Formel Samling

Udgave 1.0

Frederik Hansen

11. jan., 2024

Indhold

KAPITEL 1

Algorithmer

1.1 Hashmaps / Hash tables

Hashmaps er en datastruktur hvor posistionen af dataen i memory er beskrevet gennem en funktion.

1.1.1 key

det data der beskriver inputtet i din hashing funktion

1.1.2 index mapping (trivial hashing)

beskriver hashing functionen f(x)=x altså er din key lig med posistionen i memory

1.1.3 collision

hvis en funktion retunere den samme område i memory ved 2 forskellige keys

1.1.4 seperate chaining

en måde at løse collisions problemer ved at erstate indgangen i memory med en liste som indeholde begge værdier der skal være på denne key.

1.1.5 Open addressing

open adressing betyder at din indang i hastabellen ikke behøver at være det præcise resultat af hash functionen. Når man ændre på hvilken posistion noget ligger på kalder man det probing

VIGTIGT Når der laves open adressing skal den originale key gemmes sammen med data'en således at programmet kan chekke om der skal laves probing eller ej.

linear probing

en probing methode hvor man bare tager den næste frie plads i memory

quadratic probing (mid-square method)

her søger man igennem index+x^2 hvor x stiger med 1 indtil man finder en plads

dobbelt hashing

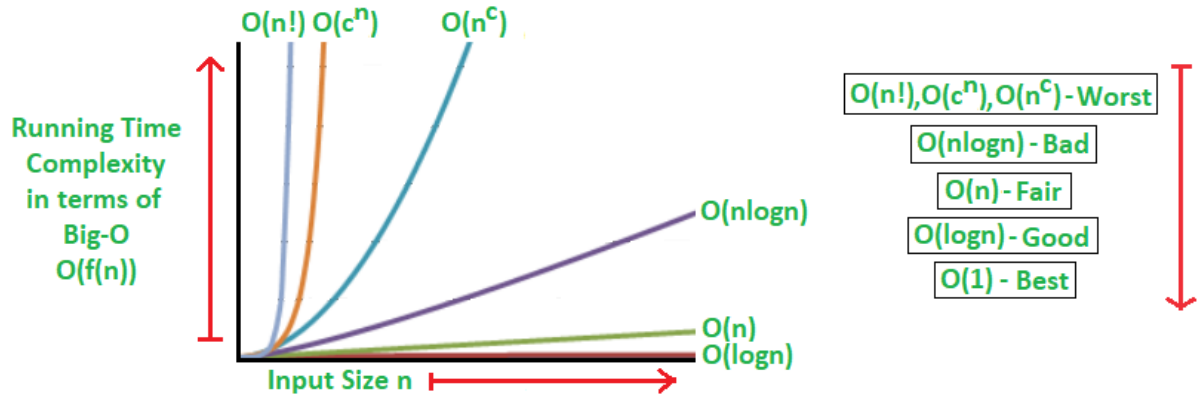
ved dobbelt hashing bruges en anden hashing function som sådan til at lave probing. dette gøres ved formlen h1(key)+h2(key)*x hvor x stiger med 1 indtil en plads er fundet.

