er $O(n), n = |S|$.

Theorem 19. *Antag at vi lagrer n distinkte nøgler i en hashtabel af størrelse $m = n^2$ ved brug af en tilfældig $h \in \mathcal{H}$, når $\mathcal{H}$ er universal. Så er sandsynligheden at der er **ingen** kollisioner mindst $1/2$.*

Bevis. Der er $\binom{n}{2}$ mulige kollisioner. Lad $Z_{kl} = \begin{cases} 1 & \text{hvis } h(k) = h(l) \\ 0 & \text{ellers} \end{cases}$

Da h er universal, gælder det at $p(Z_{kl} = 1) \leq \frac{1}{m} = \frac{1}{n^2}$ når $k \neq l$.

Så $Z = \sum_{k,l \in S, k \neq l} Z_{kl}$ er antallet af kollisioner.

Vi bruger naturligvis vores elskede linearity of expectation til dette.

$$
\begin{aligned}
E(Z) &= E(\sum_{k,l \in S, l \neq s} Z_{kl}) \\
&= \sum_{k,l \in S, k \neq l} E(Z_{kl}) \\
&\leq \sum_{k,l \in S, k \neq l} \frac{1}{n^2} \\
&= \frac{\binom{n}{2}}{n^2} \\
&< \frac{1}{2}
\end{aligned} \tag{3}
$$

Vi vil så gerne finde sandsynligheden for at antallet af kollisioner er større end 1. Det gør vi ved hjælp af **Markov's Inequality** (Teorem **??**):

$$p(Z \geq 1) \leq \frac{E(Z)}{1} = E(Z) < \frac{1}{2}$$

Dermed er chancen for at der er 0 kollisioner større end $\frac{1}{2}$. □

Så, hvor mange gange skal vi køre algoritmen, før vi finder noget uden nogen kollisioner? I gennemsnit vil det være: $\frac{1}{p(Z=0)} < \frac{1}{\frac{1}{2}} = 2$

Vi møder dog et problem nu. **Hvad hvis n er stort, og n^2 så er for stort?**. F.eks., hvis $n = 1000$, så er $n^2 = 1000000$, og det er heller ikke usandsynligt at $n > 1000000$; så der kan du begynde at se nogle store problemer.

Vi **løser** dette problem ved at udelukkende bruge størrelsen at tabellen n^2 til andet niveau, og lade det første niveau være $n = m$.

Lad $h \in \mathcal{H}$ være hash funktionen vi bruger på level 1.

Lad $n_j = |\{x|h(x) = j\}|$[1] og lad $S_j, j \in [m]$ være en tabel med n_j^2 entries og h_j en kollisionsfri hash funktion der mapper fra $\{x|h(x) = j\}$ til S_j.

Ved level 1, når vi antager at $m = n$, altså, størrelsen af tabellen er lig n, så bruger vi $O(n)$ hukommelse til at lagre:

- Den primære hashtabel (da der er n slots)

[1] Altså, antallet af elementer der hasher til værdi j

- Tallene $m_j = n_j^2 \quad j \in [m]_0$
- $a_j \in \mathbb{Z}_p^*, b_j \in \mathbb{Z}_p$ hvilke definerer det andet andet niveau af hash funktionen n_j som bliver brugt når $\{x|h(x) = j\}$.

Theorem 20. *Antag at vi lagrer n nøgler i en hash funktion af størrelse $m = n$ ved brug af universal hashing, og lad $n_j, j \in \{0, 1, 2, \ldots, m\}$ være antallet af nøgler der bliver hashet til j $(h(x) = j)$. Så $E\left(\sum_{j=0}^{m-1} n_j^2\right) < 2n$ (n_j er en random variable der afhænger af valget af h)*

Bevis. Husk at $\forall a \in \mathbb{Z}^+ \quad a + 2\binom{a}{2} = a + a(a-1) = a^2$

$$\begin{aligned}
E\left(\sum_{j=0}^{m-1} n_j^2\right) &= E\left(\sum_{j=0}^{m-1} n_j + 2\binom{n_j}{2}\right) \\
&= E\left(\sum_{j=0}^{m-1} n_j\right) + 2E(\sum_{i=0}^{m-1} \binom{n_j}{2}) \qquad (4) \\
&= E(n) + 2E(r) \\
&= n + 2E(r)
\end{aligned}$$

Hvor r er der totale antal kollisioner når vi bruger $h \in \mathcal{H}$, som altså er lig $\sum_{i=0}^{m-1} \binom{n_j}{2}$

Fordi vi bruger universal hashing gælder det at $E(r) \leq \binom{n}{2} \cdot \frac{1}{m} = \binom{n}{2}\frac{1}{n} = \frac{n-1}{2}$ Dermed

$$E\left(\sum_{j=0}^{m-1} n_j^L\right) \leq n + 2 \cdot \frac{n-1}{2} < 2n$$

□

Corollary 21. *Hvis du vælger et hashing scheme således at $m = n$ på level et og $m_j = n_j^2 \;\; j \in \{0, 1, \ldots, n-1\}$ på level 2, så er det forvetenede plads brugt p åden sekundere hash tabel mindre end 2n.*

Bevis.

$$E(\sum_{j=0}^{m-1} m_j) = E\left(\sum_{j=0}^{m-1} n_j^2\right) < 2n$$

□

Corollary 22. *Ved brug af at hashing scheme som det nævnt før, er sandsynligheden for at vi har brug for mere end* $4n$ *hukoemmelse i alt for det andet niveau af hash tabeller mindre end* $\frac{1}{2}$.

