Notater: INF2220

Veronika Heimsbakk
veronahe@ifi.uio.no

14. november 2014

Innhold

Introduksjon

Dette er notater til kurset INF2220–Algoritmer og datastrukturer[4] ved Universitetet i Oslo. Disse notatene er i all hovedsak basert på forelesningsfoiler, egne notater og læreboka. Brukes til repetisjon før eksamen og som notater under eksamen. Merk at notatene helt sikkert inneholder feil og mangler.

1 Kjøretid

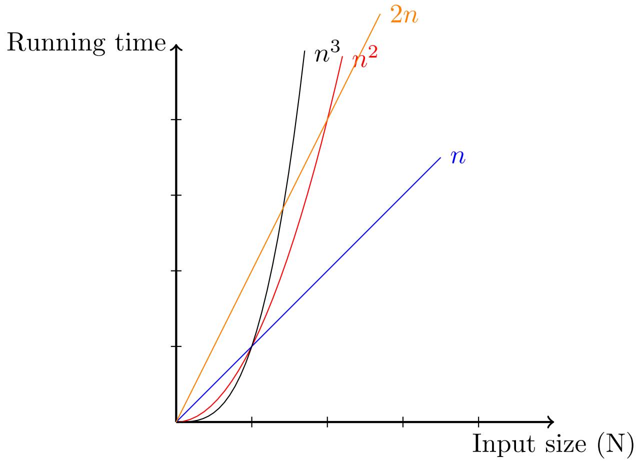
Definisjon 1.1: O-notasjon *La f og g være funksjoner $f, g : \mathcal{N} \to \mathcal{R}^+$. Sier at* ***f(n) = O(g(n))*** *hvis det eksisterer positive heltall c og n_0 slik at for hvert heltall $n \geq n_0$,*

$$f(n) \leq cg(n).$$

Når $f(n) = O(g(n))$, sier vi at $g(n)$ er ***upper bound*** *for $f(n)$, eller mer presist at $g(n)$ er* ***asymptotic upper bound*** *for $f(n)$.*

Funksjon	**Navn**
1	Konstant
$\log n$	Logaritmisk
n	Lineær
$n \log n$	
n^2	Kvadratisk
n^3	Kubisk
2^n	Eksponensiell
$n!$	

Tabell 1: Vanlige funksjoner for O(n)



Figur 1: Noen funksjoner for O(n).

1.2 Analyse av kjøretid

Når man analyserer en algoritme, så må man se på *stegene* den bruker og analysere de hver for seg. Ta for eksempel denne kodesnutten:

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
```

```
    array[i][j] = 0;
```

Her har vi to steg: løkken som går n steg, og løkken som går m steg. Den første løkken går i O(N), mens den andre løkken går i O(M). Siden den innerte løkken går M ganger N ganger (siden den er nestet i den ytterste løkken). Så vil den totale tiden være O(M) $\times$ O(N), dermed O(n^2).

1.3 Logaritmisk tid

Algoritmer som tar logaritmisk tid, O($\log n$), finner man i operasjoner på binære trær, eller når man bruker binære søk.

2 Trær

Definisjon 2.1: Tre *Et* ***tre*** *er en samling* ***noder****. Et ikke-tomt tre består av en* ***rot****-node, og null eller flere ikke-tomme* ***subtrær****. Fra roten går en* ***rettet kant*** *til roten i hvert subtre.*

2.2 Traversering

Pre-order rot, venstre barn, høyre barn

In-order venstre barn, rot, høyre barn

Post-order venstre barn, høyre barn, rot

```
/* IN-ORDER TRAVERSAL */

public void inOrder (Node node) {
  if (node != null) {
    inOrder(node.left);
    // do something with the node
    inOrder(node.right);
  }
}

/* PRE-ORDER TRAVERSAL */

public void preOrder (Node node) {
  if (node != null) {
    // do something with the node
    preOrder(node.left);
    preOrder(node.right);
  }
}