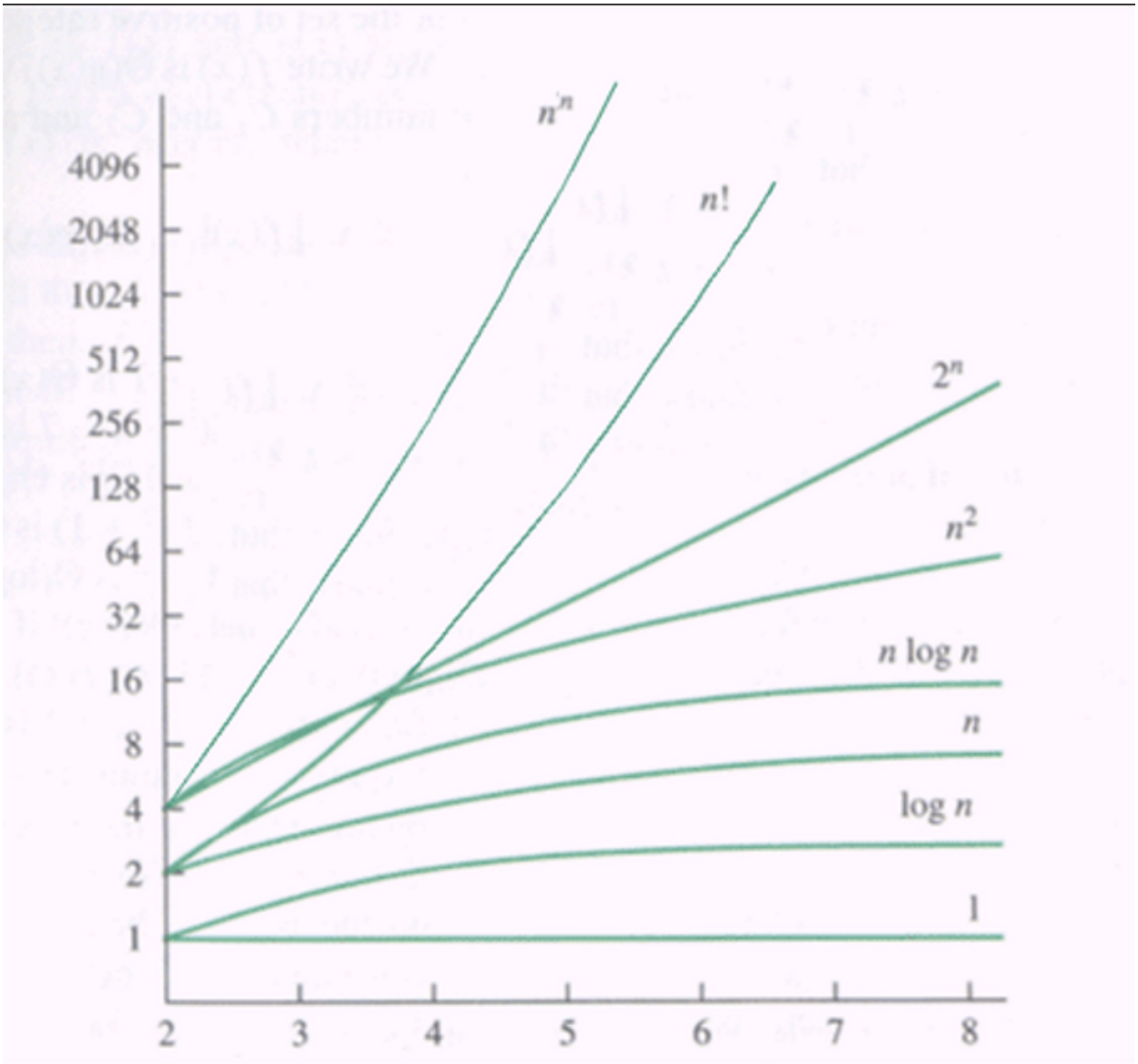
1.1.6 universal hashing

Når man laver universal hashing vælges en tilfældig hashing funktion hver gange et nyt table laves, på denne måde kan en malicius actor ikke vide hvilken hashing algoritme er valgt.

1.2 Tids og størrelses kompleksitet

1.2.1 big O thetha and omega notation





S.No	Big O	Big Omega ()	Big Theta ()
1.	It is like (<=) rate of growth of an algorithm is less than or equal to a specific value.	It is like (>=) rate of growth is greater than or equal to a specified value.	It is like (==) meaning the rate of growth is equal to a specified value.
2.	The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. Asymptotic upper bound is given by Big O notation.	The algorithm's lower bound is represented by Omega notation. The asymptotic lower bound is given by Omega notation.	The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.
3.	Big O – Upper Bound	Big Omega () – Lower Bound	Big Theta () – Tight Bound
4.	It is define as upper bound and upper bound on an algorithm is the most amount of time required (the worst case performance).	It is define as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).	It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.
5.	Mathematically: Big Oh is 0 <= f(n) <= Cg(n) for all n >= n0	Mathematically: Big Omega is 0 <= Cg(n) <= f(n) for all n >= n0	Mathematically – Big Theta is 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0

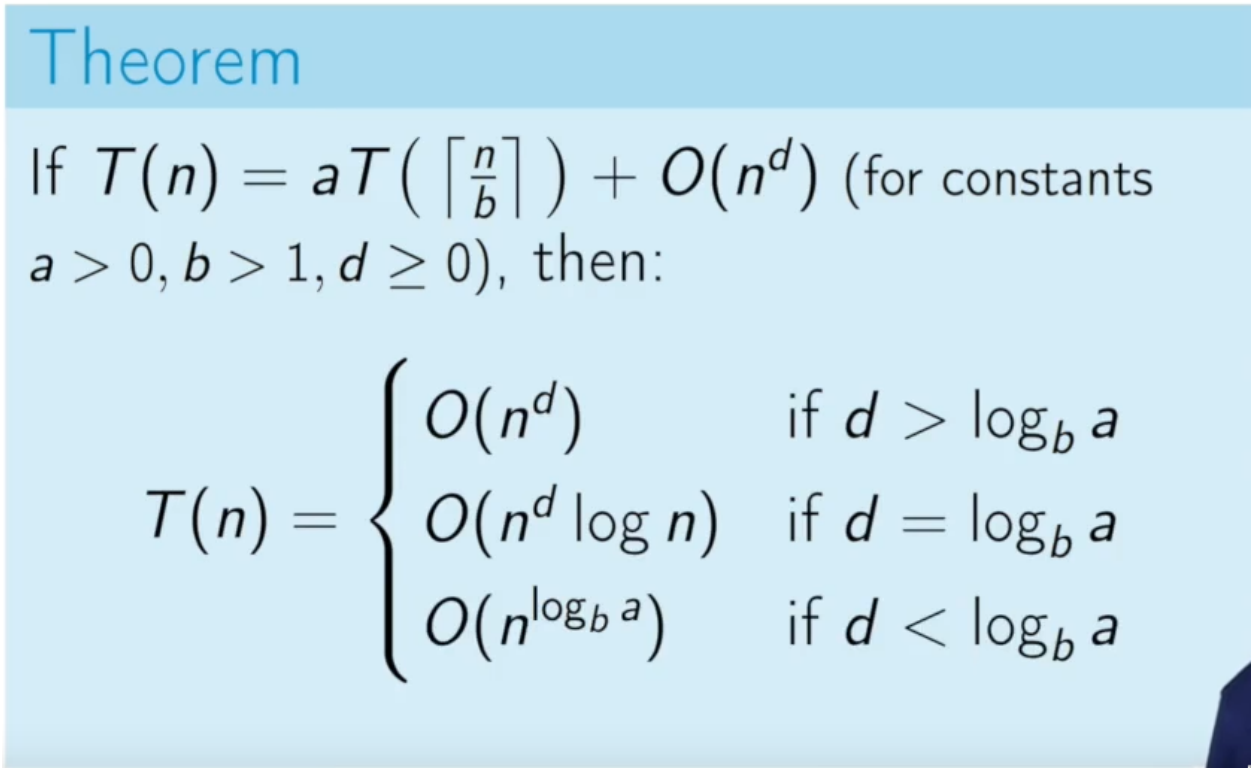
1.2.2 little o theta and omega notation

lille notation har som sådan de samme regler som big notation, med den undtagelse at definitaionen skal gælde for alle værider af n (altså alle steder på funktionen) og ikke kun i et enkelt scenarie.

