21
Strenge og Patternmatching.

Leksikografisk ordning af strenge.

Operationer på strenge.

Algebraisk specifikation af strenge.

Forskellige repræsentationer af strenge.

Kompleksitet af operationer på strenge.

Strenge i Eiffel.

Knuth, Morris, Pratt patternmatching.

Boyer Moore patternmatching.

PS1 -- Strenge og patternmatching

Noter

Introduktion til strenge.

"s_1 s_2 s_3 s_4 .... s_n" n >= 0.

En **tekststreng** er en endelig liste af tegn fra et alfabet.

Der findes en **manifest notation** for strenge: "...".

Den **tomme streng** er listen af længde 0.

Et **alfabet** er en mængde af symboler.

ASCII tegnsættet

- "American Standard Code for Information Interchange".
- Et alfabet med 128 forskellige tegn
- Total ordning af tegnene.

Noter

Vi har tidligere i disse noter (under vores diskussion af konstanter) talt om *manifeste udtryk.* Lad os erindre om, at dette betyder en synlig, direkte notation (i dette tilfælde for strenge).

Leksikografisk ordning af strenge.

- Givet en ordning af symbolerne i alfabetet
- Givet to strenge S, T:

S < T hvis og kun hvis et af følgende tilfælde er opfyldt.

- S = *e*
  T = t T'

- S = s S'
  T = t T'

  s < t or
  s = t and S' < T'

PS1 -- Strenge og patternmatching

Noter

Den leksikografiske ordning er den ordning, vi kender mellem ord i f.eks. ordbøger og telefonbøger.

Hvis alfabetet er ASCII tegnsættet, må vi leve med, at talsymbolerne er mindre end store bogstaver, som igen er mindre end de små bogstaver. I visse situationer kan det være hensigtsmæssigt at transformere strengene (f.eks. til strenge, hvori der kun er små bogstaver), inden de sammenlignes.

Den leksikografiske ordning udtrykker, at den tomme streng, som her benævnes *e* , er mindre end en hvilken som helst ikke-tom streng. Endvidere udtrykkes via rekursion hvad det vil sige, at to ikke tomme strenge står i relation til hinanden via <.

Bemærk, at operatoren "<" anvendes på både tegn og strenge på denne slide. Operatoren anvendes dels for den givne ordning på elementerne i alfabetet, dels for den ordning vi er igang med at definere på strenge. Det er vigtigt at forstå, at der er tale om to forskellige operationer (dels på tegn, dels på strenge), som blot har samme navn. Vi siger undertiden, at operatoren er "overloadet".

På denne slide benyttes en rekursiv definition af "<" i en ad hoc notation, som forhåbentlig er klar for læseren. Vi får måske en bedre definition ved at lave en *less-than* operation i en algebraisk specifikation af datatypen string. Dette vender vi tilbage til et par sider længere fremme.

Den abstrakte datatype 'streng'.
Uformel specifikation.

empty	Returner den tomme streng.
char-insert (c, s)	Indsæt tegnet c som det første element i strengen s.
char(i, s)	Returner det i'te tegn af s.
Substring (i, j, s)	Returner delstrengen fra tegn i til og med tegn j af s.
String-insert (i, s, t)	Indsæt strengen s efter tegn nummer i i strengen t.
String-delete(i, j, s)	Slet delstreng fra tegn i til og med tegn j fra s.
Equal (s, t)	Returner om s og t er ens.
Less-than (s, t)	Returner om s er leksikografisk mindre en t.
Length (s)	Returner antallet af tegn i s.
Match (s, t)	Returner om s er en delstreng af t.

Noter

En abstrakt datatype for strenge kan selvsagt defineres på mange måder.

For det første kan der være et forskelligt repertoire af operationer til rådighed. I nogle tilfælde vil der være en lille og måske minimal mængde af operationer til rådighed. I andre vil der være en meget righoldig mængde af operationer defineret. Forskelligheden i repertoiret af operationer på strenge afspejler sig også i valg af terminologi.

For det andet kan strenge repræsenteres på forskellige måder. I første omgang er vi mest interesseret i den eksterne opførsel af strenge, altså operationerne. En uhensigtsmæssig repræsentation kan dog smitte af på de eksterne aspekter af strenge; dette er f.eks. tilfældet, hvis der er en øvre grænse på længden af strenge.

Senere i forelæsningen vil vi se på forskellige mulige repræsentationer for strenge i en implementation af ADT-en String.