/* POST-ORDER TRAVERSAL */

public void postOrder (Node node) {
  if (node != null) {
    postOrder(node.left);
    postOrder(node.right);
    // do something with the node
  }
}
```

2.3 Terminologi

Definisjon 2.4: Sti *En sti fra node* n_1 *til* n_k *er definert som en sekvens* $n_1, n_2, \ldots, n_k$*, slik at* n_i *er forelder til* n_{i+1} *for* $1 \leq i \leq k$.

Definisjon 2.5: Lengde *Antall kanter på veien;* $k - 1$.

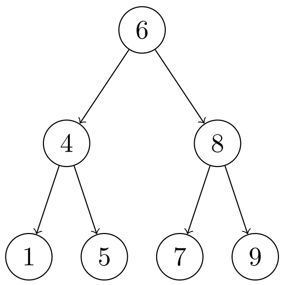
Definisjon 2.6: Dybde *Definert av den unike veien fra roten til noden. Rotens dybde er 0.*

Definisjon 2.7: Høyde *Definert som lengden av den lengste veien fra noden til en løvnode. Alle løvnoder har høyde 0. Høyden til et tre er lik høyden til roten.*

2.8 Binært søketre

Definisjon 2.9: Binært søketre *Et binært søketre (BST) er et* ***ordnet*** *binært tre. Følgende kriterier må være oppfylt for at treet skal være et BST:*

- *Verdiene i* ***venstre*** *subtre er* ***mindre*** *enn noden i seg selv.*
- *Verdiene i* ***høyre*** *subtre er* ***større*** *enn noden i seg selv.*
- *Venstre og høyre subtre må også være binære søketrær.*
- *Hver node kan ha maksimum to barn.*
- *Det eksisterer en* ***unik*** *sti fra roten til enhver node i treet.*



Figur 2: Eksempel på et binært søketre.

2.9.1 Tidkompleksitet

	Average	Worst case
Plass	O(n)	O(n)
Søk	O($\log n$)	O(n)
Innsetting	O($\log n$)	O(n)
Sletting	O($\log n$)	O(n)

Tabell 2: Tidkompleksitet for BST.

2.9.2 Søking

```
function find(key, node):
  if node = Null or node.key = key then
    return node
  else if key < node.key then
    return find(key, node.left)
  else
    return find(key, node.right)
```

2.9.3 Sette inn

```
function insert(root, data):
  if (!root)
    root = new Node(data)
  else if (data < root.data)
    insert(root.left, data)
  else if (data > root.data)
    insert(root.right, data)
```

2.10 Rød-svarte trær

Definisjon 2.11: Rød-svart tre *En datastruktur som er et* ***selv-balanserende*** *binært søketre. Følgende skal gjelde for et rød-svart tre:*

1. *En node er enten rød eller sort.*
2. *Roten er sort.*
3. *Alle løvnoder (NIL) er sorte—alle løvnoder har samme farge som roten.*
4. *Enhver røde node må ha to sorte barn.*
5. *Enhver sti fra roten til en løvnode skal inneholde samme antall sorte noder.*

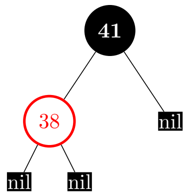
Dette sikrer at høyden på et slikt tre er maks $2 \times \log_2(N+1)$.

2.11.1 Eksempel på innsetting

	Average	Worst case
Plass	$O(n)$	$O(n)$
Søk	$O(\log n)$	$O(\log n)$
Innsetting	$O(\log n)$	$O(\log n)$
Sletting	$O(\log n)$	$O(\log n)$

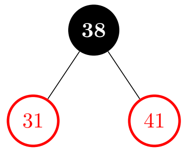
Tabell 3: Tidkompleksitet for rød-svarte trær.

1



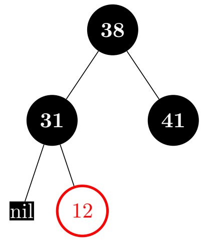
Setter inn 41 og 38 etter reglene.

2



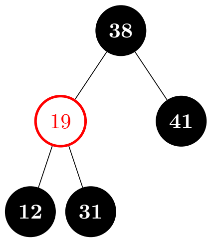
Når 31 nå kommer inn, så må vi snu på treet og farge om 38 og 41.

3



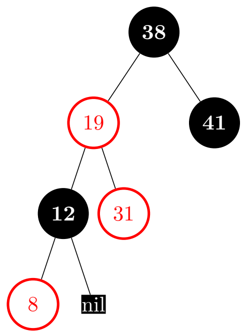
Siden løvnodene var røde når 12 skal inn, så må 31 og 41 farges om. 41 skal farges iht. regel nr. 3.

4.



Når 19 nå kommer inn, så må vi igjen snu på treet for å opprettholde reglene til et BST. Farger nodene på nytt iht. reglene for rød-svarte trær.

5.



Til slutt setter vi inn 8 som løvnode til 12. Her trenger vi ikke å snu eller fargelegge.

Figur 3: Eksempel på innsetting av tallene 41, 38, 31, 12, 19 og 8 i et rød-svart tre.

2.12 B-trær

3 Maps

4 Hashing

Idéen i hashing er å lagre alle elementene i en array (hashtabell) og la verdien i elementet x bestemme indeksen til x i hashtabellen. Egenskaper til en god hash-funksjon er at den er rask å beregne, kan gi alle mulige verdier fra 0 til tabellens størrelse -1. Og den git en god fordeling utover tabellindeksene. En hashtabell tilbyr innsetting, sletting og søking med konstant tid.

4.1 Hash-funksjoner

Eksempel med heltall som nøkkel, begrenset antall tabellindekser. La hashfunksjonen være `hash(x, taleSize) = x mod tableSize`. Husk at hvis `tableSize` er 10 og alle nøklene slutter på 0, vil alle elementene havne på samme indeks. Huskeregel for dette: la alltid tabellstørrelsen være et primtall. Funksjonen under summerer verdiene til hver bokstav. Dette er en dårlig fordeling dersom tabellstørrelsen er stor.