1.2.3 rekursive funktioner

rekursive funtioner er funktioner der indeholder sig selv. fx: t(n) = 1+t(2/n)

1.2.4 Master Theorem



der findes også en udvidet master Theorem

ud fra formlen

T(n) = aT(n/b)+O(n^k*log_P(n))

her gælder:

if a > bk, then T(n) = (nlogba) if a = bk, then (a) if p > -1, then T(n) = (nlogba logp+1n) (b) if p = -1, then T(n) = (nlogba loglogn) (c) if p < -1, then T(n) = (nlogba)

if a < bk, then (a) if p >= 0, then T(n) = (nk logpn) (b) if p < 0, then T(n) = (nk)

1.2.5 algoritme cheat sheet

Algorithm	Best Case	Average Case	Worst Case
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Tree Sort	O(nlogn)	O(nlogn)	O(n^2)
Radix Sort	O(dn)	O(dn)	O(dn)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
Counting Sort	O(n+k)	O(n+k)	O(n+k)

1.3 Sorterings algorithmer

1.3.1 Heapsort

Max-Heapify sort seudo kode A er arrayet vores Heap er gemt i i er vores index i heap

```
Max-Heapify (A, i)

l = LEFT(i)
r = RIGHT(i)

if l <= A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r <= A.heap-size and A[r] > A[largest]
    largest = r
if largest != i
    exchange A[largest] and A[i]
    Max-Heapify(A,largest)
```

Min-Heapify sort seudo kode

```
Min-Heapify (A, i)

l = LEFT(i)
r = RIGHT(i)

if l <= A.heap-size and A[l] < A[i]
    smallest = l
else smallest = i
if r <= A.heap-size and A[r] < A[largest]
    smallest = r
if smallest != i
    exchange A[smallest] and A[i]
    Min-Heapify(A,smallest)
```

1.3.2 Quick sort

Quick sort is a divide and conquor algoritme

seudo kode: A er det array der skal sorteres p er vores første indgang i arrayet r er den sidste indgang i arrayet

```
Quicksort(A,p,r)
if p < r
    q = Partition(A,p,r)
    Quicksort(A,p,q-1)
    Quicksort(A,q+1,r)

Partition(A,p,r)
x=A[r]
i=p-1
for j = p to r-1
```

(fortsætter på næste side)

(fortsat fra forrige side)

```
    if A[j] <= x
        i = i+1
        exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1
```

1.3.3 Count sort

```
Counting sort(A,n,k)
let B[1:n] and C[0:k]
for i = 0 to k
    C[i] = i
for j = 1 to n
    C[A[j]] = C[A[j]] + 1
```

1.3.4 Radix sort

in radix sort you sort the numbers by the smallest digit first.

hvis alle tal i arrayet har lige mange digits er radix sort O(n)

1.3.5 Bucket sort

O(n^2) MEN! Big Theta på (n) **SHITTY SUDO CODE **

!(shitty code)[https://i.imgur.com/1U8EpXe.png] Legit den her kode er så dårligt skrevet at den legit ikke giver mening. så her er den lige i C++ /

```
// Function to sort arr[] of
// size n using bucket sort
void bucketSort(float arr[], int n)
{

    // 1) Create n empty buckets
    vector<float> b[n];

    // 2) Put array elements
    // in different buckets
    for (int i = 0; i < n; i++) {

        // Index in bucket
        int bi = n * arr[i];
        b[bi].push_back(arr[i]);
    }

    // 3) Sort individual buckets
    for (int i = 0; i < n; i++)
```

(fortsætter på næste side)

(fortsat fra forrige side)

```
        sort(b[i].begin(), b[i].end());

    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}
```

1.4 programmerings paradimer

1.4.1 Dynamic programming

Dynamic programming er er programming uden recursion. Dette er ofte opnået ved at loope igennem en funktion flere gange og gemme resultaterne. Ofte resultere dette i en hurtigere algoritme da man kan gemme resultatet for flere funktions kald som ville give det samme resultat istedet for at køre funktionen flere gange