Bevis. Vi Bruger markov's inequality:

$$p\left(\sum_{j=0}^{m-1} m_j \geq 4n\right) \leq \frac{E\left(\sum_{j=0}^{m-1} m_j\right)}{4n} < \frac{2n}{4n} = \frac{1}{2} \tag{5}$$

□

7.3.1 Konklusion

Ved at bruge få trials til at finde en god $h \in \mathcal{H}$ når $\mathcal{H}$ er universal, så kan vi hurtigt få et skema (h ved niveau 1 , $h_1, h_2, \ldots, h_{m-1}$ ved niveau 2) som bruger en fin mængde hukommelse.

7.4 Count-min Sketch

Antag at S er en datastrøm hvor vi vil estimere frekvenserne af elementerne som forekommer ofte i S, for eksempem til at løse heavy hitters.

- Lad b, l være heltal.
- Lad $\mathcal{H}$ være en univerel familie af hash funktioner, $h \in \mathcal{H}$ hasher $U \to [b]$ når U er universet af alle mulige elementer i strømmen.
- Lad $h_1, h_2, \ldots, h_l$ være dinstikte medlemmer fra $\mathcal{H}$
- Når vi siger at $h_i \in \mathcal{H}$ er universal, mener vi at h_i er et tilfældigt medlem af $\mathcal{H}$

Vi bruger $h_1, h_2, \ldots, h_l$ til at bygge en $l \times b$ array M af tællere som følger:

Til at starte med $M_{ij} = 0$ for $i \in [l]$ og $j \in [b]$.

For hvert element x i datastrømmen, processer vi det som følger:

1. Vi går igennem hver række l, og så finder vi hash-værdien af x, $h_l(x)$, hvis den f.eks., er 6, så increaser vi tælleren ved første række, 6. kolonne med 1.

Count-min sketch finder et upper bound på frekvensen. Det er et upper-bound, fordi den muligvis tæller nogle andre tal med.

Vi har set at $M_i, h_i(x)$[2] altid er mindst frekvensen af x of ofte højere. Dette er fordi:

(a) Hver occurence af x increaser $M_i, h_i(x)$ med en, så $M_i, h_i(x) \geq fx$ når fx er den rigtige frekvens af x indtil videre.

(b) Hver occurence af a $y \neq x$ med $h_i(x) = h_i(y)$ wil også increase $M_i, h_i(x)$

Lad S_n være de første n elementer i datastrømmen. Vi lader f_y være frekvensen af y i S_n. Vi lader $M_i, h_i(x)$ være $Z_{i,x}$. $Z_{i,x}$ er en random variable der afhænger af det tilfældige valg af $h_i \in \mathcal{H}$. Vi definerer indicatoer random variable $I_{i,x}$ som følger:

$$I_{i,x}(y) = \begin{cases} 1 & \text{hvis } h_i(x) = h_i(y) \\ 0 & \text{ellers} \end{cases}$$

Da h_i er en universal hash funktion er $p(I_{i,x}(y) = 1) \leq \frac{1}{b}$ hvor b er størrelsen på hash tabellen.

Det følger fra (a) og (b) at:

$$Z_{i,x} = f_x + \sum_{\{y \in S_n | y \neq x\}} f_y \cdot I_{i,x}(y) \geq f_x$$

Hvad er så den forventede værdi af $Z_{i,x}$?

Vi kommer til at bruge $\sum_{y \in S_n} f_y = n = |S_n|$

[2] Hvor M_i er række i i tabellen

$$
\begin{aligned}
E(Z_{i,x}) &= E(f_x + \sum_{\{y \in S_n | y \neq x\}} f_y \cdot I_{i,x}(y)) \\
&= E(f_x) + E(\sum_{\{y \in S_n | y \neq x\}} f_y \cdot I_{i,x}(y)) \\
&= f_x + \sum_{\{y \in S_n | y \neq x\}} f_y \cdot E(I_{i,x}(y)) \\
&\leq f_x + \sum_{\{y \in S_n | y \neq x\}} f_y \cdot \frac{1}{b} \qquad (6) \\
&\leq f_x + \frac{1}{b} \sum_{\{y \in S_n | y \neq x\}} f_y \\
&\leq f_x + \frac{1}{b} \sum_{y \in S_n} f_y \\
&= f_x + \frac{n}{b}
\end{aligned}
$$

Dermed er den forventede værdi "off" med **højest** $\frac{n}{b}$ (den forventede værdi!). Desværre, gennem dueslagsprincippet vil der være mange collisions, så længe $n > b$, hvilket den jo er.

Ved brug af Markov's Inequality vil vi nu gerne bounde sandsynligheden for at vores estimat for f_x er mere end $\frac{2n}{b}$ væk.

$$p(Z_{i,x} - f_x \geq \frac{2n}{b}) \leq \frac{E(Z_{i,x} - f_x)}{\frac{2n}{b}} = \frac{\frac{n}{b}}{\frac{2n}{b}} = \frac{1}{2}$$

Så, sandsynligheden for at vores estimat er mere end $\frac{2n}{b}$ væk er $\frac{1}{2}$, hvilket, er en ret stort sandsynlighed.