Den intuitive og uformelle beskrivelse af operationener på denne slide følges op på næste slide, hvor vi giver grundlaget for en formel, algebraisk specifikation af strenge.

En bemærkning om match: I praktiske sammenhænge er det ikke så meget bevendt, at match returnerer et boolsk resultat, der fortæller, om s er en delstreng af t. I de fleste tilfælde skal vi derefter til at undersøge, hvor i t strengen s matcher. Derfor ønsker vi, at match returnerer noget mere informativt end true eller false, f.eks. indekset hvor matchet starter. I tilfælde af manglende match, kan vi vælge at returnere 0 eller et negativt tal. Dette er udtryk for en generel iagttagelse.

Algebraisk specifikation af strenge.

Type String[Character]

Functions

empty: -> String[Character]
char-insert: Char x String[Character] -> String[Character]
char: Integer x String[Character] -> Character
substring: Integer x Integer x String[Character] -> String[Character]
string-insert: Integer x String[Character] x String[Character] -> String[Character]
string-delete: Integer x Integer x String[Character] -> String[Character]
equal: String[Character] x String[Character] -> Boolean
less-than: String[Character] x String[Character] -> Boolean
length: String[Character] -> Integer
match: String[Character] x String[Character] -> Boolean

Axioms

for all c, d in Char, s, t in String[Character], i, j in Integer
substring(i, j, empty()) = empty()
substring(i, j, char-insert (c, s)) = if i = 1
then string-prefix(j, s)
substring(i - 1, j - 1, s)
less-than(empty(), char-insert (c, s)) = true
less-than(s, empty()) = false
less-than(char-insert (c, s), char-insert (d, t)) = c < d or
c = d and less-than(s, t)

PS1 -- Strenge og patternmatching

Noter

Vi angiver en typeparameter på String, selvom man plejer at sige, at strenge er sekvenser af typen character (typisk tegn fra ASCII tegnsættet).

Der er dog ingen grund til at begrænse os unødigt i denne specifikation. Vi kræver kun nogle få egenskaber ved typen Char: Lighed skal være defineret; og der skal være defineret en total ordning < på det underliggende alfabet.

Der er to konstruktorer (generatorer): empty og char-insert.

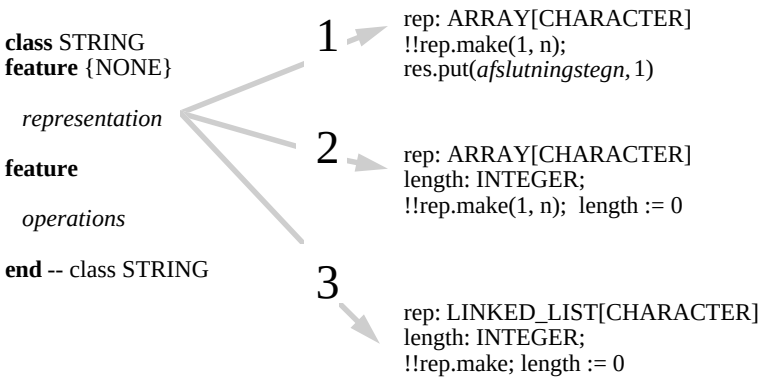
Der er tre transformationer (extensorer): substring, string-insert og string-delete.

De øvrige operationer er accessorer (observatorer).

Der vises kun ligninger for en lille delmængde af operatorerne (substring og less-than); Dette er ikke, fordi jeg er doven, men fordi opskrivning af resten af ligninger er en god øvelse, der giver fin træning i definition af algebraiske specifikationer. Endvidere er der jo ikke mere plads på denne slide...

Opgave: Gør specifikationen af String[Character] færdig (incl. den anvendte, men ikke specificerede hjælpeoperation string-prefix). Vær opmærksom på fejlcheck (betingelser for at operationerne er meningsfulde, gerne i form af prebetingelser på operationerne ala OOSC).

Repræsentation af strenge.



PS1 -- Strenge og patternmatching

Noter

På denne slide er vist tre forskellige, blandt mange mulige repræsentationer af strenge. Vi har også til højre antydet initialiseringen af strengen til at være tom. I alle tilfælde er STRING en klasse ala Eiffel.

1. Denne repræsentation består i et array, hvor sidst benyttede plads er mærket på en speciel måde (med et *afslutningstegn*), typisk med et null tegn. Dette er løsningen, som både Modula-2 og C benytter sig af. (Begge disse sprog har, i modsætning til Eiffel, sproglig understøttelse af strengbegrebet. Se dog senere diskussionen om Eiffel og strenge).