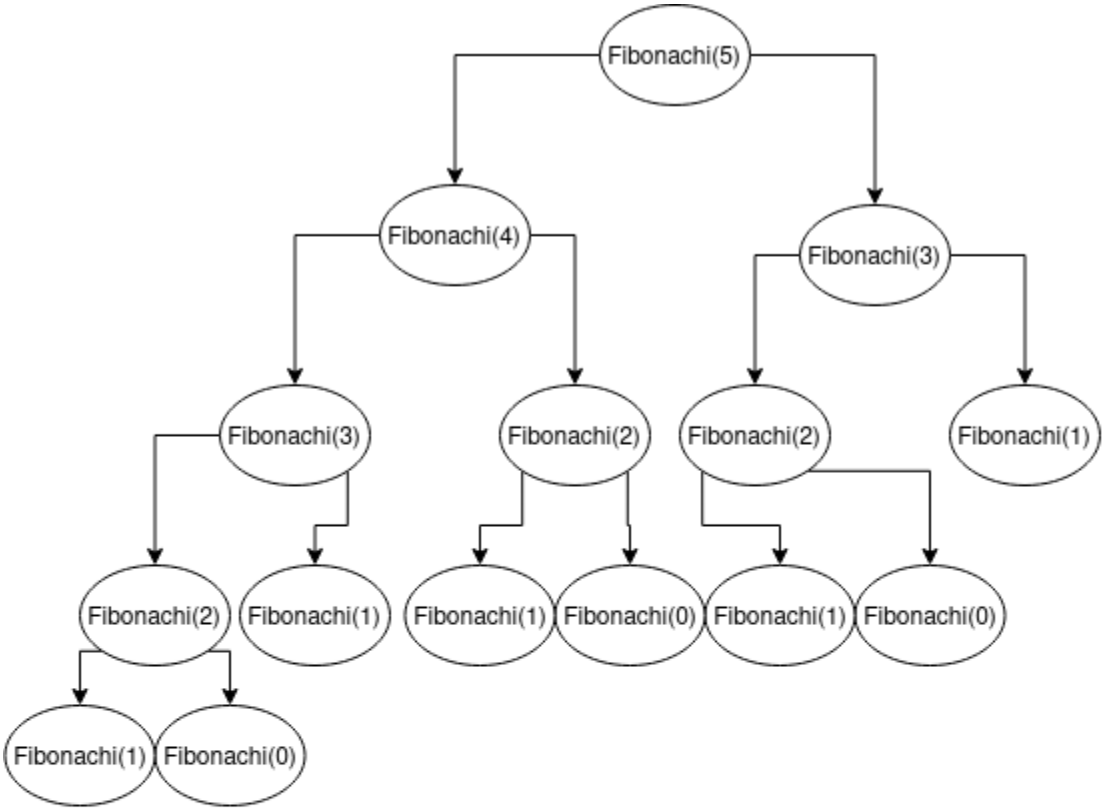
subproblem graf (afhængligheds graf)

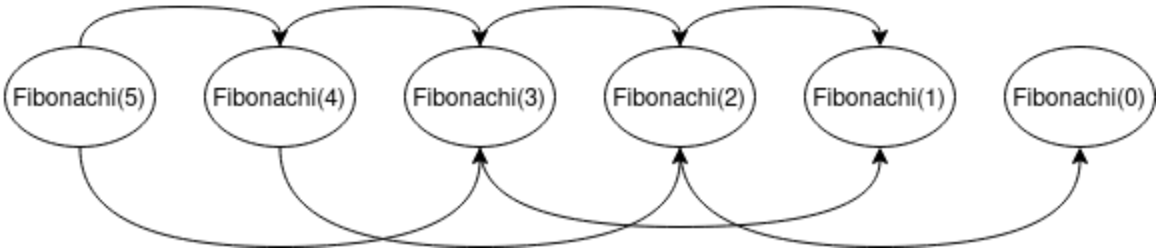
et recursivt problem kan beskrives som et problem med flere subproblemer. den originale funktion afhænger af løsningen af disse subproblemer. En subproblem graf viser os hvilket subproblem der ikke afhænger af nogle andre, og dermed hvilen rækkefølge subproblemerne skal løsses i.

fx hvis vi havde koden.

```
function fibonachi (int n){
    if (n == 0){
        return 0;
    }
    if (n == 1){
        return 1;
    }
    return fibonachi(n-1)+fibonachi(n-2)
}
```

hvis n = 5 fåes subproblem grafen:


Fibonachi(5)
Fibonachi(4)
Fibonachi(3)
Fibonachi(3)
Fibonachi(2)
Fibonachi(2)
Fibonachi(1)
Fibonachi(2)
Fibonachi(1)
Fibonachi(1)
Fibonachi(0)
Fibonachi(1)
Fibonachi(0)
Fibonachi(1)
Fibonachi(0)


Fibonachi(5)
Fibonachi(4)
Fibonachi(3)
Fibonachi(2)
Fibonachi(1)
Fibonachi(0)

1.4.2 Greedy algorithms

En greedy algoritme der »gårdigt« vælger hvad der virker som den bedste løsning lokalt, for forhåbenligt at komme til den bedste generelle løsning.

1.4.3 Divide and Conquer

Ved denne metode løses et eller uoverskueligt problem ved at dele det op, løse undederlene og derefter ligge resultatet sammen.

1.5 Binær træer

Binære træer er en data type hvor hver node kun har 2 børn. Dermed kan hvert »spring« beskrives med et Binært 1 eller 0

1.5.1 binær søge træ

tal som er større end forældrene vil atid ænde til venstre i et binær træ og tal som er større end forældre vil ende til højre i et binær træ. Dette gør det nemmere at søge i, men kan dog også være træet rimlig skævt.

søge algorithmen for disse træer er oplagt:

```
function binary_Search(int searchNumber){

    if(node = searchNumber){
        return nodeIndex
    }

    if(node > searchNumber){
    go left on binary trees
    }

    if( node < searchNumber) {
    go right on binary tree
    }

}
```

1.5.2 Max Heap / Min Heap

Heaps er binær træer med regler for strukturen af træet

Max Heap	Min Heap
forældre >= børn	forældre <= børn

1.5.3 Sort Rød træer

Sort røde træer deler nodes op i 2 farver med disse regler

1. hver node skal enten være rød eller sort
2. roden skal altid være sort
3. røde nodes må ikke have røde forældre eller børn
4. hver slut node »null« node skal være sort
5. der skal altid være den samme mængde sorte nodes mellem roden og null nodes.