Hvad så hvis vi kun kigger på den hash funktion der giver os det tætteste estimat? Lad $\hat{f}_x = \min_{i \in [l]} Z_{i,x}$, så $\hat{f}_x \geq f_x$ og siden $h_1, h_2, \ldots, h_l$ er uafhængige af hinanden betyder det at:

$$p(\hat{f}_x - f_x \geq \frac{2n}{b}) \leq \frac{1}{2^l}$$

Antag at vi er givet værdier ε, δ og vi vil finde

$$p(\hat{f}_x - f_x \geq \varepsilon n) \leq \delta$$

Vi ved fra den tidligere ligning at hvis vi tager $b = \frac{2}{\varepsilon}$ og $l = \log_2\left(\frac{1}{\delta}\right)$ så

$$p(\hat{f}_x - f_x \geq \varepsilon n) = p(\hat{f}_x - f_x \geq \frac{2n}{b}) \leq 2^{-l} = 2^{-\log(\frac{1}{\delta})} = \frac{1}{\frac{1}{\delta}} = \delta$$

Så $p(\hat{f}_x - f_x \geq \varepsilon n) \leq \delta$

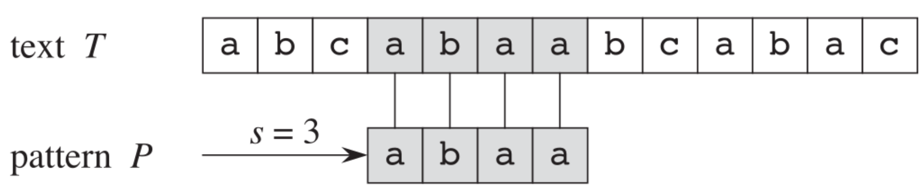
Vi bruger $b \cdot l = \frac{2}{\varepsilon} \cdot \log\left(\frac{1}{\delta}\right)$ tællere til at implementere sketchen og så får vi akkuratheden $p(\hat{f}_x - f_x \geq \varepsilon n) \leq \delta$, uanset længden af datastrømmen.

8 String Matching

8.1 Notation

Jeg tænker ikke at der skal snakkes om det her til eksamen, men følgende er en liste af notation der er nødvendige for forforståelse:

- **Strenge**: Arrays med karakterer (ligesom i programmeringssprog)
- **Shift**: Hvor langt inde i en streng
- $P[1..m]$: Mønster med længde m
- $T[1..n]$: Tekst med længde n
- $T[1..n-m]$: Den tekst vi leder efter. Vi er ikke interesseret i de sidste m, da de er længere end mønsterstrengen.
- **P forekommer med shift s**: Du finder mønstret s karakterer inde i teksten.
- **Validt shift** et shift hvor mønsteret P forekommer
- **Invalidt shift** et shift hvor mønsteret P **ikke** forekommer.



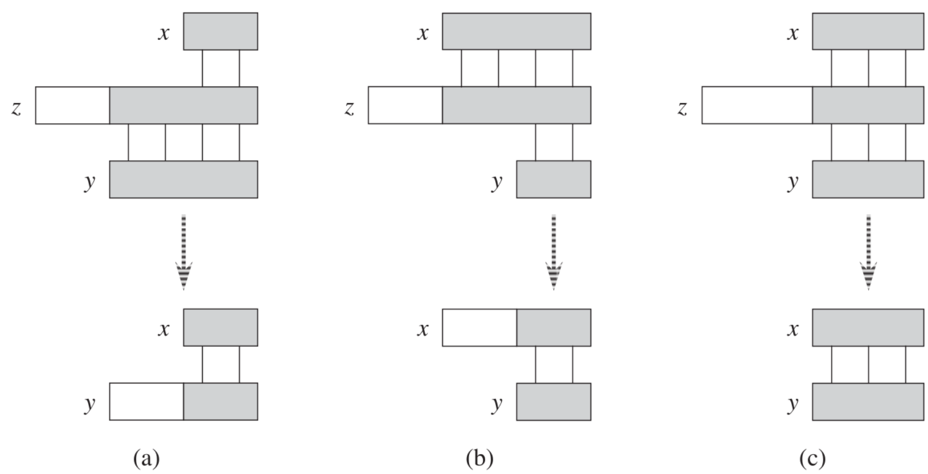
- Σ^* (Sigma-Stjerne) er sættet af alle endelige strenge der kan bliver lavet fra karaktererne i Σ.
- ε, den **tomme streng**, er strengen uden noget indhold. Den er også en del af Σ^*.
- $|x|$ er længden af streng x.
- **Concatenation** af to strenge x og y, skrevet xy har længde $|x|+|y|$ og er karaktererne i x efterfulgt af karaktererne i y.

- **Præfiks** af streng x, denoted $w \sqsubset x$, gælder hvis $x = wy$ hvor $y \in \Sigma^*$, altså, w er en del af streng x i starten af strængen. y er den resterende del af streng x, som ikke er w.

- **Suffiks**: denoted $w \sqsupset x$ omvendt.

Lemma 23 (31.1 (Overlapping-suffix lemma) (Cormen))**.** *Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$ then $x = y$.*

Bevis. Se Figur 1 □



Figur 1: Vi antager at $x \sqsupset z$ og $y \sqsupset z$. De tre dele af figuren illustrerer de tre cases af lemmaet. **(a)** Hvis $|x| \leq |y|$, så $x \sqsupset y$. **(b)** Hvis $|x| \geq |y|$, så $y \sqsupset x$. **(c)** Hvis $|x| = |y|$, så er $x = y$.

Vi antager at tiden det tager for at finde ligheden mellem to strenge er $\Theta(t+1)$ hvor t er størrelsen af den længste streng. $+1$, til hvis $t = 0$.

8.1.1 Køretids Overview

8.2 Naive Algoritme

- Hvad er den?

- Hvorfor er den dårlig?

Algorithm	Preprocessing Time	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

- Hvad er worst-case?

Den naive algoritme er virkelig det, naiv. **Source Code**:

```
Naive-String-Matcher(T,P)
n = T.length
m = P.length
for s = 0 to n - m
    if P[1..m] == T[s+1..s+m]
        print "Pattern occurs with shift " s
```

8.2.1 Køretid

Den er virkelig skrald. Køretiden er $O((n-m+1)m)$. Dens worst case sker hvis teksten er a^n og mønsteret der ledes efter er a^n (begge er mængder af a'er, på længde hhv. m og n. I dette tilfælde finder den matches hver gnag, og der tager dermed $O(n^2)$ tid.