2. Denne repræsentation adskiller sig fra den forrige ved, at vi løbende holder styr på antallet af tegn i array-et. Derved er det ikke nødvendigt at mærke afslutningen på en særlig måde.

3. I modsætning til repræsentationerne 1 og 2, består denne i en dynamisk allokeret liste af kædede tegn. På samme måde som i repræsentation 2, holder vi løbende styr på antallet af tegn i strenge. Bemærk, at string klassen er en klient af klassen linked_list--enkeltkædededede lister---som vi tidligere har diskuteret.

I de tre ovenstående forslag repræsenteres en streng i termer af instanser af eksisterende klasser. Dette giver ikke nødvendigvis den mest kompakte eller effektive datatype. Hvis eksempelvis ordlængden i maskinen er 32 bit vil det være optimalt at repræsentere 4 tegn i hvert ord. Vi har ikke i nogen af ovenstående forslag kontrol over om tegnene pakkes effektivt i maskinord. Derfor er det måske fornuftigt at falde tilbage på en mere maskinnær repræsentation, således at krav til kompakthed kan tilgodeses.

Generelt kan det diskuteres om der skal bruges en konsekutiv repræsentation (med arrays), en kædet repræsentation (med kædede lister), eller en kombination af disse. Fordelen ved den konsekutive repræsentation er kompakthed og effektiv tilgang til delstrenge; ulempen er ineffektivitet ved indsættelse eller sletning af tegn "i midten" af strengen. Fordelen ved den kædede repræsentation er til gengæld fleksibilitet ved indsættelse (og sletning) af tegn inde i strengen; ulempen er at pointerne fylder meget, og at der ikke er direkte tilgang til delstrenge med givne positioner. Derfor kan det være attraktivt at forsøge sig med "et kompromis", i hvilken klumper af konsekutive tegn (af fast eller variabel længde) kædes sammen i lister.

Køretider af operationer på strenge.

n = length(s)
m = length(t)

	1	2	3
empty	O(1)		
char	O(1)		
char-insert (c, s)	O(n)		
Substring (i, j, s)	O(j - i)		
String-insert (i, s, t)	O(n + m - i)		
String-delete(i, j, s)	O(n - j)		
Equal (s, t)	O(min(n, m))		
Less-than (s, t)	O(min(n, m))		
Length (s)	O(n)		
Match (s, t)	O(n * m)		

Noter

Diskussionen på denne slide er relativ til repræsentation 1 fra forrige slide.

less-than: man kan risikere at skulle gennemløbe begge strenge i en længde svarende til den mindste af strengene.

substring: Vi ønsker, at den oprindelige streng og delstrengen er disjunkte. Tegnene i delstrengen skal derfor kopieres over i strengen, som er resultatet af operationen.

string-insert: Først skal man forskyde m - i tegn i t, hvilket koster et tilsvarende antal assignments. Derefter skal man indsætte n nye tegn.

string-delete: Det eneste, der koster ved sletning af en delstreng, er at forskyde "halen" af strengen frem i listen. Længden af denne hale er netop n - j.

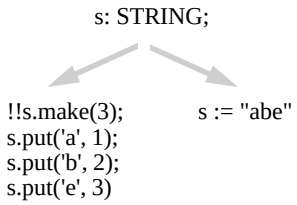
char-insert: Køretidsovervejelserne er tilsvarende som for string-insert.

match: I en naiv implementation vil man for hver position i t forsøge at "matche" s, indtil matchingen lykkes, eller indtil der opstår et mis-match. Dette giver en køretid på O(n * m). Mere om dette lidt senere i dette kapitel, hvor vi studerer patternmatching.

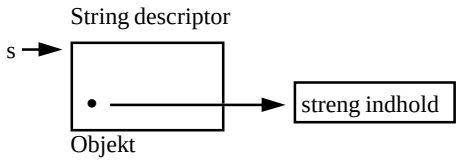
Opgave: Udfyld resten af skemaet på denne slide. Med andre ord: vurder på en kvalificeret måde kompleksiteten af operationerne givet repræsentationerne 2 hhv. 3 fra forrige slide.

Strenge i Eiffel.

Instantiering og initialisering



Repræsentation



PS1 -- Strenge og patternmatching

Noter

En streng i Eiffel er et objekt, beskrevet ved en klasse.