1.6 Graf teori

1.7 Adjecency lists / Adjecency matrix

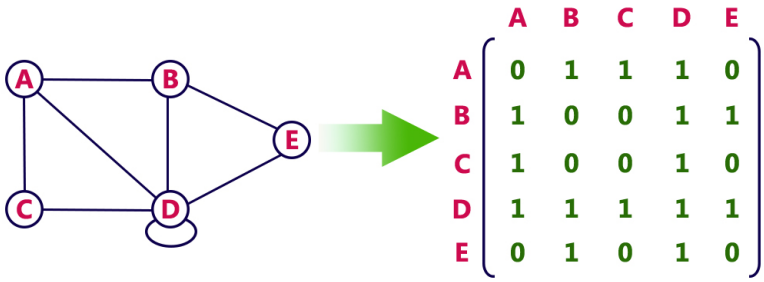
en adjacency list er en liste over hvilke nodes en anden node har edges til.

i graften til Ford-Fulkerson eksemplet ville det fx for v2 være

s	v1	v2	v3	v4	g
0	1	0	0	1	0

disse kan være vægtet. Det betyder bare at vægten af edgen skrives i stedet for 1

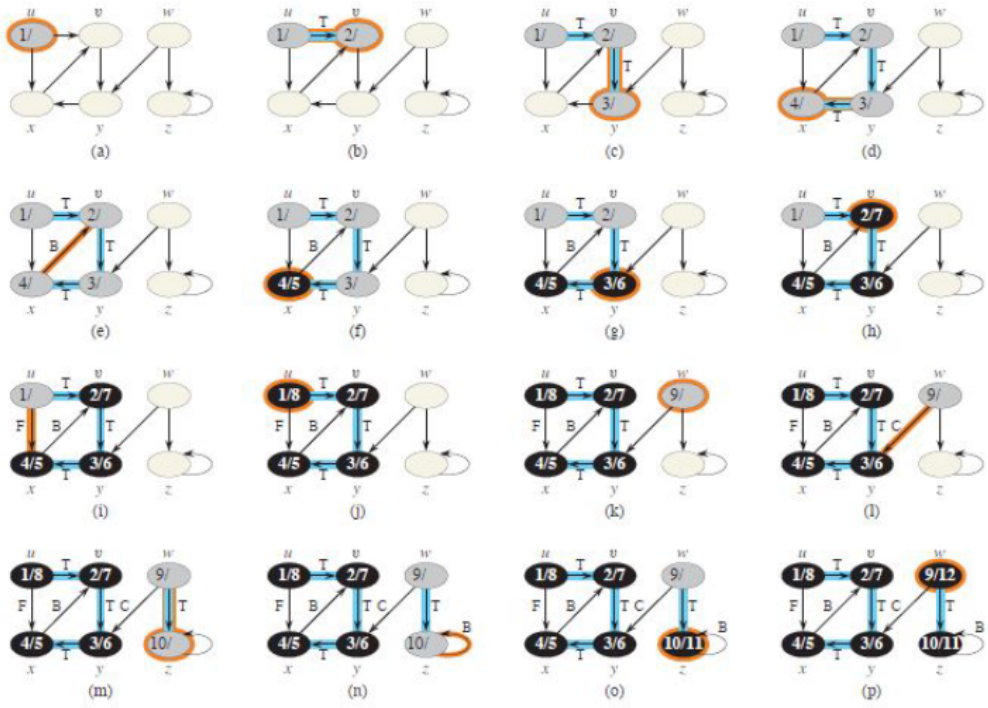
s	v1	v2	v3	v4	g
0	4	0	0	14	0



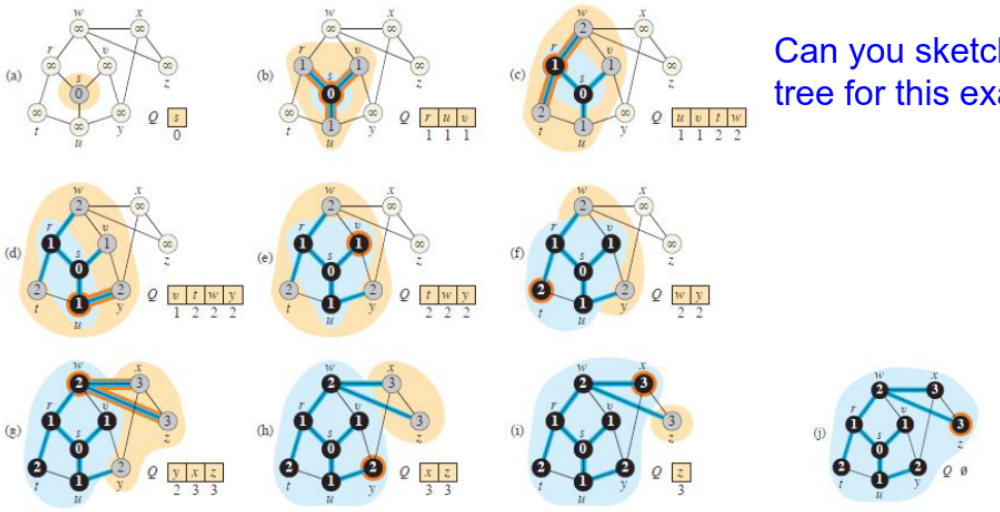
en Adjecency matrix er en matrice af disse lister

1.8 dybte og bredte søgning (BFS og DFS)