```
/* Red-black insertion */

public void put (Key key, Value val) {
  root = put(root, key, val);
  root.color = BLACK;
}

// insert key-value pair in subtree with root node
private Node put (Node node, Key key, Value val) {
  if (node == null) return new Node(key, val, RED, 1);

  int cmp = key.compareTo(node.key);
  if (cmp < 0)
    node.left = put(node.left , key, val);
  if else (cmp > 0)
    node.right = put (node.right), key, val);
  else
    node.val = val;

  if ( isRed(node.right) && !isRed(node.left) )
    node = rotateLeft(node);
  if ( isRed(node.left) && isRed(node.left.left) )
    node = rotateRight(node);
  if ( isRed(node.left) && isRed(node.right) )
    flipColors(node);
  node.N = size(node.left) + size(node.right) + 1;

  return node;
}
```

Figur 4: Eksempel på innsetting i rødsvart tre [1].

```
int hash (String key, int tableSize) {
  int hashValue = 0;
  for (i = 0; i < key.length(); i++)
    hashValue+= key.charAt(i);
  return (hashValue % tableSize);
}
```

4.2 Forskjellige typer hash-tabeller

- **Seprate chaining** er en hash-tabell hvor hver indeks peker til en liste av elementer.
- **Probing** er rommet mellom hver indeks.
  - *Lineær probing*; intervallene er satt (normalt 1).
  - *Kvadratisk probing*; intervallene øker med å legge til et kvadratisk polynom ved startverdi gitt av hashberegningen.
  - *Dobbel hashing*; intervallene er gitt av en annen hash-funksjon.

I tabell 4.2 har vi forskjellige hash-tabeller på tallene {4371, 1323, 4199, 4344, 9679, 1989} som skal settes inn i en tabell på størrelse 10, og med hash-funksjon `H(X) = X mod 10`.

Index	Linear probing	Quadratic probing	Separate chaining
0	9679	9679	
1	4371	4371	
2	1989		
3	1323	1323	1323 → 6173
4	6173	6173	4344
5	4344	4344	
6			
7			
8		1989	
9	4199	4199	4199 → 9679 → 1989

Tabell 4: Eksempel på hashing.

Lineær probing her tar man tallet og setter inn i hash-funksjonen. Så for første tall (4371) vil indeksen bli 1. Hvis indeksen er opptatt, så plusser man på 1.

Kvadratisk probing er så og si det samme, men i stedet for å legge til 1 hvis indeksen er opptatt, så legger man på $1^2, 2^2, 3^2, \ldots$

Dobbel probing her lager vi en ny hash-funksjon når indeksen krasjer. Da tar man et tall P, som er det største primtallet som er mindre enn tabellstørrelsen. Dette brukes til å lage en ny hash-funksjon `H(X) = R - (X mod R)`.

Separate chaining her legges det nye tallet på i en liste om indeksen er opptatt fra før.

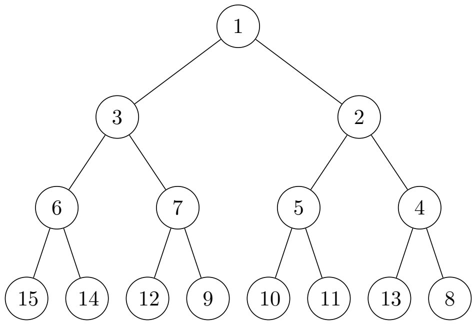
5 Heap

En **binær heap** er et komplett binærtre, hvor barna alltid er større eller lik sine foreldre. Og et komplett binærtre har følgende egenskaper:

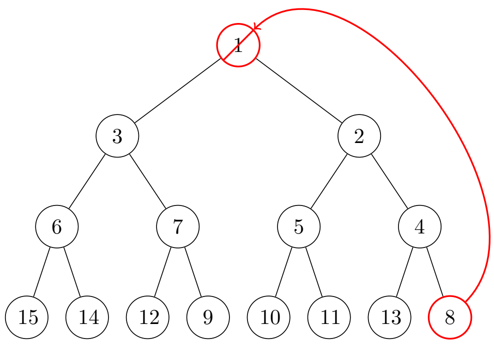