Der er **ingen** preprocessing tid, da der ikke skal gøres noget før algoritmen kører.

8.3 Rabin-Karp

- Hvad er hovedidéen?
- Hvorfor er den bedre end naive?

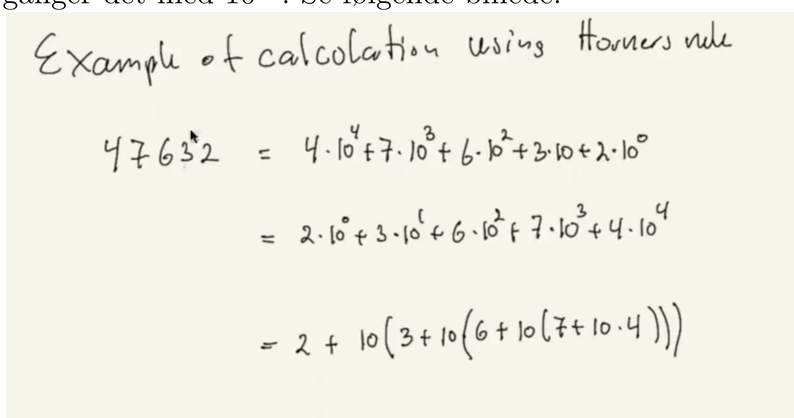
Trods at Rabin-Karp har en worst-case køretid på $\Theta((n-m+1)m)$ er dens gennemsnitlige køretid bedre.

Algoritmen konverterer bogstaverne til tal, i radix-d notation, hvor d er størrelsen på alfabetet, $|\Sigma|$.

I følgende eksempler vil vi gå ud fra at $d = 10$, og $\Sigma = \{0, 1, \ldots, 9\}$. Husk at $P[1..m]$ er mønsteret vi leder efter. Ved rabin-karp skelner vi mellem $P[1..m]$ og p, hvor p er dets decimalværdi. Dvs., hvis $P[1..m] = 1372$, så er $p = 1372$ i decimalværdi. Eksemplet virker forsimplet idet vores alfabet også er tal, men tænk hvis alfabetet var $\Sigma = \{a, b, ...j\}$, i dette tilfælde ville p ikke være ændret, men $P[1..m] = acgb$. Ydermere er teksten $T[1..n]$'s decimal counterpat t_s. Den bliver udregnet på samme måde. Hvis $t_s = p$ så $T[s+1..s+m] = P[1..m]$.

Vi vil gerne have en måde hvorpå vi kan lave alfabetet om til tal, som vi kan regne på. Hvis vi kan konverterer mønsteret $P[1..m]$ til p på $\Theta(m)$ tid, så kan vi konvertere t_s på $\Theta(n-m+1)$ tid. Til at gøre dette bruger vi **Horner's Rule**, som er meget vigtig at kende, se Definition 1.

Definition 1 (Horner's Rule)**.** Horner's Rule er en regel hvorpå du hurtigt (specielt for computere) kan udregne decimaltal. Dette gør du ved at tage det sidste tal der skal udregnes først, derefter tager du tallet på 10'ernes plads, ganger det med 10^1, etc. indtil du er ved d'ende plads, og ganger det med 10^d[3]. Se følgende billede.



Noget af det smarteste med Horner's Rule, er at, når du går til næste værdi, så kan du udregne det hurtigt uden at tage det hele om igen. Dette giver køretid $\Theta(n-m)$:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Forklaring 24. *Skip dette hvis du ikke har meget tid.* $10^{m-1} \cdot T[s+1]$ *fjerner det højeste ciffer. Ved at gange det med 10 skifter du tallet til venstre med en cifferposition. Ved at tilføje* $T[s+m+1]$ *får du det nye, laveste ciffer.*

[3] Dette gælder kun i base-10. Rabin-karp kører i base-b. Konverter dette til b^d

Problem! p og t_s er muligvis for storre til at de kan være i et computer `word`. Hvis dette er tilfældet, og P indeholder m karakterer, så tager vi tallet **modulo** q. $p \mod q$ bliver udregnet på $\Theta(m)$ tid (størrelsen af p.) Alle t_s værdier i $\Theta(n-m+1)$ tid.

Hvilken q skal vi dog vælge? Simpelt! Vælg et primtal således der er plads til $10q$ i én computer `word`. Derefter kan vi udføre alle udregninger simpelt. Ved at bruge modulo-udregning, ændrer Horner's udregning sig til at blive: $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q$, hvor $h \equiv d^{m-1} (\mod q)$ er værdien af ciffret 1 i højeste position.