Streng objekter er specielle derved, at *sproget* stiller en manifest notation til rådighed for notering af streng konstanter; den sædvanlige "dette er en streng". På denne slide har vi som venstre alternativ angivet, hvordan det ville være at arbejde med strenge uden denne mulighed. Tegnvis definition af strenge er særdeles uhensigtsmæssig i praksis. Bemærk dog, at dette stort set svarer til vores streng konstruktor char-insert i den algebraiske specifikation af strenge.

Parameteren til make giver ikke en bindende længde på den streng, som objektet indeholder. Dette er en meget behagelig pragmatisk egenskab ved strenge, som vi forøvrigt også kender fra arrays i Eiffel.

Forneden på denne slide er det vist, hvordan en streng repræsenteres i Eiffel. Denne repræsentation giver tre mulige semantikker af assignments af t til s (t := s) hvor s og t er erklærede som strenge:

1. Efter assignmentet deler t og s streng descriptor (reference semantik af assignment).

2. Efter assignmentet har t og s hver sin streng descriptor, men fælles strengindhold.

3. Efter assignmentet er t og s helt separate, med hver sin descriptor og indhold (værdisemantik).

Mulighed 1 gives af effekten ved assignment :=. Mulighed 2, som opnåes via standard clone, er vi sjældent interesseret i. Derfor er clone redefineret i STRING, således at

```
t := clone(s)
```

realiserer mulighed 3.

Lighed af strenge (s = t) giver ganske tilsvarende problemer. t = s har som sædvanlig referencesemantik. Således er “abe” = “abe” falsk, idet hver forekomst af “abe” skaber et nyt objekt, som kun er lig med (... = ...) sig selv. Equal(s,t) er redefineret i STRING til at sammenligne strengindholdet. Equal er altså gjort én tand mere dyb end standard equal.

Læs mere om alt dette i afsnit 28.8 i Eiffel the Language. (Dette afsnit giver kort og kontant besked).

Naiv patternmatching.

```
match(pat, s: Streng): Indeks:
  let n be length(s)
      m be length(pat)
  in  i := 1;
      while i <= n - m + 1
      do  j := 1;  i0 := i;
          while j <= m and char(i, s) = char(j, pat)
          do  i := i + 1;  j := j + 1 end;
          if j = m + 1
          then return i0   -- stop nu: der er fundet et match
          else i := i + 1 end    -- prøv næste position i s
      end;
      return 0  -- der blev ikke fundet et match
  end
```

Et eksempel på worst case data:

s = "aaaaaaaaaaaaaaaaaa ... a"
pat = "aaaa ... b"

Køretiden er O(n · m)

Typisk er n >> m

PS1 -- Strenge og patternmatching

Noter

Forslag til litteratur om patternmatching: Man kan konsultere Horowitz & Sahni's veltjente datastrukturbog *Fundamentals of Data Structures* for en diskussion af naiv patternmatching, og Knuth, Morris og Pratt patternmatching. Hvis man ønsker at se hvordan patternmatching kan programmeres i C++ kan man studere Budd´s bog *Classic Data Structures in C++*. I denne bog er der også en diskussion af Boyer Moore patternmatching.Fremstillingen af patternmatching i disse noter er i et stort omfang baseret på materialet i ovennævnte bøger.

Lidt terminologi:

> Et *pattern* kalder vi på dansk et *mønster*. Mønstret er den tekst, som vi ønsker at søge efter i problemstrengen.
>
> Strengen, hvori vi søger efter forekomster af mønstret, kalder vi *problemstrengen*.
>
> Et *match* betegner en forekomst af mønstret i problemstrengen.
>
> Et *del-match* betegner en forekomst af et prefix af mønstret (en sammenhængende forende af mønstret) i problemstrengen.

Positioner i strenge betegnes som et *indeks*. Føste position har her (og fremover) altid indeks 1.

På denne slide illustreres den "naturlige", men kostbare patternmatching algoritme. Algoritmen returnerer et indeks i problemstrengen s hvorfra mønstret pat matcher. Kort fortalt prøver vi i algoritmen på en systematisk måde at matche alle mulige delstrenge af s op mod pat. Algoritmen løber sålænge der (ud fra en længdemæssig betragtning) er mulighed for at matche pat i s. I det værste tilfælde løber den indre løkke over m-1 elementer uden at findet et match. Den ydre løkke løber fra 1 til n - m + 1. Det giver en køretidsfunktion K på

