INF2220 - Algoritmer og datastrukturer

HØSTEN 2015

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 10:

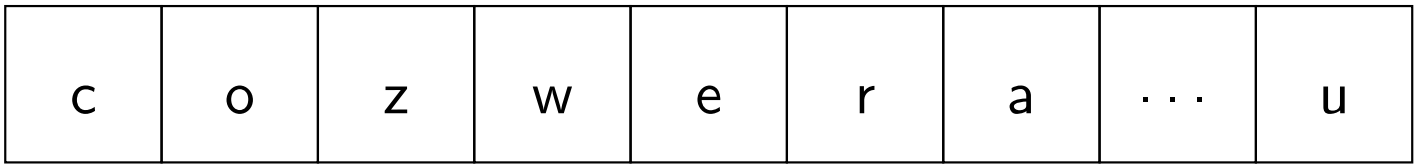
Tekstalgoritmer 1

Pattern matching algorithms

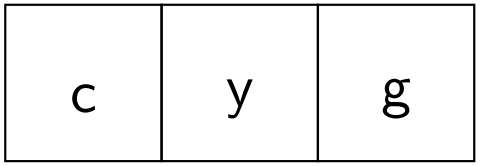
Algoritmer for lokalisering av substrenger

- Brute force
  - Enkleste tenkelige algoritme for å løse problemet
- java.lang.String `indexOf`
  - Vi ser på hvordan det gjøres i standard biblioteket
- Boyer Moore (Horspool)
  - Relativt komplisert algoritme, med rask **worst case**

Lokalisering av Substrenger



Høystakk



Nål

- Fins nåla i høystakken?
- Hvis JA: hvor?

Brute force

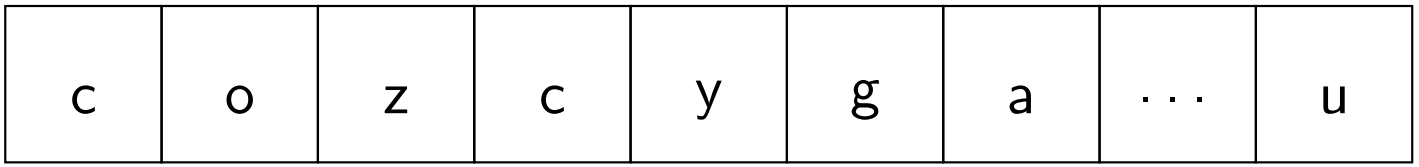
Brute force (rå kraft) brukes ofte synonymt med

- unødvendig tung
- dårlig
- treg
- lite gjennomtenkt
- nødløsning

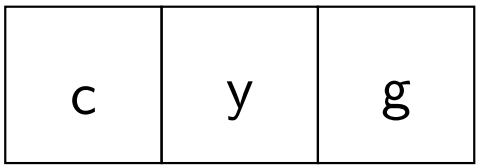
men er noen ganger nødvendig

- Brute force løsninger er typisk den første ideen vi får
- Stort sett hele dette kurset går ut på å unngå de

Brute force



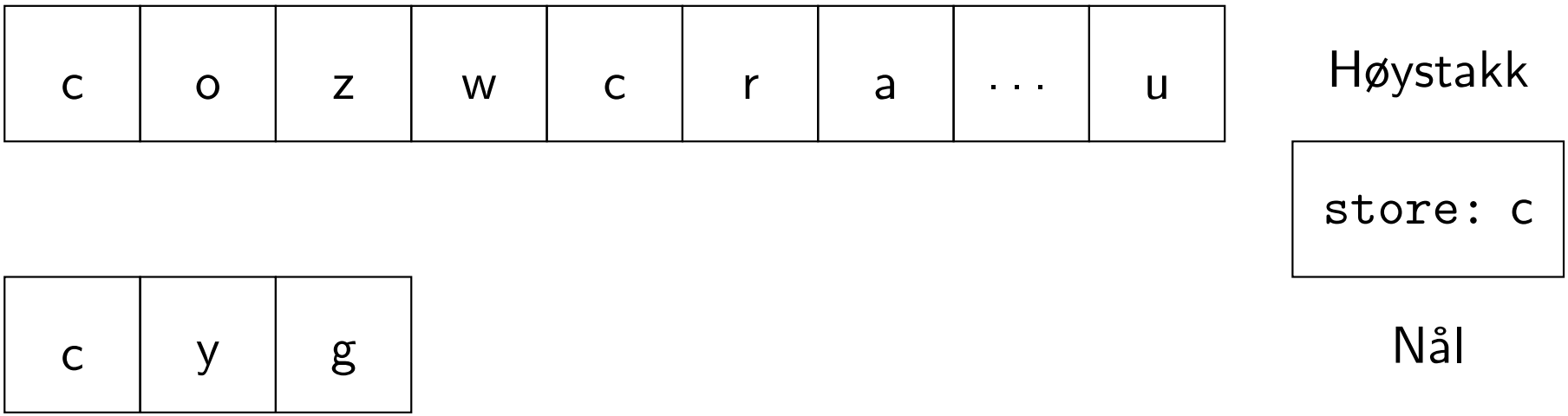
Høystakk



Nål

- Var ikke det den første algoritmen du tenkte på?
- Hva blir kompleksiteten av et sånt søk?