Problem igen! Hvad hvis $p \equiv t_s$, men $P[1..m] \neq T[s+1..n-m]$? Altså, tallene er ens, men de er strengene ikke grundet modulo? Dette kalder vi et **spurious hit**, og er pisse irriterende, men desværre end nødvendig onde. Derfor, når vi finder et **hit** om det er spurious eller ej, så tjekker vi også strengene.

```
RABIN-KARP-MATCHER(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m                      // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (dt_0 + T[i]) mod q
for s = 0 to n - m                  // matching
    if p == t_s
        if P[1..m] == T[s+1..s+m]
            print "Pattern occurs with shift" s
    if s < n - m
        t_(s+1) = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

Figur 2: Rabin Karp Algoritmen

Worst-case er i samme situation som ved den naive algoritme. Hvis teksten er en del a'er, og det samme med mønstret, så vil vi få en masse hits.

8.3.1 Forventede antal hits

Vi vil gerne finde det forventede antal hits. Vi antager at $\mod q$ agerer som en tilfældig mapping (funktion) fra alfabetet til heltal base q, altså $\Sigma^* \rightarrow \mathbb{Z}_q$. Ydermere antager vi at alle værdierne modulo q er lige sandsynlige, i.e, $p(t_s \equiv p \mod q) = \frac{1}{q}$. Det vil sige at antallet af falske hits er $\frac{O(n)}{q} = O(\frac{n}{q})$. Dette får vi fra antal af hvor mange der modulerer til samme værdi. Hvis der er 10 forskellige bogstaver, og vi er i base-3, så $\frac{10}{3} = 3.\overline{3}$ ca. tal der mapper til det samme.

Den forventede køretid bliver derfor $O(n) + O(m(v + \frac{n}{q}))$ hvor v er antallet af korrekte hits, det's køretid er $O(1)$ og $q \geq m$. Dermed bliver den totale køretid $O(n + m) = O(n)$ da $n \geq m$.

8.4 Finite Automaton Based

- Hvordan laver man en DFA?

Jeg tænker **ikke** du behøver at forklare hvad en DFA er osv. Du får her en kort introduktion, som du bare kan springe over, givet at du forstår finite automata fint.

8.5 Introduktion

Mange algoritmer bygger en Finite Automata (herfra forkortet som DFA), da den er utroligt hurtig til at finde matches. Hver karakter bliver kigget på præcis én gang, og bruger tid $O(1)$ per gang den bliver kigget på. Efter maskinen bliver bygget er matching tiden $\Theta(n)$. Dog kan tiden der bruges til at bygge maskinen være meget stor hvis Σ er stort.

Definition 2 (Finite Automata)**.** A **finite automaton** M, is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of **states**
- $q_0 \in Q$ is the **start state**
- $A \subseteq Q$ is a disntinguished set of **accepting states**,

- Σ is a finite **input alphabet**,
- δ is a function from $Q \times \Sigma$ into Q called the **transition function** of M.

Ved hvert state, læser maskinen et input, og går fra den state, q, til den næste defineret state, $\delta(q, a)$. Hvis q er en del af A, og, efter strengen er blevet "spist", ender maskinen i $q \in A$, så er strengen "accepteret", ellers er den ikke.

Ydermere bliver funktionen ϕ defineret som **Final-state funktion** fra Σ^* til Q således at $\phi(w)$ er den state som M ender op i, efter den scanner strengen w. Så, M accepterer streng w hvis **og kun hvis**, $\phi(w) \in A$.

$$\begin{aligned} \phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma \end{aligned} \tag{7}$$

8.6 Streng-matchende automat

For at givet mønster P, vil vi lave en streng-matchende automat som preprocessing skridt før den bruges til at søge efter strengen. Se figur 3 for hvordan vi konsturerer automaten for mønstret $P = ababaca$.

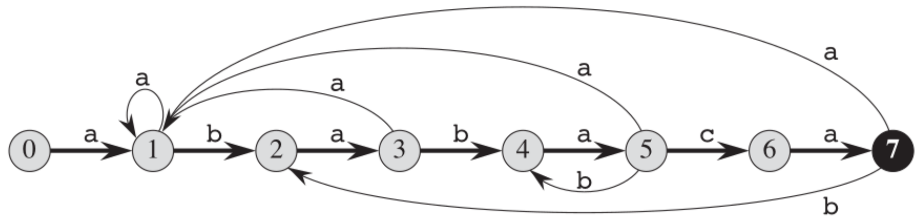
Definition 3 (Suffiks Funktion)**.** Givet et mønster $P[1..m]$, definerer vi funktionen σ, kaldet **suffiks funktion** korresponderende til P. Funktionen σ mapper Σ^* til $\{0, 1, \ldots, m\}$, således at $\sigma(x)$ er længden af det længste præfix af P som også er et suffiks af x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

Suffiks Funktionen er **well-defined** (hvert element mapper til noget), da $P_0 = \varepsilon$ er et suffiks er hver streng.

Example 1 (Eksempler på Suffiks Funktionen)**.** Givet mønstret $P = ab$, har vi $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$ og $\sigma(ccab) = 2$. Givet mønstret med længde m har vi $\sigma(x) = m$ hvis og kun hvis $P \sqsupset x$. Fra definitionen af suffiksfunktionen betyder $x \sqsupset y$ også at $\sigma(x) \leq \sigma(y)$.

Vi definerer en streng-matchende automat som korresponderende til et mønster $P[1..m]$ som følger:



(a)

	input			
state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figur 3: For mere information om figuren, se pp. 997 i Cormen

- Sættet af states Q er $\{0, 1, \ldots, m\}$. Start staten q_0, og staten m er de eneste accepteret states.
- Transition function δ er defineret ved følgende ligning, for hver state q og karakter a:

$$\delta(q, a) = \sigma(P_q a) \tag{8}$$

Det vil sige, at givet en karakter a, vil vi gå fra state q til længden af det længste præfiks af P, som også er et suffiks af x. Så, altså, hvis a får dig én længere, vil du også gå én state tilbage. Men, hvis du går tilbage til kun at være 2 inde, så er du tilbage på state 2.

Forklaring 25 (Yderligere forklaring)**.** *Vi definerer $\delta(q,a) = \sigma(P_q a)$ fordi vi vil holde fast i det længste præfix af mønsteret P der har matchet strengen T indtil videre.*

Antag at $p = \phi(T_i)$, så, efter at have læst T_i, så er automatonet i state q. Vi designer δ således at state q fortæller os længden af det længste præfiks af P der er en suffiks af T_i. Det vil sige, i state q, $P_q \sqsupset T_i$ og $q = \sigma(T_i)$. Dette vil også sige at ***hvis*** *$q = m$***, så har vi fundet et match!*** *Dermed, siden $\phi(T_i)$ og $\sigma(T_i)$ begge er lig q, ser vi at automaten holder følgende invariant:*

$$\phi(T_i) = \sigma(T_i) \tag{9}$$

Dermed, hvis vi er i state q, og automaten læser karakter $T[i+1] = a$, så skal vores transition lede til det korresopnderende længste præfiks af P som er et suffiks af $T_i a$. Den state er $\sigma(T_i a)$.

Fordi P_q er det længste præfiks af P som er et suffiks af T_i, så er det længste præfiks af P som er et suffiks af $T_i a$ ikke kun $\sigma(T_i a)$, men også $\delta(P_q a)$. (Dette bliver bevist senere)

Der er to states vi skal kigge på, den første, $a = P[q+1]$, så er $\delta(q,a) = q+1$. Den næste, $a \neq P[q+1]$, så skal vi finde et mindre præfiks af P som også er et suffiks af T_i.

Lad os kigge på et eksempel. Streng-matching automaten fra Figur 3 har $\delta(5,c) = 6$, som så er first case, hvor vi bare går videre. Et sekmepl på second case er $\delta(5,b) = 4$. Vi laver denne transition fordi, hvis automaten læser et b når $q = 5$, så $P_q b = ababab$, og det længste præfiks af P som også er et suffiks af $ababab$ er $P_4 = abab$.

Følgende er algoritmen for at lave en finite automata til streng-matching. Sættet af states $Q = \{0, 1, \ldots, m\}$, start staten $q_0 = 0$, den eneste accepting state er m, $m \in A$.

Transition function er som beskrevet tidligere. Hvis dette ikke er klart, se Forklaring 25.

Det er her nemt at se at køretiden på en tekst-streng af længde n er $\Theta(n)$. Før vi viser pre-processing tid, kigger vi på et bevis for at algoritmen kører som forventet.

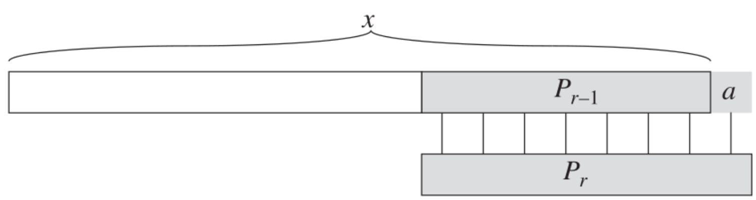
Lemma 26 (Suffix-Function Inequality)**.** *For hver streng x og karakter a, har vi at $\sigma(xa) \leq \sigma(x) + 1$.*

Bevis. Se Figur 5. Hvis $r = 0$, så $\sigma(xa) = r \leq \delta(x) + 1$ er trivielt løst, da $\sigma(x)$ er nonnegativt. Antag at $r > 0$, så $P_r \sqsupset xa$ per definition af