- Treet vil være i perfekt balanse.
- Løvnoder vil ha høydeforskjell på maksimalt 1.
- Treet med høyden h har mellom 2^h og $2^{h+1} - 1$ noder.
- Den maksimale høyden på treet vil være $\log_2(n)$.

5.1 Eksempel på sletting i min-heap

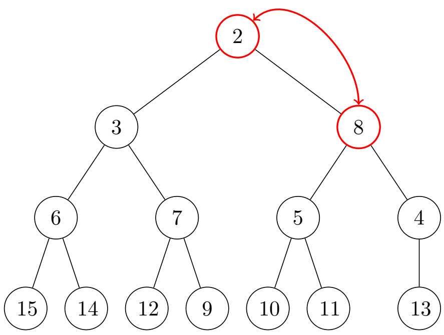
Skal gjøre operasjonen `deleteMin()` på følgende heap $\times 3$.

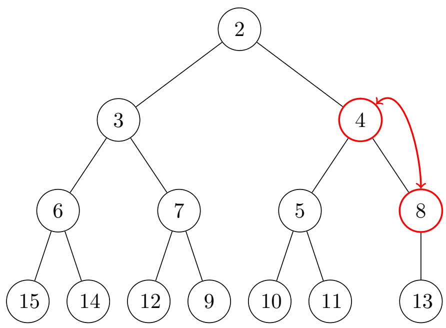


Begynner med å slette rot noden 1. Og flytter siste element (8) opp til rot posisjon.



Får da dette resultatet. Og vi må nå flytte 8 ned til riktig posisjon.





6 Grafer

Definisjon 6.1: Graf *En **graf** $\mathcal{G}$ består av en ikke-tom mengde noder $\mathcal{V}$ og en mengde kanter $\mathcal{E}$, slik at enhvet kant forbinder nøyaktig to noder med hverandre eller en node med seg selv.*

6.2 Terminologi

Definisjon 6.3: Vektet *En graf er vektet dersom hver kant har en tredje komponent. En verdi langs kanten.*

Definisjon 6.4: Sti *En sti gjennom grafen er en sekvens av noder* $v_1, v_2, v_3, \ldots, v_n$*, slik at* $(v_i, v_{i+1}) \in \mathcal{E}$ *for* $1 \leq i \leq n-1$.

Definisjon 6.5: Lengde *Lengden til stien er lik antall kanter på stien.*

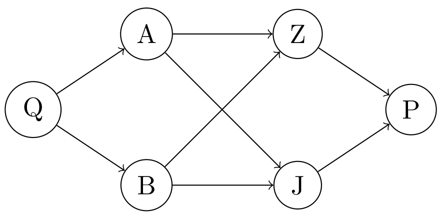
Definisjon 6.6: Løkke *Ei løkke i en rettet graf er en vei med lengde* ≥ 1*, slik at* $v_1 = v_n$*. Løkken er enkel dersom stien er enkel.*

Definisjon 6.7: Asyklisk *En rettet graf er asyklisk dersom den ikke har noen løkker. DAG (Directed, Asyclic Graph).*

Definisjon 6.8: Sammenhengende *En urettet graf er sammenhengende dersom det fins en sti fra hver node til alle andre noder.*

6.9 Topologisk sortering

Definisjon 6.10: Topologisk sortering *En topologisk sortering (topologisk orden) av en rettet graf er en lineær ordning av nodene, slik at for hver rettet kant* $\langle u, v \rangle$ *fra node* u *til node* v*, så kommer* u *før* v *i ordningen.*



Figur 5: Eksempelfigur for topologisk sortering.

I figur 5 har vi følgende lovlige topologiske sorteringer:

- Q, A, B, J, Z, P
- Q, B, A, J, Z, P
- Q, A, B, Z, J, P
- Q, B, A, Z, J, P

6.10.1 Algoritmer

De vanligste algoritmene for topologisk sortering har lineær kjøretid i antall noder, pluss antall kanter; $O(|\mathcal{V}| + |\mathcal{E}|)$. En av disse kommer det et eksempel på her.

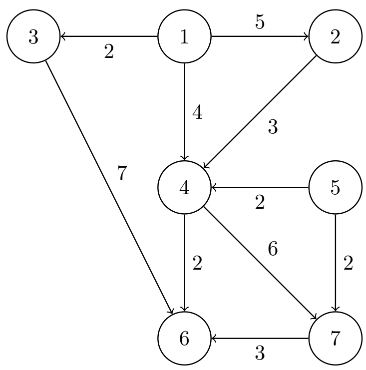
```
L <- empty list that will contain the sorted elements
S <- set of all nodes with no incoming edges

while S in non-empty do:
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do:
        remove edge e from the graph
        if m had no other incoming edges then:
            insert m into S
if graph has edges then:
    return error (graph has at least one cycle)
```