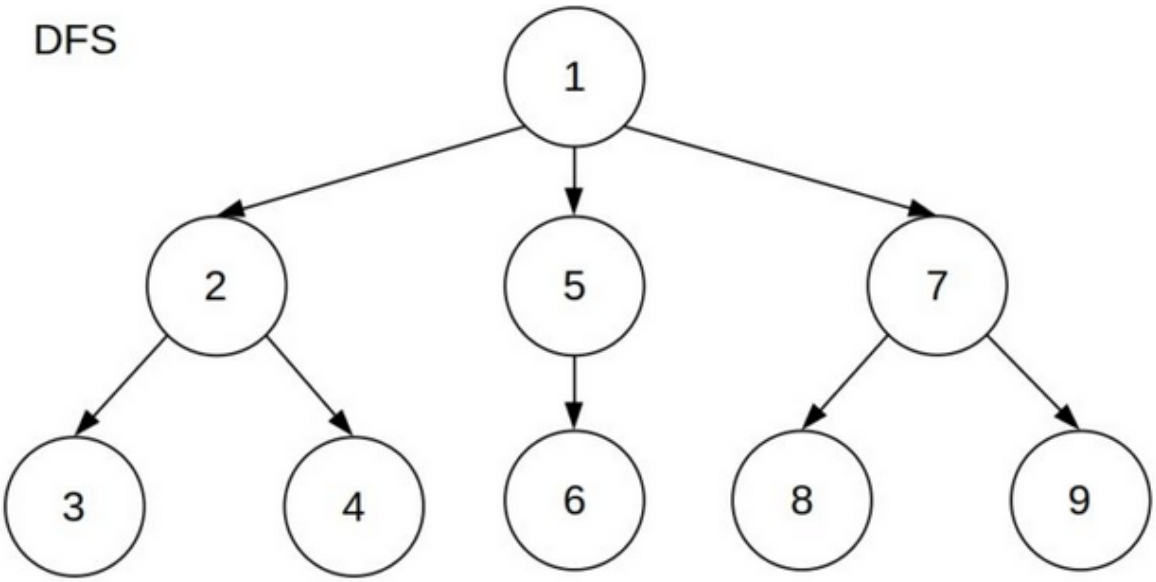
DFS - Example

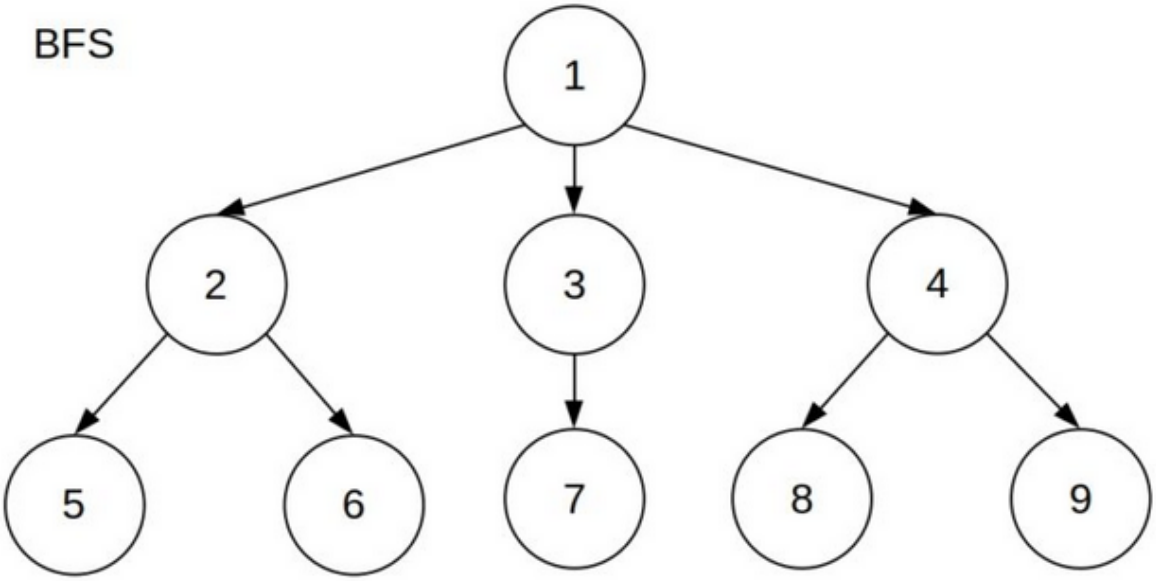


BFS - Example



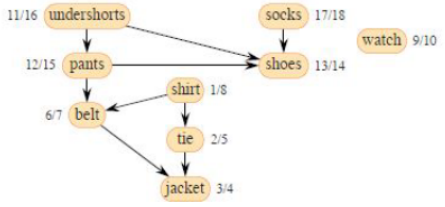
Can you sketch the BFS tree for this example?