```
Finite-Automaton-Matcher(T, d, m):

n = T.length
q = 0
for i = 1 to n
    q = d(q, T[i])
    if q == m
        print "Pattern occurs with shift" i - m
```

Figur 4:



Figur 5: En illustration til beviset af Lemma 26. FIguren viser $r \leq \delta(x) + 1$, hvor $r = \delta(xa)$

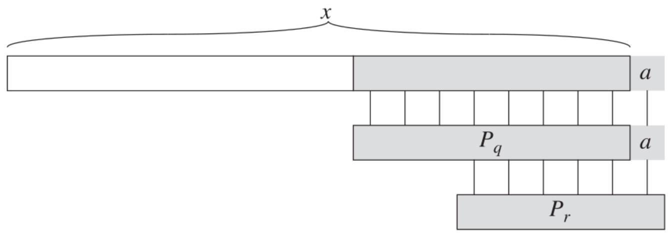
σ. Dermed, $P_{r-1} \sqsupset x$, ved at fjerne a fra enden af P_r og fra enden af xa. Dermed $r - 1 \leq \sigma(x)$, siden $\sigma(x)$ er det største k således $P_k \sqsupset x$, og således $\sigma(xa) = r \leq \sigma(x) + 1$ □

Lemma 27 (Suffix-Function Recursion Lemma). *For enhver streng x og karakter a, hvis $q = \sigma(x)$, så $\sigma(xa) = \sigma(P_q a)$.*

Bevis. Vi ved fra definitionen af σ at $P_q \sqsupset x$. Som vist i figur 6, har vi også $P_q a \sqsupset xa$. Hvis $r = \sigma(xa)$, så $P_r \sqsupset xa$ og, gennem Lemma 26, $r \leq q + 1$. Dermed har vi at $|P_r| = r \leq q + 1 = |P_q a|$. Derfor, $r \leq \sigma(P_q a)$, dermed $\sigma(xa) \leq \sigma(P_q a)$. Vi har dog også $\sigma(P_q a) \leq \sigma(xa)$, siden $P_q a \sqsupset xa$. Dermed $\sigma(xa) = \sigma(P_q a)$ □

Tid til det vigtigste skridt. Vi skal vise at automaten vedligeholder invarianten i ligning 8.

Theorem 28. *If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1..n]$ is an input text for the automaton, then $\phi(T_i) = \sigma(T_i)$ for $i = 0, 1, \ldots, n$.*



Figur 6: Illustration af beviset for Lemma 27. Figuren fiser at $r = \delta(P_q a)$, hvor $q = \sigma(x)$ og $r = \sigma(xa)$

Bevis. Vi beviser gennem induktion på i.

Ved $i = 0$ er teoremet sandt, da $T_0 = \varepsilon$ dermed $\phi(T_0) = 0 = \sigma(T_0)$. □

Vi antager nu at $\phi(T_i) = \sigma(T_i)$ og beviser at $\phi(T_{i+1}) = \sigma(T_{i+1})$. Lad q være $\phi(T_i)$, og lad a være $T[i+1]$. Så:

$$\begin{aligned}
\phi(T_{i+1}) &= \phi T(_i) \text{ (fra definition på } T_{i+1} \text{ og a)} \\
&= \delta(\phi(T_i), a) \text{ (fra definitionen på} \phi) \\
&= \delta(q, a) \text{ (af defintiion på q)} \\
&= \sigma(P_q a) \text{ (fra definitionen tidligere)} \\
&= \sigma(T_i a) \\
&= \sigma(T_{i+1})
\end{aligned}$$

8.7 Find Transition Function

Vi vil gerne finde transition funktion. Vi har allerede defineret den tidligere, men vi vil have en algoritmisk metode hvorpå vi kan gøre det.

Køretiden på algoritmen er $O(m^3|\Sigma|)$.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m = P.length
for q = 0 to m
    for each character a ∈ Σ
        k = min(m + 1, q + 2)
        repeat
            k = k − 1
        until P_k ⊐ P_q a
        δ(q, a) = k
return δ
```