```
else
    return L (a topofically sorted order)
```

6.11 Dijkstra

1. For alle noder, sett avstanden fra startnoden s lik ∞. Merk noden som *ukjent*.
2. Sett avstanden fra s til seg selv lik 0.
3. Velg en ukjent node v med minimal avstand fra s og marker v som kjent.
4. For hver ukjente nabonode w til v: Dersom avstanden vi får ved å følge veien gjennom v, er kortere enn den gamle avstanden til s. Redusér avstanden til s for w og sett bakoverpekeren i w til v.
5. Akkurat som for uvektede grafer, ser vi bare etter potensielle forbedringer for naboer som ennå ikke er kjent.



Figur 6: Eksempelgraf for Dijkstra.

	Init			1			3		
V	known	dist.	from	known	dist.	from	known	dist.	from
1	F	0	0	T	0	0	T	0	0
2	F	–	0	F	5	1	F	5	1
3	F	–	0	F	2	1	T	2	1
4	F	–	0	F	4	1	F	4	1
5	F	–	0	F	–	0	F	–	0
6	F	–	0	F	–	0	F	9	3
7	F	–	0	F	–	0	F	–	0

	6			4			7		
V	known	dist.	from	known	dist.	from	known	dist.	from
1	T	0	0	T	0	0	T	0	0
2	F	5	1	F	5	1	F	5	1
3	T	2	1	T	2	1	T	2	1
4	F	4	1	T	4	1	T	4	1
5	F	–	0	F	–	0	F	–	0
6	T	9	3	T	6	4	T	6	4
7	F	–	0	F	10	4	T	10	4

	2		
V	known	dist.	from
1	T	0	0
2	T	5	1
3	T	2	1
4	T	4	1
5	F	–	0
6	T	6	4
7	T	10	4

Tabell 5: Tabell til Dikjstra-algoritmen på grafen i figur 6.

Algoritmen Her er pseudokoden for Dijkstra.

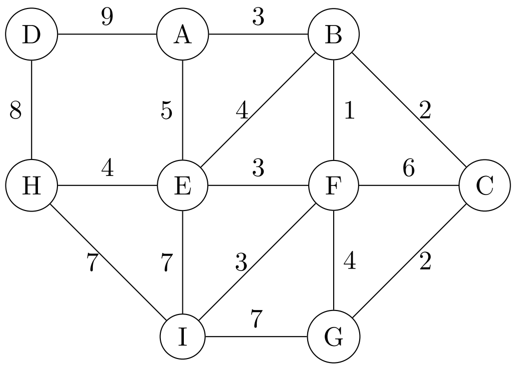
```
function Dijkstra (Graph, source):
    for each vertex v in Graph:
        distance[v] := infinity
        visited[v] := false;
        previous[v] := undefined;

    distance[source] = 0;
    insert source into Q;

    while Q is not empty:
        u := vertex in Q with smallest distance in distance[]
                    and has not been visited;
        remove u from Q;
        visited[u] := true;

        for each neighbour v of u:
            alt := distance[u] + dist_between(u,v);
            if alt < dist[v];
                dist[v] := alt;
                previous[v] := u;
                if !visited[v]:
                    insert v into Q;
    return distance;
end function;
```

6.12 Prim



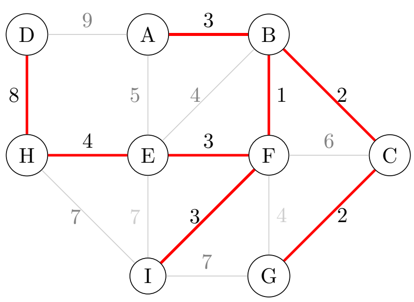
Figur 7: Eksempelgraf for Prim og Kruskal.

6.12.1 Framgangsmåte

1. Initialiser et tre med en enkelt node, valgt tilfeldig fra grafen.
2. Gro treet med én kant (av de kantene som går til en node som ikke er med i treet), finn den kanten med minst vekt og legg den inn i treet.
3. Gjenta steg 2 til alle nodene er i treet.