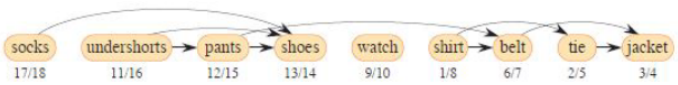




1.9 topological sort

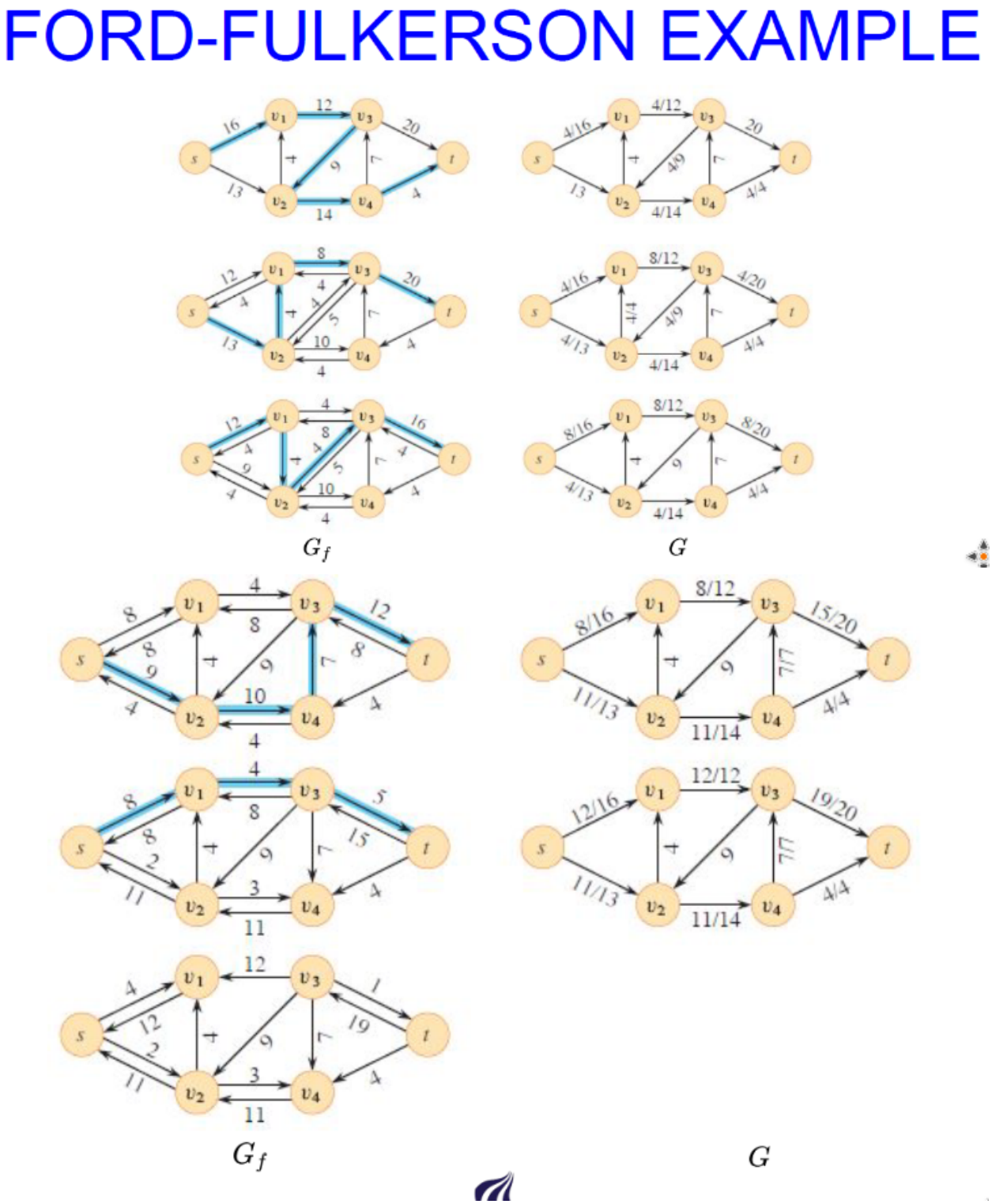
vælg en tilfældig node. Giv den tallet 1 gå til den næste node giv den tallet 2 sæt nodes i rækkefølge efter det største tal. nodes kan godt have flere tal





1.9.1 Ford-Fulkerson Algoritme / edmond karp

Bruges til at finde max netværks capacitet. Kan også give dig en iden om hvor dit netværk skal fobedres for den bedste effect

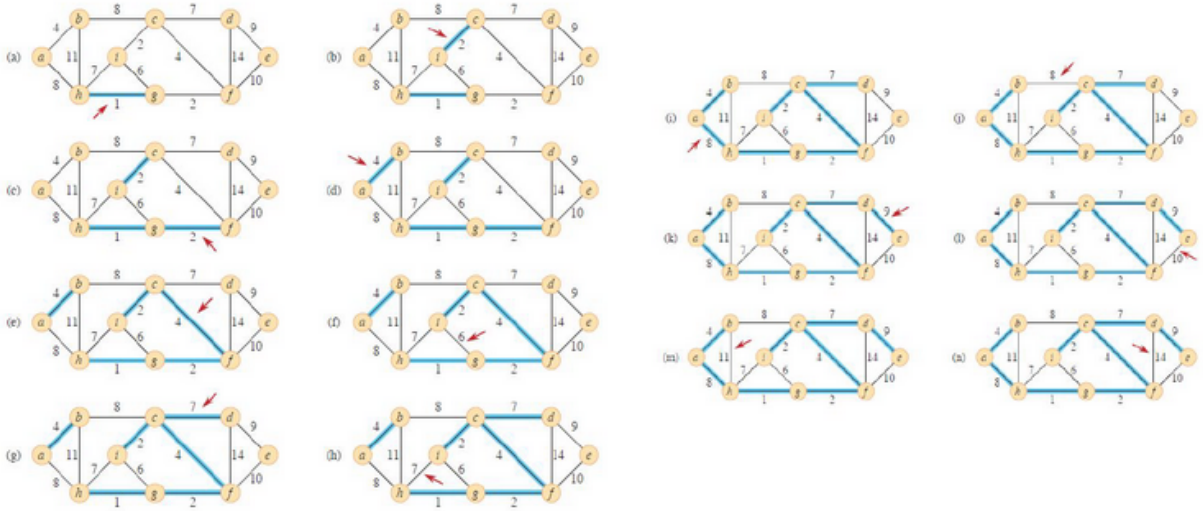


1.9.2 minimum spanning trees.

træer der har den mindste vægt fra 1 node ud til alle andre nodes i systemet.