Figur 7: Algoritmen for at finde transition function på.

9 Flows

Et **netværk** $N = (V, E, C)$ er en digraph med en associeret funktion, kapacitetsfunktionen $c : E \to \mathbb{R}_0$ $(c(u, v) \geq 0 \forall (u, v) \in A)$. I et netværk, hvis $(u, v) \notin E$, så $c(u, v) = 0$. Ydermere er der en assumption i Cormen, om at parallele grafer ikke er tilladt, altså, hvis $(u, v) \in E$, så $(v, u) \notin E$.

Jørgen's definition af flow er mere generel end den i Cormen:

Theorem 29 (Flow)**.** *Et **flow** f i N er en funktion $f : E \to \mathbb{R}_0$ således at* $0 \leq f(u, v) \leq c(u, v) \forall (u, v) \in E$

Theorem 30 (Balance)**.** ***Balancen*** b_f *af et flow* f *er funktionen*

$$b_f(v) = \sum_{(v,w) \in E} f(v, w) - \sum_{(u,v) \in E} f(u, v)$$

Altså, **balancen** af et flow er mængden af flow der kommer ud af en knude (v) minus mængden af flow der kommer ind.

Vi kan her lave den observation at, hvis vi summerer alle balancer i flows, må deres sum blive 0, i.e., $\sum_{v \in V} b_f(v) = 0$.

Bevis. $b_f(v) = \sum_{(v,w) \in E} f(v, w) - \sum_{(u,v) \in E} f(u, v)$ så i $\sum_{v \in V} b_f(v)$ bidrager hver kant (u, v) med $f(u, v)$, og $b_f(v)$ og, $-f(u, v)$ i $b_f(u)$ så 0 i alt. $\square$

Definition 4. Lad $N = (V, E, c)$ være et netværk, og lad $s, t \in V$ være distinkte punkter. Et flow f i N er et $(s,t)-$flow, hvis der er et $K \geq 0$ således at

$$b_f(v) = \begin{cases} k & \text{hvis } v = s \\ -k & \text{hvis } v = t \\ 0 & \text{hvis } v \notin \{s, t\} \end{cases}$$

Altså, *source* knuden har balance lig med flow, da der ikke kommer noget ind, og *sink* knuden har balance lig med minus flow, da intet kommer ud. Dette gælder fordi et $(s,t)-$flow overholder **flow conservation**, hvilket vil sige at hvad der kommer ind, må også komme ud, og omvendt.

Definition 5. Værdien af et (s,t)-flow f i $N = (V, E, c)$ er skrevet $|f|$ og er defineret til at være

$$|f| = \sum_{u \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Altså, alt det flow der kommer ud fra source knuden, versus det der kommer ind. Dermed er det det samme som $b_f(s)$, og $-b_f(t)$.

Definition 6. Maksimum-flows problemet på et netværk $N = (V, E, c)$ med s, t skal man maksimere K således at

$$b_f(v) = \begin{cases} k & \text{hvis } v = s \\ -k & \text{hvis } v = t \\ 0 & \text{hvis } v \notin \{s, t\} \end{cases}$$

$$0 \leq f(u, v) \leq c(u, v) \quad \forall (u, v)$$

Jørgen skelner mellem **maximum** og **maksimalt** flow. Et maksimalt flow kan ikke increases længere, **men** det er ikke maximum! Et maximumsflow er der ingen måder hvorpå værdien af $(s,t)-$flowet kan blive større.

Vi er nu efterladt med to spørgsmål:

- Hvordan ved vi at et flow er maximum?
- Hvordan finder vi et flow der er maximum?

Cuts

Definition 7. Lad $N = (V, E, c)$ være et netværk med source s og sink t. Et $(s,t)-$cut er en partition $V = S \cup T$ hvor $T = V \setminus s$ og $s \in S, t \in T$. Kapaciteten af $(s,t)-$cut (S,T) er $c(S,T) = \sum_{u \in S, v \in T} c(u,v)$

Lemma 31. *Lad $N = (V, E, c)$ være et netværk og f et (S,T)-flow i N. Så, for hvert (s,t)-cut (S,T) i N, har vi at:*

$$|f| = f(S,T) - f(T,S)$$

Bevis.

$$\begin{aligned}
|f| &= \sum_{v \in S} b_f(v) \\
&= \sum_{v \in S} \left(\sum_{(v,w) \in E} f(v,w) - \sum_{(u,v) \in E} f(u,v) \right) \qquad (10) \\
&= \sum_{v \in S} \sum_{w \in T} f(v,w) - \sum_{u \in T, v \in S} f(u,v) = f(S,T) - f(T,S)
\end{aligned}$$

□

Lemma 32. *For hvert (s,t)-cut (S,T) i $N = (V, E, c)$ og hvert (s,t)-flow f i N, har vi at*

$$|f| \leq c(S,T)$$

Bevis.