```
PRIM(G, w, r)
  for each u in G.V
    u.key = infinite
    u.parent = NIL
  r.key = 0
  Q = G.V
  while Q is not empty
    u = extract-min(Q)
    for each v in G.Adj[u]
      if (v in Q) and w(u,v) < v.key
        v.parent = u
        v.key = w(u,v)
```

Når man utfører Prims algoritme på grafen i figur 7, så blir resultatet følgende:



Figur 8: Eksempelgraf 7 etter Prims algoritme.

MERK: løsningen i figur 8 er *ikke* unik. Eneste måten man kan få et unikt resultat er om alle vektene på kantene er unike. Det samme gjelder for Kruskal.

	avs.	fra
A	0	–
B	3	A
C	2	B
D	8	H
E	3	F
F	1	B
G	2	C
H	4	E
I	3	F
	26	

Tabell 6: Tabell for Prims algoritme på figur 8.

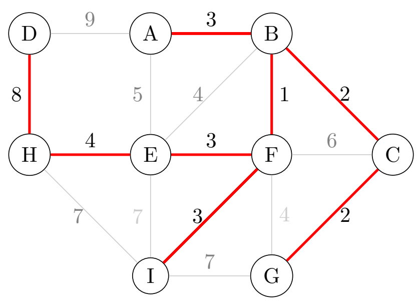
6.13 Kruskal

6.13.1 Framgangsmåte

- Lag en *skog* $\mathcal{F}$ (en sett med trær), hver hver node i grafen er et separat tre.
- Lag et sett $\mathcal{S}$ som inneholder alle kantene til grafen.
- Så lenge $\mathcal{S}$ ikke er tom og $\mathcal{F}$ ikke ennå er et spanning-tre:
  - Fjern en kant men lavest vekt fra $\mathcal{S}$.
  - Hvis kanten kobler to forskjellige trær, legg den i skogen slik at de to blir ett tre.

Når algoritmen er ferdig, så former skogen et minimum spanning-tre for grafen.

```
KRUSKAL (G):
  A = empty
  for each v in G.V:
    MAKE-SET(v)
  for each (u,v) ordered by weight(u,v), increasing:
    if FIND-SET(u) not in FIND-SET(v):
      A = A union {(u,v)}
      UNION(u,v)
  return A
```



Figur 9: Eksempelgraf 7 etter Kruskals algoritme.

Kant	FB	BC	CG	FI	FE	BA	EH	HD	
Vekt	1	2	2	3	3	3	4	8	⇒26

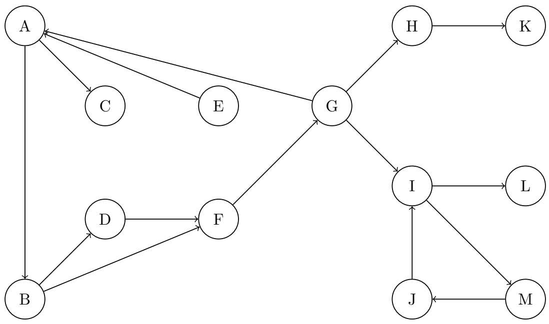
6.14 Dybde-først

Dette er klassisk graf-traversering, generalisering av prefiks traversering for trær. Gitt start node v: rekursivt traverser alle nabonodene.

```
public void depthFirstSearch(Node v)
  v.marked = true;
  for <each neighbour w to v>
    if (!w.marked)
      depthFirstSearch(w);
```

6.15 Biconnectivity

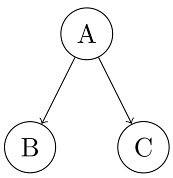
Definisjon 6.15.1. *En sammenhengende urettet graf er* ***bi-connected*** *hvis det ikke er noen noder som ved fjerning gjør at grafen blir usammenhengende. Slike noder heter cut-vertices eller articulation points.*



Figur 10: Eksempelgraf for bi-connectivity.

Grafen i figur 10 er *ikke* bi-connected, fordi nodene A, I, G og H er articulation points.

6.15.1 DFS spanning-tre av figur 10



6.16 Strongly connected components (SCC)

Definisjon 6.16.1. *Gitt en rettet graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. En* ***strongly connected component*** *av $\mathcal{G}$ er et maksimalt sett av noder $U \subseteq \mathcal{V}$. For alle $u_1, u_2 \in U$ har vi at $u \to^* u_2$ og $u_2 \to^* u_1$.*

7 Kombinatorisk søk

8 Rekursjon

9 Tekstalgoritmer

9.1 Brute force

Brute force metoden sjekker en og en karakter i teksten med nålen, og flytter med 1 om det er mismatch.

9.1.1 Analyse

Worst case så får vi mismatch $(n-m)$ ganger og suksess $(n-m)+1$ ganger. Totale sammenligninger er $((n-m)+1 \times m)$ som gir kjøretid $O(n^2)$.

9.2 Boyer Moore

- Først må man lage ≪bad match table≫.
- Sammenligne nålen med teksten, starter med karakteren lengst til høyre i nålen.
- Hvis mismatch, flytt nålen fram iht. verdien i tabellen.

9.2.1 Eksempel: Bad character shift

Pattern (nål) tooth

Tekst trusthardtoothbrushes

1. Konstruere ≪bad match table≫.