$$K(n,m) = (m-1)(n-m+1) = O(nm).$$

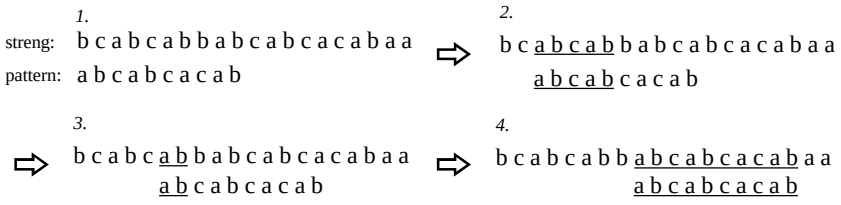
I algoritmen ovenfor antager vi, at **return** terminerer udførelsen af funktionen. Den sidste **return** får effekt, hvis vi ikke på den systematiske facon har fundet et match. Ergo returneres der indeks 0 (med betydning: intet match fundet).

Knuth, Morris & Pratt patternmatching (1).

Streng af længde n.
Pattern af længde m.

- Målet er at finde en patternmatching algoritme, som er lineær i n og m.
- I en sådan algoritme skal man undgå at "gå tilbage i strengen s".
- Den matchede del af pattern indeholder information om, hvor det næste mulige match kan forekomme i strengen.

Motiverende eksempel:



PS1 -- Strenge og patternmatching

Noter

Situation 1: Vi forsøger at matche mønstret pat op mod strengen, ovenfor. Dette giver naturligvis ikke bonus.

Situation 2: Først når vi kommer til denne situation (efter to trin) bliver det interessant. Vi ser at fem tegn i mønstret matcher strengen, men vi opnår ikke et totalt match. Her og i det følgende vil understregede dele i strengen og i pattern vise et match. I den naive algoritme ville vi nu gå tilbage i problemstrengen, dvs. vi ville prøve med flg.:

b c a b c a b b a ...

a b c a b c ...

Dette er dog ikke nødvendigt! Den centrale observation er, at vi kender så meget til både problemstreng og mønster, at vi kan springe en del trin over i forhold til den naive algoritme. Vores kendskab til problemstrengen stammer fra det faktum, at vi lige har fundet overensstemmelse mellem fem tegn i problemstrengen og pat. Ved blot at analysere mønstret pat kan vi se, at vi kan glide pat helt hen til situationen i 3 (ovenfor), og genstarte patternmatchingen fra denne situation. Bemærk, at der ikke er nogle mellemtrin mellem situation 2 og 3.

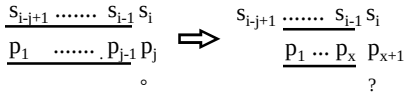
Situation 3: Vi ser, at der kun er overensstemmelse mellem to tegn i problemstrengen og mønstret. Vi springer nu et par trin over, indtil vi kommer til situation 4.

Situation 4: Vi ser, at vi i denne situation opnår et match mellem pat og strengen; Dermed er vi færdige.

Som det fremgår af eksemplet forløber patternmatchingen ved en "jævnt fremadskridende process" i strengen. Der er derfor håb om, at vi har fundet en teknik, der kan give os en pattermatching algoritme med lineær køretid i forhold til længden af problemstrengen (og mønstret).

Knuth, Morris & Pratt patternmatching (2).

Generel situation. Vi søger et x (så stort som muligt) som en funktion af j-1 så:



x vælges om muligt sådan, at et pattern prefix (så langt som muligt) bliver lig med et pattern suffix:

$$p_1 \cdots p_x = p_{j-x} \cdots p_{j-1}$$

Definition:

Givet *pat* $= p_1 \ldots p_m$

failure(*pat* , j) $=$ Det største i, i < j, så $p_1 \ldots p_i = p_{j-i+1} \ldots p_j$, hvis et sådant i >= 1 findes.

0 ellers

PS1 -- Strenge og patternmatching

Noter

På denne slide viser vi i den generelle situation hvad vi indså i termer af eksemplet på forrige slide.

Øverst til venstret har vi en situation hvor vi har et mismatch på j-te tegn i mønstret $p_1 \ldots p_j \ldots p_m$. Vi ønsker nu at finde det størst mulige suffix (endestykke) af vores pattern, som er lig med et prefix (begyndelsesstykke) af pattern. Vi lader $p_1 \ldots p_x$ betegne dette stykke. Øverst til højre (efter pilen) ser vi hvordan vi har udnyttet dette til at forskyde vores pattern hen over strengen. (I tilfælde af, at en sådan overensstemmelse mellem et pattern prefix og pattern suffix ikke findes, lader vi x være 0. Det vil komme til at betyde, at p_1 forskydes helt hen under s_i).