kruskals Algoritme

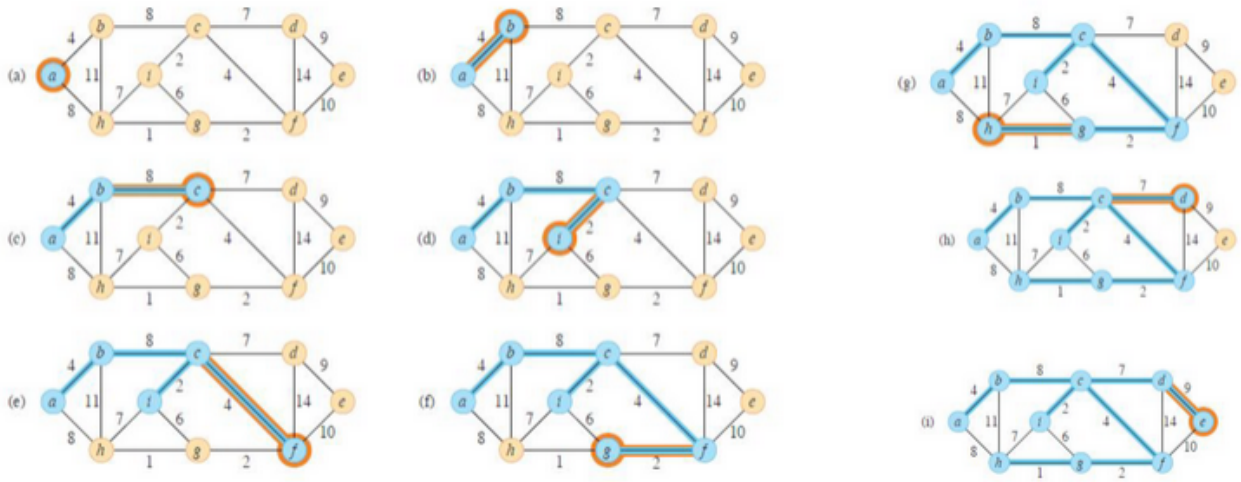
Kruskal's Algorithm



```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  create a single list of the edges in G.E
5  sort the list of edges into monotonically increasing order by weight w
6  for each edge (u, v) taken from the sorted list in order
7       if FIND-SET(u) ≠ FIND-SET(v)
8            A = A ∪ {(u, v)}
9            UNION(u, v)
10 return A
```

prims Algoritme

Prim's Algorithm

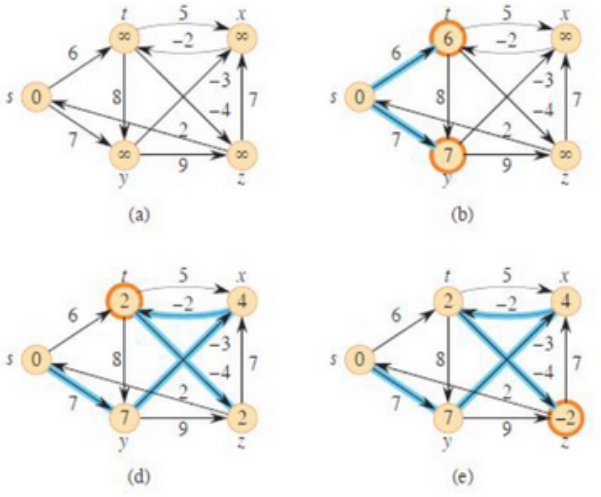


```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7      INSERT(Q, u)
 8  while Q ≠ ∅
 9      u = EXTRACT-MIN(Q)          // add u to the tree
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))
```

1.9.3 korteste vej (shortest path) Algoritmer

bellman-ford

The Bellman-Ford Algorithm - Ex

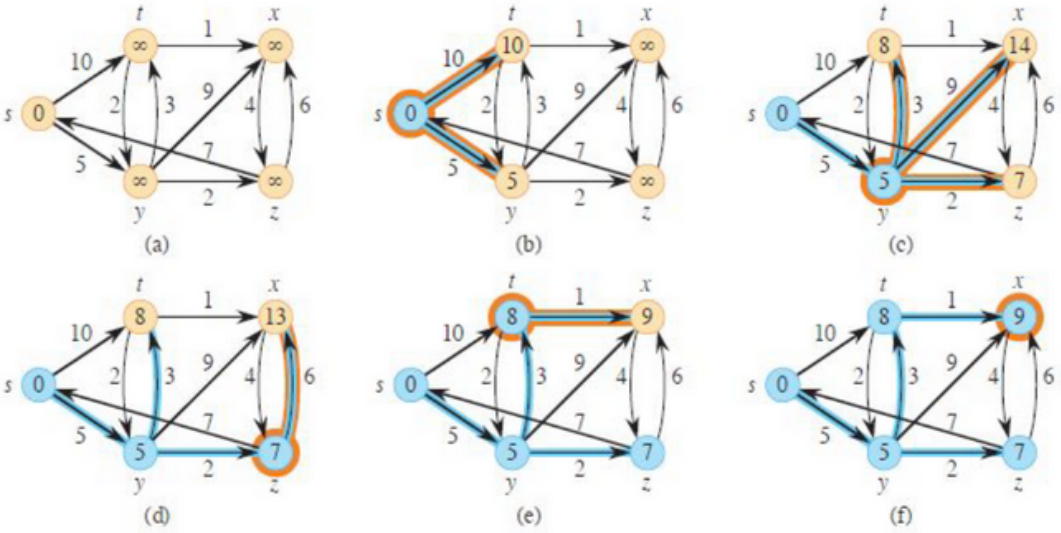



```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```


```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

dijsktra

Dijkstra's algorithm




```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5      INSERT(Q, u)
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      S = S ∪ {u}
 9      for each vertex v in G.Adj[u]
10          RELAX(u, v, w)
11          if the call of RELAX decreased v.d
12              DECREASE-KEY(Q, v, v.d)
```

1.10 korrekthed af algoritmer

1.11 Horners rule

Det her er en dum måde at regne polynomier del by del

a_0+x(a_1+x(a_2+x))

1.12