```
value = length - index - 1
```

Alle andre bokstaver har verdi lik lengden.

	T	O	O	T	H
index:	0	1	2	3	4

Lengden på denne nålen er 5. Finner verdiene ved å bruke `value = length - index - 1`.

T =	$5-0-1$	$=4$	
O =	$5-1-1$	$=3$	
O =	$5-1-2$	$=2$	Erstatter her forrige verdi av O med ny verdi til O.
T =	$5-3-1$	$=1$	Erstatter her forrige verdi av T med ny verdi til T.
H =	5		Verdien skal ikke være mindre enn 1. Får da verdi lik lengden.

Bokstav	T	O	H	$*$
Verdi	1	2	5	5

	T	R	U	S	T	H	A	R	D	T	O	O	T	H	B	R	U	S	H	E	S
1.	T	O	O	T	H																
2.		T	O	O	T	H															
3.							T	O	O	T	H										
4.									T	O	O	T	H								
5.										T	O	O	T	H							

I steg **1.** får vi mismatch på H, fordi den ikke er det samme som T. Da må vi slå opp i tabellen på den bokstaven som ***vi møter i teksten***, og hoppe fram tilsvarende antall steg. I første tilfellet skal vi hoppe 1 plass fram (for 1 er verdien til T).

Sjekker på nytt i steg **2.**, her får vi match på T og H, men mismatch på O. Da hopper vi S-plasser frem. Siden S ikke er med i tabellen vår (eller den er med som $*$, som er alle andre bokstaver), så hopper vi 5 plasser frem.

I steg **3.** får vi mismatch på H mot O, og må hoppe O-plasser frem—altså 2. I steg **4.** er det mismatch mellom H og T, vi hopper T-plasser frem—altså 1. Og vips! Så har vi funnet nålen i teksten vår!

Analyse Worst case er det samme som brute force. Input tekst 1^n kjører n ganger, og nål $011\dots1$ kjører m ganger. Dette gir $O(nm)$. Best case har input tekst 1^n og nål 0^m, som gir $O(n/m)$. Average case er $O(m/|\Sigma|)$, raskere enn brute force.

9.2.2 Eksempel: Good suffix shift

Konstruere *good suffix table* for nålen `TCCTATTCTT`.

1. Sjekker først *ikke*-`T`, det er to shift før man finner dette.
`TCCTATTCTT`
`--TCCTATTCTT`

2. Så sjekker man *ikke*-`TT`. Dette finner man etter ett shift.
`TCCTATTCTT`
`-TCCTATTCTT`

3. For å finne *ikke*-`CTT` må vi flytte 3 steg; til `ATT`.
`TCCTATTCTT`
`---TCCTATTCTT`

4. Så kommer vi til *ikke*-`TCTT`. Denne eksisterer ikke, MEN om nålen har lik suffix som prefiks (her `T` som start og slutt), så flytter vi bare fram til prefiksen. Da må vi flytte nålen 9 hakk, selv om lengden er 10.
`TCCTATTCTT`
`---------TCCTATTCTT`
Alle de neste shiftene vil også være dette, fordi de får ingen andre treff i nålen.

Vi får da følgende good suffix table:

index	mismatch	shift
0	T	2
1	TT	1
2	CTT	3
3	TCTT	9
4	TTCTT	9
5	ATTCTT	9
6	TATTCTT	9
7	CTATTCTT	9
8	CCTATTCTT	9
9	TCCTATTCTT	9

Tabell 7: Eksempel på good suffix table.

	Tid			**Rom**
Algoritme	Best	Average	Worst	Worst
Quicksort	$O(\log(n))$	$O(\log(n))$	$O(n^2)$	$O(n)$
Mergesort	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(1)$
Heapsort	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bucket sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(nk)$
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

Tabell 8: Kompleksiteten til forskjellige sorteringsalgoritmer.

10 Sortering

10.1 Quicksort

Quicksort er en *divide and conquer* algoritme. Den deler først en stor array inn i to mindre sub-arrayer: de mindre og de større verdiene. Sorterer så sub-arrayene rekursivt.

1. Velg et element, kalt *pivot*, fra arrayen.
2. Reorder arrayen slik at alle elementene med vardier mindre enn pivot kommer før pivot, og alle elementer større enn pivot kommer etter. Pivot er på sin sluttplass. Dette kalles *partisjonering*.
3. Rekursivt gjenta stegene over på sub-arrayene med mindre og større verdier.

```
quicksort(A, i, k):
    if i < k:
        p := partition(A, i, k)
        quicksort(A, i, p-1)
        quicksort(A, p+1, k)

partition(array, left, right)
    pivotIndex := choosePivot(array, left, right)
    pivotValue := array[pivotIndex]
    swap array[pivotIndex] and array[right]
    storeIndex := left
    for i from left to right - 1
        if array[i] < pivotValue
            swap array[i] and array[storeIndex]
```