Nederst definerer vi funktion failure(j). Værdien af failure(j) fortæller os hvor stor en del af mønster prefixet der matcher frem til tegnet s_{i-1}. Bemærk at i situationen øverst på denne slide har vi behov for værdien failure(j-1). Værdien af x skal altså være

failure(j-1).

Det er værd at notere sig, at failure funktionen kan beregnes én gang for alle for et givet pattern, inden vi starter på selve patternmatchingen. Vi er selvfølgelig interesseret i at kunne beregne funktionen så effektivt som muligt. Det er dog behageligt at konstatere, at mønstret typisk er meget kortere end den streng, vi "matcher op imod". Vi vender tilbage til hvordan failure funktionen kan beregnes effektivt (et par sider længere fremme).

Eksempel 1 på tabellægning af failure-funktionen for det mønster, vi studerede på forrige slide:

position: 1 2 3 4 5 6 7 8 9 10

pattern: a b c a b c a c a b

failure fn: 0 0 0 1 2 3 4 0 1 2

Eksempel 2:

position: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

pattern: a g c t a g c a g c t a g c t g

failure fn: 0 0 0 0 1 2 3 1 2 3 4 5 6 7 4 5

Knuth, Morris & Pratt patternmatching (3)

Algoritmisk idé:

Givet en problemstreng s og et mønster p.

Metode.

1. Vi matcher p mod s fra venstre mod højre.
2. I tilfælde af et mis-match i position y i strengen s og i position x i p forskydes p således at $p_{failure(p, x-1)}$ står ud for s_{y-1}.
3. Matchingen fortsætter nu som i punkt 1, fra position y i problemstrengen s, og fra position x i mønstret p.

Noter

Knuth, Morris & Pratt patternmatching (4).

```
KMP-match(pat, s: Streng): Indeks:
  let n be length(s)
      m be length(pat)
  in  i := 1; -- i løber gennem s
      j := 1; -- j løber gennem pat
      match := false;
      while i <= n and not match
      invariant j-1 tegn i pat matcher op til tegn nummer i-1 i strengen s.
      do  if char(j, pat) = char(i, s)
          then -- tegn matcher
                i := i + 1; j := j + 1
          else if failure(pat, j-1) > 0
               then j := failure(pat, j-1) + 1        (*)
               else j := 1; i := i + 1 end
          end;
          match :=  (j = m +1);
      end;
      if match then return i-j+1 else return 0 end
  end
```

Køretid = O(n)
forudsat at failure funktionen allerede er tabellagt.

PS1 -- Strenge og patternmatching

Noter

Lige som i den naive patternmatcing algoritme, som vi tidligere har studeret, returnerer denne algoritme et indeks, hvorfra vores mønster pat matcher. Hvis ikke et sådant indeks (>= 1) findes, returneres 0, som et signal om, at mønstret pat ikke matcher nogetsteds i strengen s.

At failure funktionen er tabellagt betyder, at failure(pat,j) kan beregnes i konstant tid.

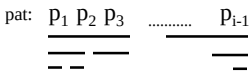
Hvad angår køretiden af KMP-match kan vi observere, at matchingen forløber i én while-løkke, og at i bliver én større i alle forgreninger af de indre if-then-else sætninger, pånær i then-delen af den yderste else-del (som vi har mærket (*)). For at have styr på køretiden af KMP-match skal vi løst sagt kunne argumentere for, at grenen (*) ikke dominerer køretiden i algoritmen.

Hvis vi antager at (*) skulle blive et problem for den lineære køretid af hele algoritmen, skal (*) udføres et ikke-konstant antal gange pr. iteration. Alt i alt skal (*) i hele while-løkkens køretid udføres m*g(m) gange, hvor g er en funktion der har større vækst-rate en den konstante funktion. Det kan imidlertid ikke lade sig gøre, da (*) tæller j ned. (j tælles op ialt m gange, og kan således også højst tælles ned m gange, alt i alt). I gennemsnit gælder altså at (*) har en køretid der er konstant, og derfor ødelægger (*) ikke den lineære køretid af hele algoritmen.

Bemærk at denne analyse, og det ræsonnement, der er nødvendigt, er anderledes og mere kompliceret end “sædvanlig algoritmeanalyse”.

Knuth, Morris & Pratt patternmatching (5).

Beregning af failure(pat, j):



failure(pat,1) = 0.