$$\begin{aligned}
|f| &= f(S,T) - f(T,S) \\
&\leq c(S,T) - 0 \qquad \text{da } f(u,v) \leq c(u,v) og f(u,v) \geq 0 \\
&= c(S,T)
\end{aligned}$$

□

9.1 Residual Networks

Lad $N = (V, E, c)$ og lad f være et $(s,t)-$flow i N. **Residual Netværket** N_f af N med respekt til f er $N_f = (V, E_f, c_f)$, når

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{hvis } (u,v) \in E \\ f(u,v) & \text{hvis } (v,u) \in E \\ 0 & \text{ellers} \end{cases}$$

Husk at vi antager ingen anti-parallele kanter. Ydermere er $E_f = \{(u,v) \in V \times V | c_f(u,v) > 0\}$, altså, kanterne i residualnetværket er kanter hvis residual kapacitet er større end 0.

Hvad skal vi bruge et residualnetværk til? Simpelt! Vi bruger det til at finde vejene til et maximum flow.

Hver direkted (s,t)-path (vej) i N_f korresponderer til en vej i N. Lad

$$\begin{aligned} \delta_-(p) &= \min\{c(u,v) - f(u,v) | (u,v) \text{ er fremadgående på } P'\} \\ \delta_+(p) &= \min\{f(u,v) | (u,v) \text{ er tilbagegående på } P'\} \qquad (11) \\ \delta(p) &= \min\{\delta_+(p), \delta_-(p)\} \end{aligned}$$

Dette resultat er skrevet $(f \uparrow f_p)$, og er et (s,t) flow af værdi $|f| + |f_p| = |f| + \delta(p)$

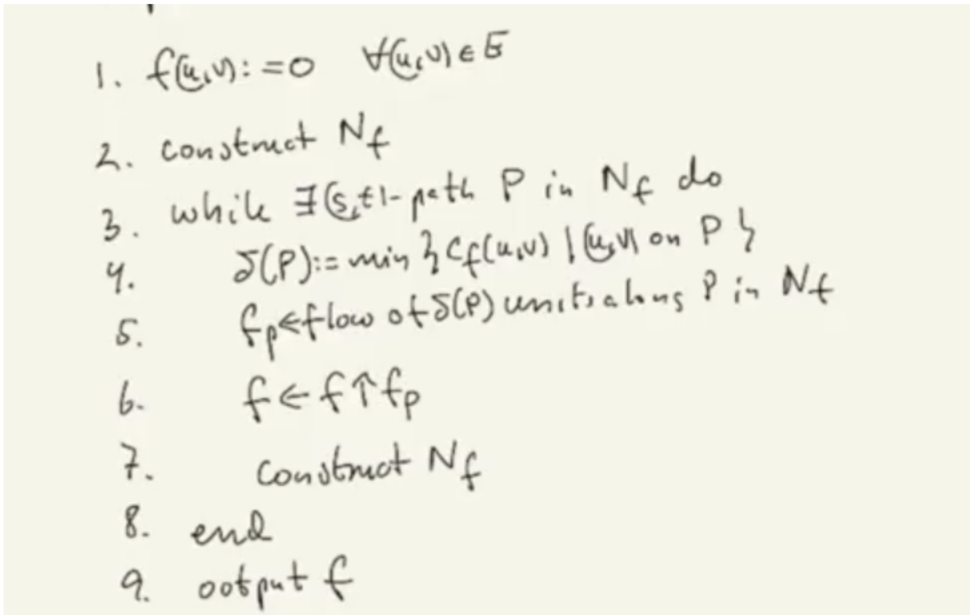
Hovedidéen er simpel:

- $0 \leq (f \uparrow f_p)(u,v) \leq c(u,v)$ af definitionen på $\delta(p)$
- $(f \uparrow f_p)$ er et $(s,t)-$flow siden we tilføjer det samme mængde flow i hver $v \neq s,t$ som ud af det.
- $|f \uparrow f_p| = |f| + |f_p| = |f| + \delta(p)$ da vi increase flowet med $\delta(p)$ på præcis en ud af s.

Vi kalder en directed (s,t)-vej P i N_f en **augmenting path** og dens kapacitet er værdien $\delta(p)$ som vi udregnede.

9.2 Ford-Fulkerson

Ford-fulkerson er en metode (ikke algoritme, da den mangler noget for implementation) til at finde maximum flow. Den tager som input et netværk hvor kapacitetsfunktionen c udelukkende bruger heltal, og $s \neq t$. Dens output er et maximum (s,t) flow f i N.



1. $f(u,v) := 0 \quad \forall (u,v) \in E$
2. construct N_f
3. while $\exists (s,t)$-path P in N_f do
4. $\quad \delta(P) := \min \{ c_f(u,v) \mid (u,v) \text{ on } P \}$
5. $\quad f_p \leftarrow$ flow of $\delta(P)$ units along P in N_f
6. $\quad f \leftarrow f \uparrow f_p$
7. $\quad$ construct N_f
8. end
9. output f

Husk at så længe N_f har et augmenting path P, så ved vi at f ikke er maksimum, da $|f \uparrow f_p| = |f| + |\delta(p)| > |f|$. Siden vi udelukkende dealer i integers, gælder det førsagte, da vi altid increaser med mindst én. Derfor findes der også et max-flow med sikkerhed. Dette bliver udvidet i følgende teorem:

Theorem 33 (Max-flow Min-Cut)**.** *Hvis én af disse gælder, gælder alle:*

(1) f er et maximum flow

(2) Der er ingen (s,t)-path i N_f

(3) $|f| = c(S,T)$ for et (s,t)-cut (S,T)

37:15

10 Min-Cut

11 Misc

Nået til 15:15 i video