```
        storeIndex := storeIndex+1
    swap array[storeIndex] and array[right]
    return storeIndex
```

10.2 Mergesort

Mergesort er en *divide and conquer* algoritme.

1. Del den usorterte listen i n sublister, hvor hver inneholder *ett* element.
2. Smelt sammen sublistene for å lage nye sorterte sublister til det er bare ei liste igjen. Dette er den sorterte listen.

10.3 Heapsort

Heapsort er en *in-place* algoritme, men den er ikke en *stable sort*.

Først bygger man en heap ut av dataene. Ofte plassert i en array etter kravene for et komplett binært tre. Roten er lagret på index 0, hvis `i` er indeksen til *denne* noden, så:

```
iParent     = floot((i-1)/2)
iLeftChild  = 2*i+1
iRightChild = 2*i+2
```

I det andre steget lager man en sortert array ved gjentatte ganger fjerne største element fra heapen (roten), og sette det inn i arrayen. Heapen er oppdatert etter hver sletting, for å opprettholde kravene. Når alle elementene er slettet fra heapen, er resultatet en sortert array.

10.4 Bubble sort

Bubble sort, også kalt sinking sort, er en sammenlignings-sorteringsalgoritme. Den er for treig for all praktisk bruk, til og med treigere enn insertion sort.

```
bubblesort(A : list of sortable items)
    n = length(A)
    do:
        swapped = false
        for i = 1 to n-1 inclusive do
            if A[i-1] > A[i] then
               swap(A[i-1], A[i])
               swapped = true
    while not swapped
```

10.5 Insertion sort

Enkel sorteringsalgoritme, som ikke er særlig effektiv på større lister. Da er algoritmer som quicksort, heapsort osv å foretrekke.

```
for i = 1 to length (A)
    x = A[i]
    j = i
    while j > 0 and A[j-1] > x
        A[j] = A[j-1]
        j = j-1
    A[j] = x
```

10.6 Selection sort

Selection sort er en *in-place* sammenlignings-algoritme.

La $\mathcal{L}$ være et ikke-tomt sett og $f : \mathcal{L} \to \mathcal{L}$, slik at $f(\mathcal{L}) = \mathcal{L}'$ hvor:

1. $\mathcal{L}'$ er en permutasjon av $\mathcal{L}$,
2. $e_i \leq e_{i+1}$ for alle $e \in \mathcal{L}'$ og $i \in \mathbb{N}$,
3. $f(\mathcal{L}) = \begin{cases} \mathcal{L} & if|\mathcal{L}| = 1 \\ \{s\} \cup f(\mathcal{L}_s), & \text{ellers} \end{cases}$
4. s er det *minste elementet* i $\mathcal{L}$, og
5. $\mathcal{L}_s$ er settet med elementer av $\mathcal{L}$ uten instansen av det minste elementet fra $\mathcal{L}$.

Algoritmen finner minste verdi, bytter denne med verdien i første posisjon, og repeterer disse stegene for resten av listen.

10.7 Bucket sort

Bucket sort, eller bin sort, er en *distribution sort* algoritme.

1. Sett opp en array med tomme «bøtter».
2. **Scatter**: gå over den orginale arrayen, og putt hvert objekt i sin bøtte.
3. Sortert hver ikke-tomme bøtte.
4. **Gather**: besøk alle bøttene etter orden, og putt alle elementer tilbake i den orginale arrayen.

```
function bucketSort(array, n)
    buckets <- new array of n empty lists
    for i = 0 to (length(array) - 1) do
        insert array[i] into buckets[mbits(array[i], k)]
    for i = 0 to n - 1 do
        nextSort(buckets[i]);
    return the concatenation of buckets[0],...,buckets[n-1]
```

10.8 Radix sort

Radix sort er en *non-comparative* integer sorterings algoritme. Er i samme familie som bucket sort.

1. Ta det minst signifikante tallet av hver nøkkel.
2. Gruppér nøklene basert på dette tallet.
3. Gjenta til grupperingsprosessen med hvert høyere signifikante bit.

Eksempel Orginal, usortert lise: 170, 45, 75, 90, 802, 2, 24, 66

- Sortere på minst signifikante (1-er plassen): 170, 90, 802, 2, 24, 45, 75, 66
- Sortere på 10-er plassen: 802, 2, 24, 45, 66, 170, 75, 90
- Sortere på 100 plassen: 2, 24, 45, 66, 75, 90, 170, 802

Referanser

[1] Unknown, *RedBlackBST.java.* algs4.cs.princeton.edu, http://algs4.cs.princeton.edu/33balanced/RedBlackBST.java.html

[2] Unknown, *Boyer Moore Horspool Algorithm.* https://www.youtube.com/watch?v=PHXAOKQk2dw

[3] Unknown, *Sorting algorithm.* http://en.wikipedia.org/wiki/Sorting_algorithm

[4] Insitutt for informatikk, Universitetet i Oslo, *INF2220–Algoritmer og datastrukturer.* http://www.uio.no/studier/emner/matnat/ifi/INF2220/

Tabeller

Figurer