Antag at failure(pat, i-1) = x
Er $p_{x+1} = p_i$: failure(pat, i) = x + 1.
Ellers antag at failure(pat, x) = y
Er $p_{y+1} = p_i$: failure(pat, i) = y + 1
Ellers ...

Givet et pattern af længde m.

Det kan vises at køretiden for tabellægning af failure er O(m).

Givet en streng af længde n.

Køretiden af KMP-match bliver O(n + m).

KMP-match på "worst case data":

s = "aaaaaaaaaaaaaaaaaa ... a"
pat = "aaaa ... b"

Tabel for failure funktion på pat:

j	1	2	3	4	...	n-1	n
failure(j)	0	1	2	3		n-2	0

PS1 -- Strenge og patternmatching

Noter

Følgende eksempel, som vi studerede tidligere, er god for at indse hvordan failure-funktionen kan tabellægges:

Eksempel 2:

position: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

pattern: a g c t a g c a g c t a g c t g

failure fn: 0 0 0 0 1 2 3 1 2 3 4 5 6 7 ? ?

Interesserede læsere, der ønsker en algoritme for tabbellægning af failure-funktionen kan konsultere afsnit 4.11.2 i Horowitz & Sahni's bog *Fundamentals of Data Structures.*

Nederst på denne slide vender vi tilbage til det eksempel, som var meget dyr for den naive patternmatcher. Hvordan forløber KMP-patternmatcing af "aaaa....b" på "aaaaaaaaa....a"?

Vi starter med at matche vores mønster, og vi får et mismatch pga. tegnet 'b':

aaaaaaaaaaaaaaaaaaaa

aaaab

I stedet for at matche en hel række a-er igen, rykker vi nu mønstret en tand til højre:

aaaaaaaaaaaaaaaaaaaa

aaaab

Vi ved, at de første tre a-er matcher, og vi finder ud af, at det fjerde a i mønstret også matcher. b giver dog igen et mismatch. Således fortsættes. Lad os tage næste trin med:

aaaaaaaaaaaaaaaaaaaa

aaaab

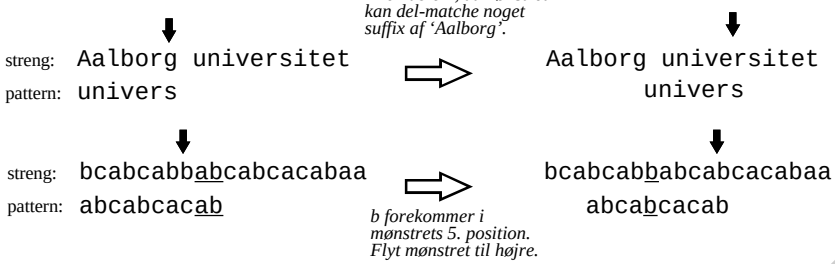
Patternmatchingen skrider lineært hen over strengen, uden antydning af nogen $\Theta(n \cdot m)$ opførsel i køretid.

Boyer Moore patternmatching (1).

Streng af længde n.
Pattern af længde m.

- Umiddelbart skulle man tro af Knuth Morris Pratt patternmatching algoritmen er optimal.
- Ved at udføre patternmatchingen fra bagenden af mønstret kan man imidlertid helt undgå at matche dele af strengen.

Motiverende eksempler:



PS1 -- Strenge og patternmatching

Noter

Vi skifter nu til en ny patternmatching strategi.

Ovenfor angiver pilen det nuværende undersøgelsessted i problemstrengen. Den del af problemstrengen, der er til venstre for pilen er allerede undersøgt for matches.

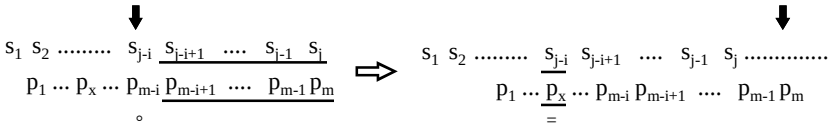
I det motiverende eksempel, hvor mønstret "univers" matches op imod "Aalborg universitet" ser vi, at der er et mismatch på den sidste position. Vi skal nu flytter mønstret til højre. Men hvor langt til højre? Den vigtige observation er, at vi skal så langt til højre at den højre-meste forekomst af tegnet 'g' i univers placeres under 'g' i "Aalborg". Men univers indeholder jo ikke et 'g'. Derfor kan vi med ét slag forskyde mønstret en hel længde, uden overhovedet at have sammenlignet mere end et bogstav i problemstrengen og mønstret (nemlig det sidste). Dette må siges at være en brandgod idé.