java.lang.String `indexOf`

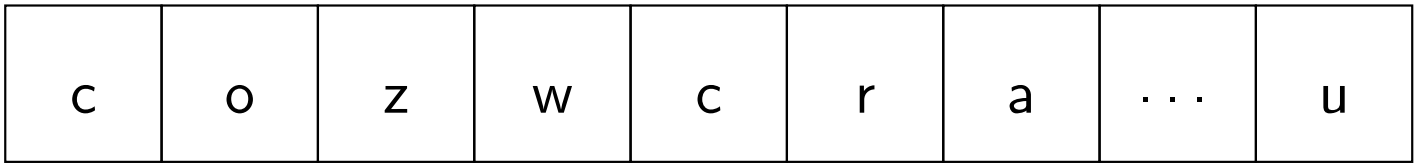


- Vi lagrer første element i **nål**
- Så leiter vi etter match på første element før vi flytter **nål**

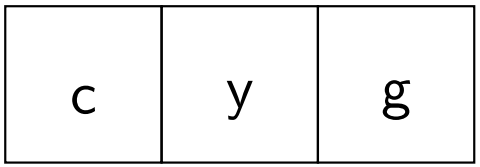
Boyer Moore

- Skal vi øke hastigheten må vi minske antall sammenligninger
  - Vi kan preprosessere informasjonen i **nål** og **høystakk**
  - Vi kan gjøre rimelige antagelser om input
- Boyer Moore antar at vi bare har 1-byte characters
- 1-byte characters gir oss 256 muligheter ($2^8 = 256$)
- Vi kan bruke den informasjonen til å preprosessere **nålen**

Boyer Moore



Høystakk



Nål

- Vi matcher baklengs med Boyer Moore
- Merk at elementet z **ikke** finnes i nålen
- Dvs. etter første match kan nålen flyttes **3** hakk frem

—	—	z	—	—	—	—	—	—
c	y	g	—	—	—	—	—	—
—	c	y	g	—	—	—	—	—
—	—	c	y	g	—	—	—	—

- Ingen match for søkestrengen helt i starten av teksten eller de neste to posisjonene etter det
- Vi kan hoppe fremover og begynne å lete etter en match på den sjette plasseringen av teksten
- Informasjonen vi trenger for å beregne dette ligger i arrayen vi kaller `bad character shift`

Bad Character Shift

- Hvordan beregne bad character shift?
- Vi må raskt kunne svare på om en bokstav er med i **nålen**
- Bokstaver er 1-byte lange dvs. (int) bokstav $\in [0, 255]$
- `badCharShift` er en array `int[256]`
- Vi fyller denne med **shift** verdier ut i fra hva som er i **nålen**

bad character shift (forenklet)

```
int[] badCharShift = new int[256]; // assume 1-byte characters

for(int i = 0; i < badCharShift.length; i++){
    badCharShift[i] = needle.length;
}

/*  shift size = 1 for characters inside needle  */
for(int i = 0; i < needle.length; i++){
    badCharShift[ (int) needle[i] ] = 1;
}
```

bad character shift (forenklet)

Hvordan ser preprosesseringen ut hvis vi bruker **nålen** fra i sta?

c	y	g

Nål

- `badCharShift[ 99 ] == 1` `99 == (int) 'c'`
- `badCharShift[ 121 ] == 1` `121 == (int) 'y'`
- `badCharShift[ 103 ] == 1` `103 == (int) 'g'`
- **For alle andre verdier** $x \in [0, 255]$
- `badCharShift[ x ] == 3` `(needle.length == 3)`

bad character shift (forenklet)

c	o	z	w	r	c	a	v	u

Høystakk

c	y	g

Nål

- `badCharShift[ 122 ] == 3` `122 == (int) 'z'`
- `badCharShift[ 99 ] == 1` `99 == (int) 'c'`
- `badCharShift[ 97 ] == 3` `97 == (int) 'a'`
- Vi kan gjøre det bedre for bokstavene som er i nålen

bad character shift

```
int[] badCharShift = new int[256]; // assume 1-byte characters

for(int i = 0; i < badCharShift.length; i++){
    badCharShift[i] = needle.length;
}

/* calculate bad shift up to needle.length - 1 */
int last = needle.length - 1;

for(int i = 0; i < last; i++){
    badCharShift[ (int) needle[i] ] = last - i;
}
```

bad character shift

Hvordan ser preprosesseringen ut hvis vi bruker **nålen** fra i sta?

c	y	g

Nål

- `badCharShift[ 99 ] == 2` `99 == (int) 'c'`
- `badCharShift[ 121 ] == 1` `121 == (int) 'y'`
- **For alle andre verdier** $x \in [0, 255]$
- `badCharShift[ x ] == 3` `(needle.length == 3)`

bad character shift

c	o	z	w	r	c	a	y	u

Høystakk

c	y	g

Nål

- `badCharShift[ 122 ] == 3` `122 == (int) 'z'`
- `badCharShift[ 99 ] == 2` `99 == (int) 'c'`
- `badCharShift[ 121 ] == 1` `121 == (int) 'y'`

Boyer Moore Horspool

```
public int boyer_moore_horspool(char[] needle, char[] haystack){

    if ( needle.length > haystack.length ){ return -1; }

    int[] bad_shift = new int[CHAR_MAX]; // 256

    for(int i = 0; i < CHAR_MAX; i++){
        bad_shift[i] = needle.length;
    }

    int offset = 0, scan = 0;
    int last = needle.length - 1;
    int maxoffset = haystack.length - needle.length;

    for(int i = 0; i < last; i++){
        bad_shift[needle[i]] = last - i;
    }

    while(offset <= maxoffset){

        for(scan = last; needle[scan] == haystack[scan+offset]; scan--){

            if(scan == 0){ // match found!
                return offset;
            }
        }
        offset += bad_shift[haystack[offset + last]];
    }
    return -1;
}
```

Shift-verdien er basert på den siste byten av nålen, uansett hvor vi ikke fikk match.

shift basert på **d**, shift 4

a	a	c	d	—	—	—	—	—
a	b	c	d					
				a	b	c	d	

shift basert på **d**, shift 1

—	a	a	c	d	—	—	—	—
a	c	b	d	d				
	a	c	b	d	d			

- Bad suffix shift er effektiv til for eksempel å søke i tekst i naturlige språk fordi mismatches er sannsynlige
- Med få alfabeter er det sannsynlig å få matching nær slutten av nålen. I dette tilfellet kan vi ha nytte av å vurdere vellykket match-suffikser av nålen.

Boyer Moore Horspool

- Horspool er en forenkling av Boyer-Moore-streng-søkealgoritmen

Boyer-Moore-algoritmen er basert på:

1. å analysere nålen baklengs
2. bad suffix shift
3. good suffix shift

- bad suffix shift unngår å gjenta mislykkede sammenligninger mot et tegn i høystakken
- good suffix shift beregner hvor langt vi kan flytte nålen, basert på antall matchende bokstaver før mismatch
- justerer bare matchende nål-tegn mot høystakk-tegn som allerede har fått match

- good suffix shift er en array som er like lang som nålen

Good suffix shift (Case 1)

Antar at vi har fått match for **nål**[$i \ldots n$]

Hvis **nål**[$i-1$] er en mismatch og **nål** inneholder en annen kopi av **nål**[$i \ldots n$] som ikke har tegnet **nål**[$i-1$] som prefiks, flytt **nål** slik at kopien matcher substrengen som allerede var matchet av **nål**[$i \ldots n$]

```
høystakk: prstabstubabvqxrst
nål:        qcabdabdab
```

```
høystakk: prstabstubabvqxrst
nål:              qcabdabdab
```

Good Suffix Shift

Anta nålen vi leter etter er: **fiskekake**

index	Mismatch	Shift	goodCharShift
0	e	1	`goodCharShift[0] == 1`
1	ke	9	`goodCharShift[1] == 9`
2	ake	4	`goodCharShift[2] == 4`
3	kake	9	`goodCharShift[3] == 9`
4	ekake	9	`goodCharShift[4] == 9`
5	kekake	9	`goodCharShift[5] == 9`
6	skekake	9	`goodCharShift[6] == 9`
7	iskekake	9	`goodCharShift[7] == 9`
8	fiskekake	9	`goodCharShift[8] == 9`

ke representerer en substreng i **fiskekake** som består av et tegn som *ikke* er en 'k' pluss et tegn 'e'

Good suffix shift (case 2)

Antar at **nål**$[i \ldots n]$ og substreng **t** av høystakken matcher, men vi har en mismatch av **nål**$[i-1]$

Hva hvis **nål**$[i \ldots n]$ ikke er en substreng i **nål**$[0 \ldots i-1]$?

```
- - - - X M A N P A N M A N - - - - - -
A N P A N M A N
```

Case 2: flytt **nål** minst mulig slik at et *suffiks* av **t** matcher et *prefiks* av **nål**

Hvis ingen slik match, flytt **nål** med |**nål**| posisjoner.

Good Suffix Shift

Anta nålen vi leter etter er: **ANPANMAN**

index	Mismatch	Shift	goodCharShift
0	N	1	`goodCharShift[0] == 1`
1	AN	8	`goodCharShift[1] == 8`
2	MAN	3	`goodCharShift[2] == 3`
3	NMAN	6	`goodCharShift[3] == 6`
4	ANMAN	6	`goodCharShift[4] == 6`
5	PANMAN	6	`goodCharShift[5] == 6`
6	NPANMAN	6	`goodCharShift[6] == 6`
7	ANPANMAN	6	`goodCharShift[7] == 6`

source: wikipedia.org

Neste forelesning: 29. oktober

Huffman-koding, Dynamisk programmering, Floyds algoritme og paradigmer for algoritmedesign