Terminologi: Den højre-meste forekomst af et tegn i en streng er den sidste (mest til højre) forekomst at tegnet i strengen.

Det nederste eksempel er det samme, som vi startede med at studere for Knuth, Morris og Pratt patternmatching. Vi ser, at b ikke matcher c, hvorfor vi forskyder mønstret til højre, således at den højre-meste forekomst af b (før den nuværende postion, hvilket reelt er en tilsnigelse) kommer på linie med problemstrengens tegn b.

Boyer Moore patternmatching (2).

Generel situation. Vi søger et x (så stort som muligt) så p_x matcher s_{j-i}



Definition:

Givet et mønster *pat* = $p_1\ p_2\ \ldots\ p_m$

right: Alfabet -> Integer

right(*pat*, x) = hvis x forekommer i *pat*: det højeste indeks af x i *pat*.
0 ellers.

Eksempel: *pat* = "regninger".

x: `abcdefghijklmnopqrstuvwxyzæøå`
right(*pat*, x): `00008070500006000900000000000`

PS1 -- Strenge og patternmatching

Noter

Ligesom ved failure funktionen i Knuth, Morris og Pratt patternmatching indfører vi her en funktion, som vi kalder right. Right anvendes til at besluttet, hvor meget mønstret skal skiftes til højre efter et mis-match.

Vi skal tabellægge funktionen right inden vi starter en Boyer Moore patternmatching. Dette er en triviel opgave i forhold til tabellægningen af failure funktionen fra tidligere. Der er følgende skridt i tabllægningen af right:

1. Alloker en tabel med plads til hvert tegn i alfabetet.
2. Nulstil hele tabellen.
3. Gennemløb nu mønstret fra venstre mod højre, og put information i tabellen om positionen af hvert enkelt tegn.

Venstre til højre gennemløbet af mønstret ender med at have den effekt, at det er positionen af den højre-meste forekomst af et tegn der får afsmitningen i tabellen. Køretiden for dette er klart O(m), hvor m er længden af mønstret.

Boyer Moore patternmatching (3).

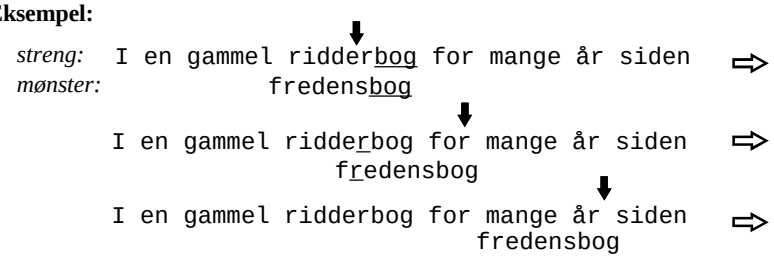
Algoritmisk idé:

Givet en problemstreng s og et pattern p.

Metode.

1. Vi matcher p mod s fra højre mod venstre.
2. I tilfælde af et mis-match i position y i strengen s forskydes p_1 i mønstret til position y - right(s_y) + 1 i forhold til strengen s.
3. Matchingen fortsætter nu som i punkt 1.

Eksempel:



PS1 -- Strenge og patternmatching

Noter

Det skal bemærkes, at hvis vi flytter mønstret, som angivet i den algoritmiske idé, kan man komme til at flytte mønstret til venstre. Dette sker hvis tegnet s_y forekommer i den allerede matchede del i vores mønster. Vi er naturligvis ikke interesserede i at flytte mønstret til venstre. Derfor skal vi i dette tilfælde blot flytte mønstret en tand til højre (ganske som i naiv patternmatching).

Nederst viser vi tre trin af Boyer Moore pattern matching. Bemærk, at der ikke forekommer skjulte mellemresultater mellem de viste tre trin.

Boyer Moore patternmatching (4).

```
BM-match(pat, s: Streng): Indeks:
  let n be length(s)
      m be length(pat)
  in   i := length(pat);  i-last := i;    --i løber gennem s
       while i <= n
       do j := length(pat);                -- j løber gennem pat
          while j >= 1 and char(s, i) = char(p,j)
          do i := i - 1;  j := j - 1 end;
          if j = 0
          then return i+1
          else -- i hopper frem i strengen s:
                delta := Max(1, j - right(pat, char(s,i)) ); --beregner forskydning
                i-last := i-last + delta
                i := i-last  end
       end;
       return 0
  end
```


Noter

Øjebliksbillede i ovenstående algoritme: